

CENTRO DE INVESTIGACION Y DE ESTUDIOS AVANZADOS DEL
INSTITUTO POLITECNICO NACIONAL

DEPARTAMENTO DE INGENIERIA ELECTRICA

SECCION COMPUTACION

X I N I X

SISTEMA OPERATIVO PARA COMPUTADORA PERSONAL

Tesis que presenta el Lic. Jorge Buenabad Chávez para obtener el grado de MAESTRO EN CIENCIAS en la especialidad de INGENIERIA ELECTRICA con opción en COMPUTACION.

Trabajo dirigido por el Dr. Manuel E. Guzmán Rentería.

SECRETARIA DE EDUCACION Y DE
CULTURA
BIBLIOTECA
INGENIERIA ELECTRICA

México D.F., Septiembre 1989.

21.10
RIF: _____
UIS: 31-1135 _____
HA: _____
CED: _____
\$ _____

A mis padres María Luisa y Jorge

A mis hermanos, Paty, Raúl y Luis

A Patricia Corrales Ayala

A mis amigos, compañeros y alumnos.

UNIVERSIDAD NACIONAL AUTÓNOMA DE MEXICO
FACULTAD DE INGENIERIA
LIBRERIA Y TIENDA
BIBLIOTECA
INGENIERIA ELECTRICA

Agradezco al Dr. Manuel Guzmán Rentería por
su confianza y paciencia en este proyecto,

a Andrés Vega, César Galindo, José Rangel y
al Dr. Jan Janecek por su interés y
comentarios,

a Ruth Delgado por su colaboración en la
redacción,

a todos los que criticaron, gracias.

Escuela de Ingeniería y Tecnología
1996
ELECTRÓNICA
INGENIERÍA ELECTRONICA

CONTENIDO

INTRODUCCION

CAPITULO 1.

ACERCA DE LOS SISTEMAS OPERATIVOS

1.1	Sistemas Operativos (SO).	1
1.2	Tipos de servicios.	2
1.2.1	Llamados al sistema: comandos internos.	2
1.2.2	Programas de sistema: comandos externos.	3
1.3	Ejemplos y características de SO's.	5
1.3.1	DOS, Disk Operating System.	5
1.3.2	XINU.	8
	Concurrencia.	9
	Estados de procesos en XINU. Proceso nulo.	10
	Manejo de reloj en tiempo real.	11
	Cambio de contexto. Semáforos.	
	Características generales de XINU original.	18
	Características generales de XINU-PC.	19
	Características generales de MINIX.	19

CAPITULO 2.

XINIX

2.1	Características generales de XINIX.	21
2.2	Intérprete de comandos	22
	Manejo de archivos.	23
	Ejecución de programas. Manejo de procesos.	24
	Salida del sistema.	26
2.3	Servicios internos. Ejemplos de utilización	
2.3.1	Servicios para el manejo y coordinación de procesos.	
	Creación de procesos.	27
	Sincronización.	28
	Exclusión mutua.	30
	Comunicación. Mensajes cortos.	32
	Mensajes largos.	34
	Procesos dormilones.	35
2.3.2	Servicios para el manejo de memoria	
	Manejo simple	36
	Manejo controlado, despensas de memoria	37
2.3.3	Servicios para manejar dispositivos	40
2.3.4	Otros servicios.	42

CAPITULO 3.

UTILIZANDO A XINIX

3.1 Medio ambiente de programación XINIX.	43
Inicio del sistema.	45
Preparando una aplicación.	
Ejecutando un programa.	46
Saliendo de XINIX.	48
XINIX versión disco duro.	
3.2 Comandos y servicios XINIX según su aplicación. Qué se ofrece.	
Comandos del intérprete.	53
Servicios internos.	56

CAPITULO 4.

XINIX - XINU

4.1 Organización lógica y física de XINIX. Iniciación.	
Organización lógica.	65
Organización física.	66
Iniciación de XINIX.	68
4.2 Cambios y adiciones a XINU	
4.2.1 Cambios y adiciones generales.	
Apuntadores.	69
Acceso a dispositivos.	72
4.2.2 Cambios y adiciones particulares a cada capa	
Manejador de memoria.	75
Manejador de procesos.	77
Coordinación y comunicación entre procesos	79
Manejadores de dispositivos	
Manejador de reloj.	80
Manejador de terminal.	81
Manejador de disco.	82
Sistema de archivos.	83
Intérprete de comandos.	83
Otros	
Lectura y escritura formateada.	84
Interfaz XINIX.	84

APÉNDICE A

ACERCA DE TERMINALES	87
----------------------	----

APÉNDICE B

XINIX PARA MODIFICACION. INSTALACION	90
--------------------------------------	----

APÉNDICE C

COMANDOS Y SERVICIOS XINIX EN ORDEN ALFABÉTICO. COMO USARLOS.	92
--	----

REFERENCIAS	144
-------------	-----

INTRODUCCION

En un principio los Sistemas Operativos se programaban en lenguaje ensamblador, y su enseñanza, por lo regular, era más teoría que práctica. Cuando recién apareció UNIX (quizás el primer SO escrito en lenguaje de alto nivel, y actualmente el más utilizado en diferentes máquinas), su código fuente estaba disponible a las universidades. Los cursos de SO's balancearon así la teoría y la práctica.

UNIX se difundió también comercialmente, mucho más de lo que se esperaba, y esta fue la razón para no contar más con su código fuente en la educación. Para cubrir esta carencia, en la universidad de Purdue (Indiana USA), se diseñó el SO XINU; y poco después, en la universidad de Vrije (Holanda), el SO MINIX. Ambos incorporan los conceptos de UNIX, y están programados en lenguaje C con pequeñas partes en lenguaje ensamblador. XINU corre en la computadora LSI 11 de DEC, la versión micro de la familia PDP-11, mientras que MINIX corre en la IBM PC-XT y compatibles. Para éstas, en la universidad de Wisconsin (USA) se desarrolló también una versión de XINU, XINU-PC.

La tesis que se presenta es otra portación de XINU a la IBM PC-XT y compatibles, cuyo nombre es XINIX. El cambio de nombre a esta portación obedece a los cambios y adiciones, que se consideran sustanciales (ver la última parte del cap. 1 e inicio del 2), y a que, de MINIX, se tomó parte de los manejadores de dispositivos de terminal y de diskette. Así, el nombre XINIX se forma de las letras mayúsculas de XINU y minIX.

Cabe mencionar que el desarrollo de XINIX inició antes de la aparición de MINIX y de XINU-PC: no se conocía de éstos; y que el motivo inicial de la portación fue enriquecer los cursos de SO's de la Sección de Computación del CINVESTAV del IPN; razones suficientes para decidir transportar XINU a la IBM PC; máquina más popular, de fácil manejo y adquisición y, de la cual, la mencionada sección tiene un número considerable de ellas para que los estudiantes realicen sus tareas y proyectos.

A la fecha, XINIX se ha utilizado para enseñar Sistemas Operativos, Programación de Sistemas en Tiempo Real y Computación Distribuida en la mencionada sección; en la Universidad de Guadalajara y en el Tecnológico de Monterrey, campus Querétaro, se ha utilizado para impartir cursos de Sistemas Operativos; y se espera que muy pronto se utilice en otras escuelas.

Algunos de los trabajos adicionales que se están haciendo y planeando con XINIX son: un sistema de archivos y un intérprete de comandos semejantes a los de UNIX (los originales de XINU son muy sencillos); la adición hecha en Querétaro, "booteo desde floppy", para trabajar como un SO autónomo también se ha de integrar; ejecución de programas DOS; implementar manejadores de disco duro e impresora; y por

ahora, transportarlo a máquinas con el microprocesador 80286 y 80386.

El documento presente está organizado como sigue:

En el capítulo 1 se describen las funciones básicas de un sistema operativo en general; se ilustra como trabajan éstas en DOS, un SO que puede designarse "monotarea" o "monousuario": sólo un proceso se encuentra en ejecución; entonces se ilustra como estas mismas funciones básicas trabajan en XINU (XINIX), SO's que pueden designarse "multitarea"; finalmente, se listan las características generales de XINU, XINU-PC y MINIX.

En el capítulo 2 se dan las características generales de XINIX, se presenta el uso de los comandos del intérprete en una típica sesión de usuario, y se incluyen ejemplos de programas que utilizan los servicios internos de XINIX. Los programas ilustran el tipo de aplicaciones que se pueden desarrollar en XINIX.

El capítulo 3 es el manual del usuario XINIX, describe como preparar y ejecutar una aplicación, como obtener una versión XINIX disco duro para desarrollar aplicaciones (actualmente se encuentra disponible en dos diskettes), y finalmente, el capítulo 3 incluye una lista de los comandos y servicios, por tipo de aplicación, con los que cuenta el usuario.

En el capítulo 4 se explica la organización de los módulos que constituyen a XINIX, que hacer para realizar modificaciones, y se mencionan sólo los cambios y adiciones más significativos respecto a XINU. En este documento no se ve en detalle la implementación de XINIX, pues hubiese requerido demasiado espacio; sin embargo, para beneficio de los estudiantes, se está preparando un reporte técnico que describe la implementación "detallada" de XINIX. De hecho, las notas que han de ser este reporte técnico ya se han utilizado en dar los cursos mencionados, con "muy buena aceptación".

Por último, esta tesis incluye 2 pares de diskettes: el primero es el medio ambiente de trabajo para desarrollar aplicaciones XINIX (ver el cap. 3 para obtener una versión disco duro); el segundo par de diskettes es el código fuente de XINIX, el apéndice B explica que hacer para instalarlo en disco duro y modificarle.

CAPITULO 1

ACERCA DE LOS SISTEMAS OPERATIVOS

Un sistema de computadora tiene dos componentes básicos: hardware y software, los elementos físicos y los programas respectivamente. Los programas determinan el comportamiento del hardware; y siempre tienen un objetivo definido: calcular una diferencial, imprimir una nómina, actualizar una contabilidad etcétera. Un sistema operativo es un programa.

1.1 SISTEMAS OPERATIVOS

El objetivo de un sistema operativo (SO) es manejar y administrar el uso de los componentes físicos y virtuales (como el sistema de archivos). Entre sus funciones se encuentran las siguientes: ejecutar a otros programas y controlarlos, dar y recuperar la memoria que éstos necesitan, manejar los dispositivos externos, como discos e impresoras, y manejar situaciones erróneas. Las funciones de un SO se realizan cuando se le solicita un servicio, tal como la ejecución de un programa.

Un sistema operativo se compone de módulos; uno de éstos, el intérprete de comandos, tiene la función de actuar entre el usuario y el resto del SO, facilita el manejo del sistema. El usuario usa un lenguaje significativo para solicitar las operaciones que desea realizar. El intérprete recibe la solicitud, verifica su validez sintáctica y semántica, y si es correcta, la traduce en un llamado a la rutinas o programas del SO capaces de realizarla.

Estas rutinas son los módulos restantes del SO, y constituyen los servicios disponibles al usuario: manejo de memoria, de archivos, lectura o escritura de datos, control de ejecución etcétera. Estos módulos manejan con detalle a los componentes físicos (el hardware), además de que construyen estructuras lógicas, como los archivos, para facilitar al usuario la realización de sus tareas.

El sistema operativo transforma el hardware, por medio de los servicios que ofrece, en una máquina virtual con habilidades bien definidas.

1.2 TIPOS DE SERVICIOS

La utilidad de un sistema operativo depende de la cantidad y calidad de los servicios que ofrece a los programadores. Hay dos maneras básicas de proveerlos: como "llamados al sistema" y como "programas de sistema", los que también se conocen como "comandos internos" y "comandos externos" respectivamente. Esta definición de los servicios considera la ubicación del conjunto de instrucciones que los realizan al momento de solicitarlos.

Otro tipo de servicios, que de aquí en adelante se les llamará servicios básicos, se encarga de manejar eventos fortuitos; uno no sabe en que momento han de ocurrir. Estos eventos son: errores durante la ejecución de un proceso, las interrupciones del reloj del sistema, y las interrupciones por recibir o transmitir información entre los dispositivos externos conectados al procesador: terminales, diskettes, unidades de cinta, etcétera. Los servicios básicos conservan la integridad del trabajo que se realiza, y no se pueden ejecutar explícitamente por parte del usuario. Por otra parte, cabe mencionar que la mayoría de los procesadores actuales manejan dos tipos de interrupciones: las generadas casualmente por los dispositivos externos, y las generadas explícitamente por los procesos en ejecución; ambas, provocan la "ejecución indirecta" de un procedimiento del SO que las atiende. En términos del microprocesador 8086, se habla de "interrupciones externas" e "interrupciones internas" respectivamente.

1.2.1 Llamados al sistema: comandos internos

Los comandos internos son procedimientos o funciones que, junto con los servicios básicos constituyen el sistema operativo; siempre se encuentran en memoria al momento de solicitarlos. Los servicios que por lo regular se proveen como "comandos internos" (algunos pueden proveerse como externos), son los siguientes:

1) Servicios para el control de procesos:

- Carga y ejecución de programas. Los programas son archivos que regularmente se encuentran en discos o cintas; la carga y ejecución consiste en traerlos a memoria y transferirles el control. Ya en memoria y listo para ejecución, el programa se convierte en proceso: una entidad que compite por la utilización del procesador.
- Terminación o cancelación de un proceso. Al concluir su tarea, un proceso solicita al SO que lo termine. Y aun sin concluirlo, tal vez por un error (una división entre cero), el SO aborta a un proceso. En ambos casos, el SO se encarga de recuperar los recursos que el proceso tenía asignados: memoria, archivos, ...
- Obtención y modificación de los atributos de un proceso: prioridad, memoria disponible, número de archivos que puede manejar, ...
- Suspensión temporal de un proceso.

- Suspensión de un proceso hasta que ocurra un evento particular: recibir datos de la terminal o un mensaje de otro proceso.
- Manifiestar un evento en el sistema, tal vez por el que está esperando otro proceso.

2) Servicios para el manejo de archivos:

- Crear o borrar un archivo.
- Abrirlo o cerrarlo.
- Leer/escribir datos, o posicionarse en un lugar específico del archivo.
- Obtener o modificar sus atributos: derechos de acceso, oculto o no,....

3) Servicios para el manejo de dispositivos:

- Obtener o liberar un dispositivo.
- Leer/escribir datos, o posicionarse en un lugar específico. En el caso de una cinta, uno podría requerir posicionarse al inicio del siguiente archivo.
- Obtener o modificar sus atributos.

4) Servicios para mantener la información del sistema,

- Obtener y/o actualizar la hora y fecha del sistema.
- Obtener y/o modificar los atributos de procesos, archivos o dispositivos.
- Obtener y/o modificar otros datos del sistema como: tiempo asignado a cada proceso; número máximo de terminales, usuarios o procesos.

1.2.2 Programas de sistema: comandos externos

Son programas ejecutables que permanecen en disco o cinta hasta el momento de ser requeridos, momento en el que son cargados y ejecutados; el sistema operativo no aprecia diferencia entre éstos y los programas ejecutables de los usuarios. Los programas de sistema se orientan a crear un medio ambiente para el desarrollo de programas, y se pueden agrupar en una de las siguientes categorías.

1) Modificación de archivos y mantenimiento de la información

Los editores de texto permiten crear o modificar el contenido de archivos almacenados en disco o cinta. Copiadores de dispositivos completos (no archivos), permiten respaldar la información. Formateadores de disco hacen a éstos utilizables por el sistema operativo. Los programas "format" y "diskcopy", en el SO DOS, ejemplifican comandos externo de éste tipo.

2) Soporte a lenguajes de programación.

Compiladores, ensambladores e intérpretes para lenguajes de programación Fortran, Cobol, Pascal, C, Basic, ...; son provistos con el sistema operativo. Aunque últimamente se venden también por separado.

3) Carga y ejecución de programas.

Una vez que un programa se ensambla o compila (se traduce a lenguaje máquina a partir de instrucciones en lenguaje ensamblador o de alto nivel, respectivamente), debe unirse con los módulos o librerías donde están los procedimientos a los que hace referencia, cargarse a memoria, relocalizarle y ejecutarse; para esto se utilizan encadenadores, cargadores y relocalizadores. Es frecuente que los dos últimos constituyan un sólo módulo que hace ambas funciones: carga y relocalización

4) Programas de aplicación.

Son formateadores de texto, sistemas de bases de datos, paquetes de análisis estadístico, contabilidad, inventarios, etcétera.

Obsérvese que aun cuando los "programas de sistema" no son parte del SO, éstos constituyen las herramientas con las que el usuario desarrolla sus aplicaciones. Sin embargo, un programa de sistema depende del sistema operativo para el que se desarrolla: no puede ejecutarse en otro SO. Para aclarar esta situación supongase la compilación del programa prog en lenguaje C, el comando externo luciría de la siguiente manera:

```
§ CC PROG1
```

... el sistema operativo ejecuta al compilador CC a través de un comando interno, el cual recibe el nombre del archivo a convertir en proceso, CC en este caso. El compilador, ya en ejecución, recibe a su vez el nombre del archivo a compilar, PROG1 en este caso. CC abre a PROG1 y lee sus datos (las instrucciones en lenguaje C), ambas operaciones a través de comandos internos al SO; entonces traduce las instrucciones de alto nivel a instrucciones de máquina, las cuales escribe en otro archivo, el programa objeto, después de abrirlo. Esta apertura y escritura del archivo objeto también ocurren a través de comandos internos del SO.

1.3 EJEMPLOS Y CARACTERISTICAS DE SISTEMAS OPERATIVOS

La manera de organizar e implementar los sistemas operativos ha cambiado por el deseo de optimizar el uso de los recursos. En un principio, una aplicación de usuario recibía el control total del sistema. Si ésta efectuaba operaciones de lectura/escritura, el procesador esperaba por la terminación de éstas para continuar con la ejecución de la aplicación. Con el fin de aprovechar el tiempo de espera, varias aplicaciones se cargaban en memoria simultáneamente; si una leía o escribía datos, se bloqueaba hasta que éstos estuviesen disponibles en el sistema, mientras tanto, otra aplicación recibía el procesador; .

Un sistema operativo se designa monitor o monousuario cuando se construye con el primer enfoque, y como SO multiprogramado o multitareas (multi-tasking en inglés), cuando se usa el segundo. Este último evolucionó al punto en que varios usuarios utilizan en forma simultánea (aparentemente) a una computadora, y se le llamó SO de tiempo compartido (time-sharing). Lo anterior se logra asignando, periódicamente, un poco de tiempo del procesador a las aplicaciones de los usuarios en diferentes terminales. Los usuarios perciben a la computadora como totalmente asignada a ellos debido a la velocidad de ésta.

Para ilustrar estos conceptos, a continuación se expone la organización general de los sistemas operativos DOS y XINU-PC. Ambos corren en las computadoras personales (PC) de IBM y compatibles, que utilizan el procesador 8086 de Intel. XINU-PC es la transportación del sistema operativo XINU, desarrollado para la versión micro de las computadoras PDP-11 de Digital Equipment Corporation (DEC). También se menciona un poco de MINIX, para PC's, con propósitos de comparación.

1.3.1 DOS, Disk Operating System.

Actualmente es el SO más utilizado en la PC, debido principalmente a la gran cantidad de programas de utilería que se han desarrollado con él: lenguajes de programación, editores de texto y formateadores, paquetes de bases de datos, hojas de cálculo, sistemas administrativos, etcétera.

DOS provee servicios internos y externos. Para ejecutar un servicio interno, su identificación (un valor entero) y los parámetros que utiliza se pasan en los registros del 8086; el control se pasa DOS generando la interrupción interna 21H. Además, DOS es monousuario, y su intérprete de comandos se ofrece como un servicio externo (se ejecuta como un programa) y, a diferencia de MINIX y XINU, el usuario promedio no tiene acceso al código fuente. Veamos el comportamiento de un SO monousuario.

Una vez en memoria, DOS ejecuta al intérprete de comandos (el archivo COMMAND.COM). El intérprete se encarga de desplegar los conocidos mensajes (prompts): A>, B>, C> o D>, indicando el manejador de disco en el que uno se encuentra y que está listo para recibir un comando.

Para ejecutar al programa-1 de la figura 1.1 se teclea su nombre:

```
A>>programa-1
```

...el intérprete solicita a DOS el servicio de ejecución. DOS busca el archivo "programa-1.exe" en el disco A, y si lo encuentra lo carga en memoria, lo relocaliza y le pasa el control, en caso contrario devuelve un mensaje de error y regresa el control al intérprete.

El prompt A> no aparece durante la ejecución de programa-1; cuando éste termina DOS libera la memoria que ocupó, y pasa de nuevo el control al intérprete, el cual vuelve a desplegar el prompt. ¿Cómo entonces se ejecutan los programas programa-2,... y programa-n? Bueno, DOS conoce la dirección inicial y la cantidad en bytes de la memoria disponible para ejecutar programas; al ejecutar a uno actualiza estos dos valores: la dirección aumenta según el tamaño del programa, y la cantidad disminuye en lo mismo; la ejecución de un 2do o n-avo programa sigue el mismo esquema: se carga a partir de la dirección inicial actual y se actualiza la dirección y la cantidad de la memoria disponible, sólo que en este caso, el servicio de ejecución lo solicita, para el programa-2, programa-1, y para el programa-n, programa-n - 1.

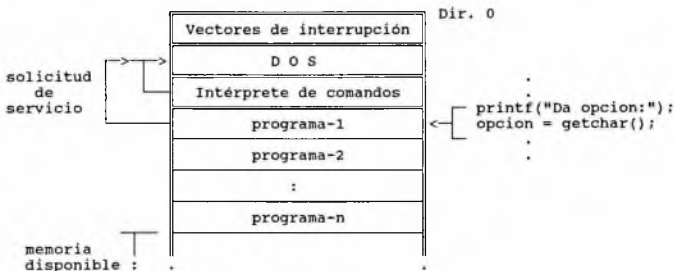


Figura 1.1. Entorno de programación DOS

Una vez en ejecución un programa recibe todo el control, como se verá más adelante, el mismo SO pasa a formar parte de él como un procedimiento más. Un programa pierde el control al terminar, al ejecutar a otro programa (pérdida voluntaria), o en caso de error durante su ejecución. Cuando un programa termina, el control vuelve al programa que le ejecutó, a programa-n - 1 cuando termina programa-n, ..., y al intérprete cuando termina programa-1. Se podría pensar de las ejecuciones de programas como llamadas a procedimientos, que empiezan y terminan en el intérprete.

Analicemos ahora lo que sucede al ejecutar el supuesto código de programa-1:

```

:
printf("Da opcion:");
opcion = getchar();
:

```

Ambas operaciones, leer o escribir datos, devienen en solicitudes de servicio a DOS; getchar() en particular se atendería con un procedimiento de DOS semejante a

```
Getc() {
    char ch;                               /* Instrucción */

    while (num_cars == 0)                  /* 1 */
    {
        ch = datos[devuelve++];           /* 2 */
        if (devuelve == TAMAÑO_BUFFER)   /* 3 */
            devuelve = 0;                 /* 4 */
        num_cars--;                         /* 5 */
        return(ch);
    }
}
```

Las variables num_cars, devuelve, inserta y datos[], serían globales y definidas con los siguientes tipos:

```
#define TAMAÑO_BUFFER 30;

char datos[TAMAÑO_BUFFER];
int num_cars = 0; /* número de caracteres en datos */
int devuelve = 0; /* sig. carácter en datos a devolver */
int inserta = 0; /* posn. en datos para meter nvo carácter */
```

Se declaran globales con el fin de que las utilice Getc() y además el siguiente procedimiento:

```
void interrupt Terminal() ( /* Instrucción */
    char ch;

    ch = Teclado();          /* 1 */
    if (num_cars == TAMAÑO_BUFFER) /* 2 */
        Beep(); /* trata con hardware */ /* 3 */
    else {
        datos[inserta++] = ch; /* 4 */
        num_cars++;           /* 5 */
        if (inserta == TAMAÑO_BUFFER) /* 6 */
            inserta = 0;      /* 7 */
    }
}
```

Getchar(), Terminal(), Teclado() y Beep() conforman un manejador de terminal para DOS. Terminal() es además parte de un servicio básico (los que atienden eventos fortuitos), se ejecuta cada vez que se oprime una tecla. Terminal() obtiene el carácter asociado a la tecla oprimida y lo guarda en ch, instrucción 1. Teclado() se encarga de los detalles del hardware: si se oprimió una letra, revisa si antes se oprimió una tecla "shift", en cuyo caso devuelve una mayúscula y no una minúscula, etcétera.

Al oprimir una tecla por primera vez, las variables num_cars, inserta y devuelve valen 0, por lo que la instrucción 2 en Terminal() continúa en la 4: el primer carácter se guarda en datos[0], el segundo se ha de guardar en datos[1], pues el operando "++" incrementa en 1 a inserta después de depositar el carácter. Num_cars también se

incrementa: su valor representa el número actual de caracteres en datos[], instrucción 5. Eventualmente, tecleando, num_cars e inserta llegarán al valor de TAMANO_BUFFER; en tal caso la instrucción 2 ignora el carácter que se haya tecleado, el sonido provocado por la función beep() avisa al usuario que la capacidad del arreglo datos[] está en el límite. Cuando inserta iguala a TAMANO_BUFFER, instrucción 6, justo rebasando la capacidad de datos[], se pone a 0 para apuntar de nuevo al principio: dar la vuelta; por esta razón, datos[] se esta manejando como una "lista circular".

Para satisfacer el servicio que getch() constituye en la instrucción "opcion = getch()" en programa-1, DOS ejecuta a Getc(). Esta revisa, en la instrucción 1, si se han tecleado caracteres; si no hay, la instrucción while permanece mientras num_cars sea igual a cero: Getc(), DOS, programa-1, el intérprete de comandos: el 8086, están esperando que cambie el valor de num_cars; lo que ocurrirá cuando, al oprimir una tecla, se ejecute Terminal().

Si hay caracteres en datos[], la instrucción 1 en Getc() pasa inmediatamente a la 2, el carácter que se devolverá al usuario se guarda en la variable ch; el siguiente a devolver se direcciona gracias al código "devuelve++"; y al igual que en Terminal(), si se ha alcanzado el límite de datos[], devuelve se pone a 0 para darle la vuelta; num_cars se decrementa con el operador "--", instrucción 5, ahora hay un carácter menos en el datos[]; finalmente se devuelve el carácter al usuario, instrucción "return(ch);".

El esquema anterior se presenta en DOS cada que se leen o escriben datos en un dispositivo: la máquina completa queda bajo el control del programa que hace la petición. Esta situación en una máquina grande es incosteable e ineficiente.

Por último, y en favor de DOS, recuérdese que la utilidad de un SO se mide en la calidad y cantidad de servicios que ofrece, sean internos o externos, y DOS es el SO para PC's en el que se ha desarrollado más software para beneficio de los usuarios.

1.3.2 XINU

XINU y MINIX, a diferencia de DOS, son sistemas operativos concebidos con propósitos educacionales. Para cada uno, los autores escribieron un libro, [COMER84] y [TANEN87] respectivamente, en el que explican la organización y ofrecen el código fuente en lenguaje C. Ambos autores, Douglas E. Comer de XINU, y Andrew S. Tanenbaum de MINIX, han contribuido enormemente a la enseñanza práctica, diseño y construcción, de sistemas operativos.

XINU y MINIX son SO's multi-tareas, varios "procesos" pueden ejecutarse "concurrentemente": comparten el procesador y tal vez datos e instrucciones durante su ejecución. Este es el esquema de la máquinas grandes, optimizar el uso de los recursos, lo cual no es fácil diseñar, ni programar (ni explicar). Para empezar se ilustrará el concepto de...

Concurrencia

Sean dos programas independientes a correr en DOS. Uno tendría que dar los siguientes comandos:

```
A>prog1 /* esperamos a que termine y aparezca de nuevo el prompt */
A>prog2
```

Haciendo una gráfica de la asignación de tiempo del procesador observaríamos lo siguiente:

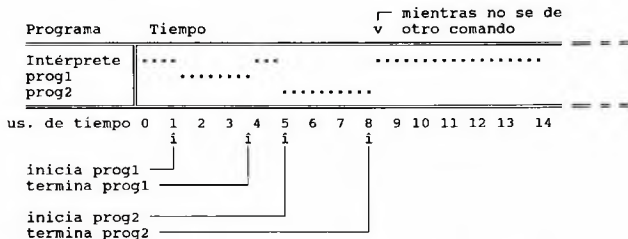


Figura 1.2. Asignación del procesador en DOS.

Puede pensarse del intérprete como un programa principal que llama a los procedimientos prog1 y prog2. Cuando ejecuta a prog1, el intérprete espera a que "termine totalmente" para ejecutar a "prog2".

La misma gráfica en XINU luciría de la siguiente manera:

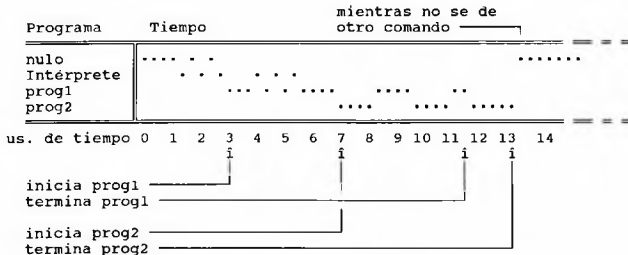


Figura 1.3. Asignación del procesador en XINU.

Estados de procesos en XINU

Antes de explicar la figura 1.3, es necesario mencionar que en XINU existen varios estados en los que puede estar un proceso, éstos son:

- READY.- el proceso esta listo para recibir el procesador.
- SUSPEND.- el proceso se encuentra en el limbo, ni compite por el procesador ni espera por algún evento, sólo esta latente, otro proceso debe sacarlo de su letargo.
- WAITING.- el proceso espera por un evento; mientras tanto no compite por el procesador; el evento lo provoca otro proceso, y cuando ocurre pasa al estado READY.
- RECEIVING.- el proceso espera por un evento particular: un mensaje por parte de otro proceso; mientras espera no compite por el procesador; cuando recibe el mensaje pasa a READY.
- SLEEPING.- el proceso espera a que transcurra un intervalo de tiempo para volver a pasar al estado READY, y
- CURRENT.- es el estado del proceso que se ejecuta actualmente.

XINU comparte el procesador entre los procesos listos (READY) que compiten por él. La manera de dar tiempo de procesador se basa en la prioridad de los procesos, a los que se les asigna al momento de crearlos. La prioridad varia de 0 a 32767, siendo 0 la menor. Cuando varios procesos tienen la prioridad actual más alta, cada uno recibe un intervalo de tiempo en el orden en que llegaron a competir por el procesador; al terminar su intervalo se colocan al final de los procesos con su misma prioridad, de esta manera, antes de recibir el segundo intervalo sus competidores han recibido el primero. Este esquema de asignación del procesador se llama "Round Robin", y tiene la característica de ignorar completamente a los procesos de menor prioridad: si existe un sólo proceso con la prioridad más alta, sólo él se ejecutaria.

A diferencia del intérprete de DOS el de XINU no acapara al procesador, en su lugar, espera (WAITING) a que se teclee el comando que debe ejecutar. (En realidad, el acaparamiento o la espera se debe, a la manera como está construido el servicio de lectura de caracteres.) Debido a que la máquina es mucho más rápida que el usuario al teclear, el intérprete pasa varias veces al estado WAITING antes de recibir el comando completo; cuando esto sucede, lo ejecuta y vuelve a esperar por otro comando. Ya que en el estado WAITING no se compite por el procesador, y en el intervalo 0-1 de la figura 1.3 no se ha ejecutado prog1 ni a prog2, ¿quién utiliza al procesador ?

Proceso nulo

La existencia de un proceso nulo es característica general de los SO's multi-tareas, y se encarga de consumir el tiempo del procesador cuando no existen "procesos READY" que lo utilicen. Su prioridad es 0, la mínima; el intérprete y los procesos del usuario deben crearse con una prioridad mayor, para que al estar READY, el proceso nulo quede totalmente relegado. El código del proceso nulo es un loop infinito, en

XINIX, en lenguaje C, se codificó de la siguiente manera:

```
while (TRUE) /* TRUE es igual a 1 */
```

Volviendo a la figura 1.3, en los intervalos 1 a 3 y 4 a 6, el intérprete recibe el comando para ejecutar a prog1 y prog2 respectivamente. Una vez READY, prog1 y prog2 "comparten el procesador a intervalos regulares" mientras no terminen, ver la distribución del intervalo 6 a 13 en la misma figura. Tal es la concurrencia. Bueno, ¿y qué se necesita para que exista la concurrencia ?

Manejo de reloj en tiempo real

En todas las computadoras actuales existe, además del procesador que ejecuta las instrucciones de los programas, un dispositivo (chip) reloj que interrumpe periódicamente al primero. Cada interrupción de reloj es atendida por un procedimiento provisto para tal fin por el SO. En el caso DOS, este procedimiento incrementa el contenido de dos variables enteras; de éstas, al aplicar una conversión, se obtiene la hora del sistema en forma de cadena de caracteres.

Además de esto, el procedimiento que atiende el reloj en XINU realiza, cada segundo (aunque es posible aumentar o disminuir este tiempo), si procede según prioridades, un "cambio de contexto": asigna el procesador a otro proceso. Actualiza también el tiempo que deben esperar los procesos durmientes, los que están en el estado SLEEPING, y para los que es tiempo de despertar se cambia su estado a READY y provoca "cambio de contexto". En otras palabras, el manejo de reloj en tiempo real, en XINU (ver [COMER84] p. 125), consiste en "limitar la cantidad de tiempo que un proceso puede ejecutarse, así como proveer a los programas de usuario con servicios para realizar retardos de tiempo".

Cambio de contexto

Antes de ejecutarse, una aplicación es un archivo. Una vez en ejecución se convierte en "proceso", ahora no solo importa su constitución, también es importante el punto de ejecución donde se encuentra; él cual está determinado por el contenido de los registros del procesador. Un proceso se compone entonces de instrucciones, datos y del contenido de los registros del procesador en un momento dado. En XINU los datos de cada proceso se colocan en una área "individual" de memoria llamada pila. En la pila es donde se ponen los parámetros para los procedimientos que un proceso llama, la dirección de retorno para volver de ellos, y donde se manejan las variables locales que los procedimientos utilizan. Por ser una pila por cada proceso, el contexto de cada uno se puede resumir al contenido actual de los registros cuando se encuentran en ejecución. Un cambio de contexto, entonces, implica guardar el contenido actual de los registros, que pertenece al proceso actual (CURRENT), y cargarles con los valores pertenecientes al proceso que ha recibir el control, el proceso READY con mayor prioridad que se ha de convertir en CURRENT.

Más sobre el estado de los procesos en XINU. Transiciones

Para manejar los procesos en forma concurrente, XINU utiliza una tabla de procesos (proctab[]) en la que guarda los atributos de cada uno de ellos: su estado, su contexto, donde fue cargado en memoria, su prioridad, la dirección de su espacio-pila, su nombre, etcétera. Además, provee los servicios, comandos internos, para que los procesos transiten de un estado a otro según sus necesidades y de acuerdo a la figura 1.4.

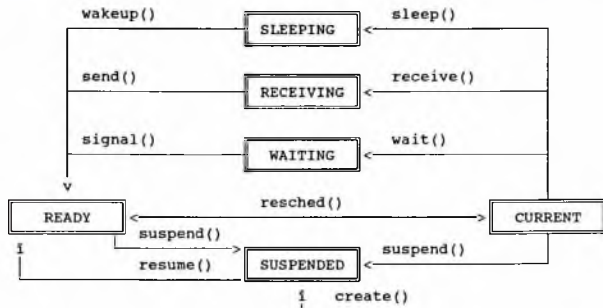


Figura 1.4. Estados de procesos en XINU.

Los estados están indicados dentro de los cuadros, las flechas indican el sentido en que los procesos pasan de un estado a otro, y los nombres seguidos de paréntesis ["()"], son los procedimientos-servicio de XINU que provocan la transición. El procedimiento que atiende las interrupciones de reloj, ejecuta a resched() para provocar cambio de contexto cada segundo, y a wakeup() para despertar a los procesos durmientes que han consumido su tiempo de dormir.

¿ Porqué tantos estados y transiciones ?

Si tan sólo un proceso existe, como en DOS, él está siempre activo, recuérdese la instrucción

```
while (num_cars == 0)
    ;
```

...en el proceso Getc() del manejador de terminal explicado antes. El intérprete ejecutó a un programa, y éste, al solicitar un servicio a través de getchar(), provocó que DOS ejecutara Getc(). Todo

empieza y termina en el intérprete, "sólo hay un proceso". Así que el procesador puede esperar, "activo" en la instrucción "while", hasta que el usuario decida teclear algo, lo que provocará que cambie el valor de num_cars y la continuación de Getc().

Supóngase que XINU lee caracteres con un procedimiento semejante a Getc() de DOS, y además, que hay 3 procesos de usuario, uno de ellos, nombrémosle "cálculo", no lee datos de la terminal y tiene prioridad 5; los otros dos, "lector 1" y "lector 2", leen datos de la terminal y tienen prioridad 6.

¿Qué pasa si no se teclea nada durante horas? Bueno, "lector 1" y "lector 2", por tener la máxima prioridad, serían los únicos en utilizar el procesador, y lo harían precisamente en la instrucción "while (num_cars == 0)", en el procedimientos Getc(), durante su estado CURRENT, pasando a READY al ocurrir cambio de contexto. Mientras tanto, el proceso "cálculo", aún cuando su estado es READY y no lee caracteres, nunca se ejecuta.

Para evitar esta situación, el manejador de terminal de XINU está programado de otra manera. Cuando un proceso espera por un evento "fortuito", como lo es la recepción de caracteres (uno no sabe en que momento se oprimirán las teclas), su estado se cambia a WAITING, en el que no compite por el procesador. Cuando el evento ocurre, su estado cambia a READY para continuar su ejecución. En el ejemplo anterior, "cálculo" se ejecutaría mientras no se teclee algo.

Estado WAITING. Implementación del servicio wait() y signal()

XINU utiliza semáforos para que un proceso transite del estado CURRENT a WAITING, y de WAITING a READY. Los servicios que provocan tales transiciones son wait() y signal() respectivamente. Antes de explicar como funcionan los semáforos se mencionarán 2 puntos importantes. Primero, XINU esta construido por capas: los servicios de una sirven para construir las que le siguen, la figura 1.5 muestra la organización lógica de XINU. Segundo, el manejador XINU del dispositivo terminal utiliza los servicios wait() y signal() de la capa de "coordinación de procesos". Cuando un programa de usuario solicita leer caracteres, "implícitamente" ejecuta a wait() para bloquearse (pasar al estado WAITING) si no hay. Por otra parte, el usuario puede utilizar explícitamente a wait(), u otro procedimiento de XINU para el desarrollo de sus aplicaciones, en el capítulo 2 se dan ejemplos.



Sistema de archivos
Manejadores de dispositivos
Comunicación entre procesos
Coordinación de procesos
Manejador de procesos
Manejador de memoria
HARDWARE de la PC

Figura 1.5. Organización lógica de XINU. Capas.

Wait() y Signal(). Semáforos. Coordinación.

Los semáforos se pueden utilizar para "coordinar" la ejecución entre procesos. Muchos esquemas de ejecución entre varios procesos pueden definirse "productor-consumidor". Un proceso produce los "recursos" que otro consume: un manejador de terminal (MT) produce los caracteres que un programa de usuario desea leer; si no hay caracteres éste no puede continuar su ejecución; la que se "coordina" de acuerdo a la producción de caracteres del MT. El esquema productor-consumidor tiene variantes: varios procesos pueden solicitar leer los caracteres que el MT produce; y si no se tecléa nada, varios procesos quedarán "esperando".

A continuación se muestran los atributos de los semáforos XINU:

```
struct sentry {          /* entrada en la tabla de semáforos */
    char sstate;         /* semáforo trabajando o no */
    short semcnt;        /* cuenta del semáforo: recursos disponibles */
    short sqhead;        /* inicio de la lista del semáforo */
    short sqtail;        /* fin de la lista del semáforo. */
};
```

Un semáforo puede o no estar trabajando; tiene una cuenta asociada, que en el caso del MT significaría el número de caracteres teclados y disponibles; y el inicio y final de la lista de procesos bloqueados si existen. Los atributos de todos los semáforos se encuentran en el arreglo semaph[]. Un semáforo se identifica entonces por su número de entrada en semaph[]. A continuación el código (simplificado) del servicio wait() en XINU, [COMER84] p. 85:

```
/*-----
 * wait -- make current process wait on a semaphore
 *-----
 */
SYSCALL wait (sem)
int sem;
{
    struct sentry *sptr;
    struct pentry *pptr;

    if (isbadsem(sem) || (sptr= &semaph[sem])->sstate==SFREE)
        return(SYSERR);
    if (--(sptr->semcnt) < 0) {
        (pptr = &proctab[currpid])->pstate = PRWAIT;
        pptr->psem = sem;
        enqueue(currpid, sptr->sqtail);
        resched();
    }
    return(OK);
}
```

Wait() recibe la identificación del semáforo, si es inválida o el semáforo no está trabajando regresa el valor SYSERR. De otra manera, y si hay recursos, manifiesta que toma uno: --(sptr->semcnt), disminuye el número de éstos y regresa el valor OK; más sino hay recursos, el

proceso que llamó a wait() obtiene la dirección de su entrada en la tabla de procesos: &proctab[currpid], y cambia su estado a WAITING (PRWAIT), ya no competirá más por el procesador; además, en la misma entrada, en el campo psem, guarda la identificación del semáforo que lo hizo cambiar de estado; entonces se forma al final de la lista asociada al semáforo: enqueue(...); y finalmente provoca cambio de contexto al llamar al procedimiento resched().

Comentarios adicionales. En XINU, la variable global currpid contiene la identificación, entrada en la tabla de procesos, del proceso actual en ejecución (CURRENT). Resched() se encarga de actualizarla antes de realizar el cambio de contexto, le asigna la identificación del proceso READY con mayor prioridad.

Continuando con el código de wait(); el campo semcnt nos representa el número actual de recursos (en el manejador de terminal es el número de caracteres teclados y esperando que un programa los lea). Si existe un carácter, entonces semcnt es igual a 1 al ejecutar a wait(), al llegar a la instrucción

```
if (--(sptr->semcnt) < 0) { ...
```

... semcnt es disminuido antes de la comparación, quedando igual a 0; por lo tanto, la condición no se cumple y el proceso que llamó a wait() no pasa al estado WAITING: no se bloquea. Más si no hay caracteres, semcnt vale 0 al llamar a wait(), y valdrá -1 al realizar la comparación en la instrucción if; se cumplirá la condición y el proceso que llamó a wait() se bloqueará.

Con este estado y antes de teclear algo, supóngase que otro proceso lee caracteres de la misma terminal. Eventualmente ejecutará a wait(), y al llegar a la instrucción if donde compara el valor de semcnt, ya disminuido en 1, éste ahora valdrá -2; así que la condición se cumple y este otro proceso también se bloquea.

Signal(), de [COMER84], aclara la transición WAITING->READY.

```
/* -----  
* signal -- signal a semaphore, releasing one waiting process  
* -----  
*/ SYSCALL signal(sem)  
    int sem;  
{  
    struct sentry *sptr;  
  
    if (isbadsem(sem) || (sptr= &semaph[sem])->sstate==SFREE)  
        return(SYSERR);  
    if ((sptr->semcnt++) < 0)  
        ready( getfirst(sptr->sqhead), RESCHYES);  
    return(OK);  
}
```

Continuemos con el ejemplo del manejador de terminal. Ahora se va a suponer que se ha teclado un carácter. La parte de XINU que atiende la interrupción por oprimir una tecla, realiza algo semejante al procedimiento Terminal() definido para DOS: obtiene el carácter y lo

pone, si existe espacio, en `datos[inserta++]`; si no hay espacio, avisa usuario con una función `beep()`. Pero además de esto, hace un `signal()`. `Signal()` recibe la identificación del semáforo que controla al manejador de terminal, la identificación que usó el proceso de usuario al ejecutar `wait()` cuando solicitó leer un carácter. `Signal()` valida lo mismo que `wait()`. Entonces obtiene la cuenta de recursos disponibles, la compara contra cero y la incrementa: `{sptr->semcnt++}`. Nótese que el incremento a la cuenta ocurre después de la comparación. Si la cuenta era negativa, obtiene, de la lista asociada al semáforo, la identificación del primer proceso que espera por un carácter, y lo enlista, según su prioridad, en la lista de procesos en estado `READY`; por ultimo, hace cambio de contexto:

```
ready(getfirst(sptr->sqhead), RESCHYES);
```

Por otra parte, si la cuenta del semáforo no era negativa, entonces no hay procesos bloqueados esperando por un recurso, y sólo se devuelve `OK`. Lo anterior se aclara si se considera que `wait()` y `signal()` mantienen la siguiente condición invariante, [COMER84]:

Una cuenta de semáforo no negativa significa que la lista esta vacia; una cuenta de semáforo con valor `n` negativo significa que la lista contiene `n` procesos esperando.

Para terminar el ejemplo de coordinación con semáforos, se muestra el código simplificado del procedimiento de XINU equivalente a `Getc()` de DOS:

```
/* -----
 * ttygetc - read one character from a tty device
 * -----
 */
ttygetc()
(
    int ch;

    wait(term_sem); /* wait for a character in datos */
    ch = buff[devuelve++];
    if (devuelve == IBUFLEN)
        devuelve = 0;
    return(ch);
)
```

Mientras que `Getc()` espera por caracteres con la instrucción `"while(num_cars == 0); ttygetc()` llama a `wait()`. Si hay caracteres `wait()` regresa inmediatamente: la cuenta del semáforo `term_sem` es mayor o igual a 1; entonces se obtiene el siguiente carácter a devolver, el apuntado por devuelve en el arreglo `buff[]`, él que también se maneja como lista circular. Si no hay caracteres, entonces `wait()` no regresa inmediatamente: el proceso que llamó a `ttygetc()` pasa al estado `WAITING`, y el procesador pasa al proceso con más alta prioridad en la lista `READY`.

Para ilustrar la diferencia entre un `SO` monousuario y otro multitareas, se ha explicado el manejador de terminal de `DOS` y de `XINU`.

En el primero sólo existe un proceso en todo momento, y sus peticiones al SO para leer datos de la terminal, por ejemplo, utilizan al procesador en checar la obtención de éstos (while (num_cars==0)). Mientras que en XINU, el proceso que desea leer caracteres es bloqueado mientras éstos no sean teclados. Existen otras situaciones en las que DOS utiliza al procesador en checar la completitud de una operación con dispositivos externos. A continuación se explica una de ellas.

Para leer o escribir un sector de datos en el diskette, se programan los chips Floppy Disk Controller (FDC) y Direct Memory Access (DMA): se escriben en sus puertos asociados, varias secuencias de bits que constituyen el comando. Al FDC se le indica la pista, el lado, el sector y si éste se va a leer o escribir. Mientras que al DMA se le indica la dirección de memoria, utilizada por el 8086, a partir de donde se ha de leer lo que se escribirá al diskette, o donde se escribirá lo que se lea del diskette. La última secuencia de bits escrita en el FDC, según el comando, inicia la operación de transferencia. A continuación un esquema simplificado de una petición a DOS para leer un sector en un diskette; la versión real, en lenguaje ensamblador, se encuentra en el apéndice A, paginas A-36 a A-44, ver esta última, de [TRIBM83].

```
int status; /* var. global: estado de la operación actual */

lee_sector(manejador, pista, lado, sector, buff)
int manejador; /* manejador A>, B>, C> o D> */
int pista; /* o cilindro a leer */
int lado; /* lado 0 ó 1 */
int sector;
char *buff; /* donde se ponen los datos leídos */
{
    status = 0; /* operación sin terminar */
    set_dma(LECTURA, buff); /* programa al DMA */
    /* programa FDC e inicia operación */
    set_fdc(LECTURA, manejador, pista, lado, sector);
    while (status == 0)
        ; /* espera a que termine la operación */
}
```

El sector a leer podría ser el que contiene directorio del diskette. El comando "dir", dado al intérprete de comandos, pediría a DOS que ejecute el servicio lee_sector(...). Nótese que lee_sector(), una vez emitido el comando, espera la completitud de éste en la instrucción while(status==0); que es equivalente a while(num_cars==0) en Get(): el procesador se utiliza en esta espera. Cuando el FDC termina la transferencia de información, genera una interrupción que es atendida por un procedimiento semejante a Terminal():

```
void interrupt Diskette()
{
    status = 1; /* status es global */
}
```

Al ejecutarse Diskette(), temporalmente suspende la ejecución de lee_sector() en la instrucción while, cambia el valor de status, y

regresa de nuevo a `lee_sector()` en la instrucción `while`; ahora `lee_sector()` encuentra a `status` igual a 1, así que sale de la instrucción `while`, y al proceso que solicitó el servicio, en nuestro ejemplo, el intérprete de comandos.

El manejador de diskette XINIX (una versión PC de XINU), utiliza un procedimiento equivalente a `lee_sector()`, `transfer()`, para emitir el comando al FDC y al DMA, que en lugar de la instrucción `while` ejecuta el servicio `receive()`: espera, sin competir por el procesador, por un mensaje que le ha de enviar la versión correspondiente de `Diskette()`, `diskioint()`, cuando el FDC termine la operación encomendada. Véase la figura 1.4 y el apartado "Estado de procesos en XINU" al inicio de esta sección. Ver los procedimientos equivalentes en el archivo `x2dskdrv.c` en el diskette XINIX "Instala y Fuentes en C". El apéndice B habla acerca de este y otro diskette.

Otra situación semejante se presenta también con el manejador de diskette (MD). En DOS, después 2.5 segundos de la última operación con un MD, su motor es apagado con el fin de evitar desgaste prematuro en los diskettes. Por esta razón, toda operación con un MD chequea antes si su motor está andando. Si es así procede a programar al FDC y al DMA. Pero sino, al chip que maneja los motores se le indica que arranque el motor en cuestión. Empero, la transferencia no puede programarse inmediatamente, hay que esperar, aproximadamente 2.5 décimas de segundo, a que el motor alcance su velocidad de trabajo. DOS realiza esta "espera" de la siguiente manera: a una variable mueve un cierto valor que decreta en cada iteración de una instrucción `while`, cuando la variable es igual a cero, la instrucción `while` termina y ha transcurrido el tiempo para que el motor en cuestión alcance su velocidad de trabajo. El código real que lleva a cabo esta espera, en lenguaje ensamblador, se encuentra en la página A-39 de [TRIBM83]. Nuevamente, el procesador se utiliza en esperar la ocurrencia de un evento. Por su parte, el manejador de diskette de XINIX, al detectar que debe encender un motor, emite los comandos para tal efecto, pero en lugar de esperar en una instrucción `while`, ejecuta el servicio `sleep10(3)`, que lo suspende sin competir por el procesador, durante 3 décimas de segundo, dando oportunidad a otros procesos de que se ejecuten. Véase de nuevo la figura 1.4, el apartado "Estado de procesos en XINU" al inicio de esta sección, y el código del procedimiento `start_motor()`, en el archivo `x2dskdrv.c` en el diskette XINIX "Instala y Fuentes en C".

Este capítulo termina listando las características generales de XINU original, XINU-PC y de MINIX, las de XINIX se listan al inicio del capítulo siguiente, donde además se incluyen ejemplos de utilización

Características generales de XINU original

Corre en las computadoras LSI/11 de Digital Equipment Corporation.

El tamaño máximo de una aplicación, incluyendo XINU, es 64 Kbytes.

El manejador de terminal trabaja con terminales RS232. Las características de estas terminales, y las de memoria mapeada que

se mencionan en el apéndice A.

Sistema de archivos plano: no permite manejar subdirectorios. Capacidad limitada en el número de archivos, 28 por disco.

El desarrollo de una aplicación se realiza en una computadora VAX, y una vez lista, se transfiere a la computadora LSI/11 por un puerto serie. Antes de empezar la ejecución, en la VAX se corre un programa que la hace simular una terminal tipo RS232 para la LSI/11. En la VAX se encuentran el compilador, el encadenador, el relocador y el cargador.

No tiene intérprete de comandos. No es posible correr programas de manera interactiva: desde una imagen ejecutable en disco. Todos los procesos del usuario se encadenan a XINU.

Características generales de XINU-PC

Corre en las microcomputadoras IBM-PC y compatibles.

El tamaño máximo de una aplicación, incluyendo XINU, es 64 Kbytes.

El manejador de terminal trabaja con terminales de memoria mapeada (la de la IBM-PC) con capacidad para manejar 4 ventanas en forma independiente.

El sistema de archivos tiene la misma estructura que el de XINU original. Provee además una interfaz para utilizar los archivos de DOS, pues XINU-PC se ejecuta como un programa de usuario DOS.

Las aplicaciones se desarrollan bajo DOS, utilizando el compilador MSC y el encadenador "link" de Microsoft o TurboC de Borland.

No tiene intérprete de comandos. No es posible correr programas de manera interactiva: desde una imagen ejecutable en disco. Todos los procesos del usuario se encadenan a XINU.

Características generales de MINIX

Corre en las microcomputadoras IBM-PC y compatibles.

Maneja dos modelos de memoria; el primero permite, sin incluir a MINIX, un tamaño máximo de 64 Kbytes por aplicación; el segundo modelo permite, sin incluir a MINIX, 64K para instrucciones y 64K para datos y pila.

El manejador de terminal trabaja con terminales de memoria mapeada (la de la IBM-PC).

El sistema de archivos tiene la misma estructura que el de UNIX: maneja subdirectorios y protección por archivo.

Las aplicaciones se desarrollan bajo MINIX, se provee compilador encadenador, y servicio de carga y relocalización. Es posible utilizar otro compilador, en cuyo caso se provee una utileria para cambiar la cabecera del archivo ejecutable resultante, sin embargo, el autor advierte que se pueden presentar errores.

Tiene intérprete de comandos. Es posible correr programas de manera interactiva: desde una imagen ejecutable en disco. Los programas del usuario se encadenan sólo a la libreria de MINIX.

CAPITULO 2

XINIX

XINIX es la versión PC, modificada y aumentada, del sistema operativo XINU original. En este capítulo se ven sus características generales y ejemplos de como utilizar los servicios que ofrece: una sesión típica con el intérprete de comandos, así como programas ejemplo que muestran el tipo de aplicaciones que se pueden desarrollar.

2.1 CARACTERISTICAS GENERALES DE XINIX

Corre en las microcomputadoras IBM-PC y compatibles.

El tamaño máximo de una aplicación es la capacidad de la memoria menos el tamaño de XINIX y DOS. Con 640 Kb de memoria, una aplicación XINIX puede ser, más menos, de 460 Kb.

El manejador de terminal trabaja con 1 terminal de memoria mapeada, la de la PC, y con 2 terminales RS232 conectadas a un puerto serie, aunque es posible reconfigurar a XINIX para que maneje más terminales RS232.

El sistema de archivos es el de XINU original, más provee utilería para convertir archivos DOS a archivos XINIX.

Las aplicaciones se desarrollan bajo DOS, utilizando el compilador TurboC de Borland.

Tiene intérprete de comandos. Hay comandos que informan sobre el estado de los recursos del sistema, comandos que ejecutan programas (archivos ejecutables) y controlan procesos, comandos que manejan archivos y comandos de servicio.

Los procesos del usuario, antes programas en disco, utilizan los servicios XINIX por medio de una interfaz que se encadena al momento de crear el programa ejecutable. Esta interfaz, o librería de XINIX, es un conjunto de procedimientos, cada uno de los cuales constituye un llamado a un servicio a través de interrupciones. Así, el desarrollo es en DOS y la ejecución en XINIX con el intérprete de comandos.

Comentarios adicionales

Como quedó en la versión 1.0, XINIX es huésped de DOS: es un programa ejecutable que corre en DOS. Sin embargo, una vez en ejecución, XINIX toma el control de la PC: carga los vectores de interrupción con la dirección de sus manejadores de reloj, de terminales y de diskette. Cuando el usuario XINIX desea volver a DOS, sólo tiene que oprimir las teclas Ctrl, Alt y Del simultáneamente; tal y como lo hace cuando reinicia el sistema. Antes de regresar el control a DOS, XINIX reestablece los vectores de interrupción, dejando el sistema tal y como se encontraba antes de ejecutar a XINIX. Esta configuración es conveniente considerando el desarrollo actual de XINIX: actualmente no existe el software básico desarrollado para él: compiladores, ensambladores, editores de texto, etcétera. Por otra parte, la manera actual de utilizarlo es realmente cómoda para el usuario, tan sólo requiere 2 diskettes: el primero con el SO DOS y el compilador TC, y el segundo con las librerías de TC, la interfaz de XINIX y XINIX. El usuario desarrolla sus aplicaciones en el segundo diskette. El primero y segundo diskette se insertan en los manejadores de disco "A" y "B" respectivamente.

XINIX puede ser visto como un medio ambiente para el desarrollo de aplicaciones concurrentes. Una vez que están listas, el usuario ejecuta a XINIX en respuesta al prompt de DOS:

```
B>xinix
```

...después de oprimir la tecla ENTER (<↓>), el usuario se encontrará en XINIX, frente al...

2.2 INTERPRETE DE COMANDOS

LOGIN

Antes de aceptar cualquier comando, el intérprete solicita al usuario una identificación (login en inglés), que asocia con la terminal utilizada. Si un usuario desea saber quién está en que terminal, puede utilizar el comando who, que se encarga de desplegar la identificación de terminales activas y de usuarios que las utilizan.

Después de recibir el login, el intérprete despliega el "prompt" al usuario, indicándole que está listo para recibir comandos. El prompt se compone de la identificación de la terminal seguida de 2 caracteres "mayor que" (>>). Si hay 2 terminales, la de memoria mapeada (CONSOLE) y una RS232 (OTHER_1), el prompt que cada una despliega es:

```
CONSOLE>>      : Y  
OTHER_1>>
```

respectivamente. En el caso de conectar más terminales RS232, éstas desplegarían los prompts "OTHER_2>> ", "OTHER_3>> ",... Ver el apéndice A para conocer las características de la transmisión en las terminales XINIX RS232.

MANEJO DE ARCHIVOS

El intérprete de comandos ofrece varios comandos al usuario para manejar archivos.

Copiar un archivo a otro se logra con los comandos `cp` o `cat`, usándolos de la siguiente manera:

```
CONSOLE>> cp file1 file2
CONSOLE>> cat file1 > file2
```

Ambos comandos copian el contenido del archivo `file1`, si existe, al archivo que se nombrará `file2`; siempre y cuando este último no exista. No se permite la sobreposición del contenido de un archivo a otro. Más si se desea utilizar un nombre en particular,...

Renombrar un archivo se hace el comando `mv`, y se utiliza:

```
OTHER_1>> mv oldname newname
```

El archivo `oldname` ahora se llama `newname`.

Borrar un archivo se hace con el comando `rm` (remove en inglés), se utiliza:

```
OTHER_1>> rm filename
```

Desplegar el contenido de un archivo por la terminal se realiza con el comando `cat`, utilizado de la siguiente manera:

```
CONSOLE>> cat filename
```

Listar el directorio de un diskette XINIX lo realiza el comando `ls`. La versión actual de XINIX sólo puede ver al manejador de disco A, así que `ls` no necesita argumentos adicionales para especificar un manejador en especial; además, no es posible listar sólo los archivos que cumplan con características específicas en el nombre o extensión de los archivos, tal como sucede en DOS; así que `ls` se utiliza:

```
OTHER_1>> ls
```

De cada archivo en el diskette, sólo se despliega su nombre y tamaño en bytes; aún no se maneja fecha y hora en el sistema de archivos, y por lo tanto, no es posible registrar la fecha y hora de la última actualización realizada a un archivo.

Formatear un diskette se hace con el comando `fmt`, y es necesario antes de poder manejar archivos en archivos XINIX en un diskette.

```
CONSOLE>> fmt
```


Al igual que ls, no necesita parámetros adicionales, pues sólo actúa sobre el manejador de disco A.

Transportar archivos DOS a XINIX se logra con el comando cdx, se utiliza:

```
OTHER_1>> cdx dosfile xinixfile
```

El intérprete guía al usuario para insertar los diskettes DOS y XINIX en el manejador de disco A. Este servicio es muy útil; supóngase una aplicación en la que el usuario desea leer datos desde un archivo, y que la manera más fácil de prepararlos es con un editor de texto. Ya que XINIX no tiene editores de texto, el usuario puede preparar su archivo en DOS, con el de TurboC o Word Star, y transportarlo a XINIX para que su programa lo accese.

EJECUCION DE PROGRAMAS, MANEJO DE PROCESOS

Un programa es un archivo en disco que contiene, o es, la imagen ejecutable de una aplicación. El programa pasa a ser un proceso una vez que se le da de alta en la tabla de procesos de XINIX; lo que implica que se ha transferido de disco a memoria, relocalizado, y que puede competir por el procesador.

Ejecución de programas en archivos XINIX

Si el archivo ejecutable se encuentra en un diskette XINIX (se ha transferido desde DOS), puede ejecutarse de una de las siguientes maneras:

```
CONSOLE>> exec programa
```

ó

```
CONSOLE>> create programa  
CONSOLE>> resume id_program
```

En la primera, con el comando exec, se lee el programa del diskette, se carga en memoria, se relocaliza y se le pone en estado READY: compite por el procesador según su prioridad. Create hace exactamente lo mismo que exec, excepto que no pone al programa en el estado READY, sino SUSPEND: no inicia su ejecución. El usuario puede ponerlo en READY con el comando resume, él cual recibe la identificación que XINIX asignó al programa al momento de crearlo: su entrada en la tabla de procesos. Ver el comando ps.

Ejecución de programas en archivos DOS

Opcionalmente, con el fin de evitar la transferencia de su programa-archivo a formato XINIX, el usuario puede ejecutar programas en archivos DOS; también existen dos maneras:

```
CONSOLE>> execd programa
```

ó

```
CONSOLE>> created programa  
CONSOLE>> resume id_program
```

Execd equivale a exec, y created a create. La identificación id_program, que recibe el comando resume, se obtiene desplegando el...

Estado de los procesos, con el comando:

```
OTHER_1>> ps
```

De cada proceso en la tabla de procesos, ps despliega su identificación de proceso, que corresponde a su número de entrada en la tabla de procesos; también despliega su nombre, su estado (READY, SUSPEND, WAIT, RECEIVE, SLEEP o CURRENT), su prioridad, dirección en memoria donde se encuentra su pila, la longitud de ésta y cuanto se ha ocupado, el número de semáforo por el que espera en caso de estar en estado WAIT, y si el proceso se encuentra en estado RECEIVE, y ha recibido un mensaje, entonces se despliega la identificación del mensaje.

Prioridad, cambio de. El usuario puede desear cambiar la prioridad de un proceso, para lo cual utiliza el comando chprio:

```
CONSOLE>> chprio id_process newprio
```

Recuérdese que XINIX asigna el procesador de acuerdo a la política ROUND-ROBIN: a todos los procesos READY con la prioridad actual más alta, les asigna un intervalo de tiempo, de tal manera que el proceso que recibe por primera vez un intervalo, debe esperar a que los otros procesos (con su misma prioridad) reciban un intervalo antes de él recibir su segundo intervalo; los procesos READY con prioridad menor que la mayor actual son totalmente ignorados.

Terminación de un proceso, con el comando kill:

```
CONSOLE>> kill id_process
```

Kill termina a un proceso sin importar el estado en el que se encuentre. Libera la memoria y la entrada en la tabla de procesos que ocupa. Además de crear procesos a partir de archivo-programas, es posible crearlos a partir de procedimientos en el mismo archivo, esto se aclarará una vez que llegemos al apartado 2.2: ejemplos. Por otra parte, en esta versión de XINIX no se puede terminar a un proceso al oprimir las teclas "Ctrl" y "C" simultáneamente, tal y como sucede en DOS.

SALIDA DEL SISTEMA

Para salir del intérprete, el usuario utiliza los comandos:

```
CONSOLE>> exit
```

ó

```
CONSOLE>> logout
```

MISCELANEA

Se ha ejemplificado el uso de los comandos básicos, que el usuario debe conocer para realizar una sesión en XINIX. Si durante la sesión el usuario olvida la sintáxis de un comando, puede teclearlo sólo, sin argumentos, y el intérprete desplegará la sintáxis y en algunas ocasiones, comentarios adicionales.

Más si se olvida el nombre de un comando, teclear:

```
CONSOLE>> help
```

ó

```
CONSOLE>> ?
```

... y el intérprete desplegará la lista de comandos disponibles. Otros comandos son: mem, informa sobre la distribución y uso de la memoria; who, lista las terminales activas y los usuarios que las utilizan; devs, lista la identificación y atributos de los dispositivos manejados: terminales, discos y archivos. La lista completa de los comandos y servicios XINIX, según su tipo, se da en el capítulo 3, en el apéndice C se encuentran en orden alfabético, mostrando como se utilizan.

2.3 SERVICIOS INTERNOS. EJEMPLOS DE UTILIZACION

Se ha descrito a XINIX como un medio ambiente para el desarrollo de aplicaciones concurrentes. En verdad el desarrollo es en DOS, utilizando el compilador TurboC, aunque la ejecución si es en XINIX. La relación exacta entre DOS, XINIX y el programa del usuario se da en el capítulo 3. Por ahora, el usuario sólo necesita saber que cuenta, como si se tratara de procedimientos en la librería de TC, de los procedimientos-servicio que en esta parte se explican, y que en caso de usarlos, sólo sirven si su programa se ejecuta en XINIX; más aún, sólo en XINIX puede correr su programa.

2.3.1 Servicios para el manejo y coordinación de procesos

CREACION de procesos

Supongase el siguiente archivo-programa de usuario:

```
/* ej1.c: main, produce, consume */
#include "xinixh.b"

int n = 0; /* variables son compartidos globales */
          /* por todos los procesos */

/*-----
 * main - ejemplo de procesos productor-consumidor no sincronizados
 *-----
*/

main()
{
    int produce(), consume();

    resume(create(consume, 200, 5, "cons", 0));
    resume(create(produce, 200, 5, "prod", 0));
}

/*-----
 * produce -- incrementa n 2000 veces y termina
 *-----
*/

produce()
{
    int i;

    for ( i=1; i <= 2000; i++)
        n++;
}

/*-----
 * consume -- imprime n 2000 veces y termina
 *-----
*/

consume()
{
    int i;

    for ( i=1; i <= 2000; i++)
        printf("n is %d\n", n);
}
```

Suponiendo que el archivo ejecutable se llama `ej1.exe`, el usuario lo ejecutaría con el comando `execd` de la siguiente manera:

```
CONSOLE>> execd ej1.exe
```

Los procesos creados a partir de archivos ejecutables inician su ejecución en su procedimiento `main()`. Nótese que en `ej1.c`, `main()` nunca llama a los procedimientos `produce()` o `consume()`; en su lugar, de cada uno crea un proceso que corre en forma independiente, a menos que entable coordinación con otro proceso.

Para crear un proceso a partir de un procedimiento, el usuario utiliza el servicio `create()`, que recibe la dirección del procedimiento (en lenguaje C es su nombre sin paréntesis), el tamaño, en palabras, de la pila del proceso (200), su prioridad (5), su nombre ("`cons`", "`prod`"), y el número de parámetros que recibirá al ser creado, 0 en este caso.

`Create()` deja al proceso creado en estado SUSPENDIDO, pero devuelve su identificación, la cual utiliza `resume()` para ponerlo en estado READY. No debe confundirse el comando `resume` con el servicio `resume()`. El primero se construye sobre el segundo.

En `ej1.c`, la variable `n` es global, y por lo tanto utilizable por todos los procedimientos en el archivo. `Produce()` la incrementa 2000 veces, mientras que `consume()` imprime su valor 2000 veces.

¿ Que pasa al correr `ej1.exe` ?

La mayoría de los programadores pensarán: el proceso `cons`, constituido por `consume()`, imprimirá por lo menos unos cuantos, quizá todos, los valores entre 0 y 2000. Esto no es verdad. En una corrida típica, `n` tiene el valor 0 pocas veces y después su valor es 2000. Aunque los dos procesos, corren concurrentemente, ellos no requieren la misma cantidad de tiempo en cada iteración. El proceso `cons` debe formatear y escribir una línea de salida, una operación que requiere cientos de instrucciones de máquina. Aunque el formateo es tardado, lo que toma más tiempo son las operaciones de salida. Rápidamente, `cons` llena los buffers de salida, y debe esperar a que el manejador de terminal los envíe a la consola para poder continuar. Mientras `cons` espera, `prod`, constituido por `produce()`, corre; puesto que ejecuta pocas instrucciones por iteración, realiza las 2000 iteraciones en el corto tiempo que toma al manejador imprimir pocos caracteres. Cuando `cons` reanuda la ejecución, encuentra que `n` tiene el valor 2000.

SINCRONIZACION entre procesos

Para que `cons` imprima todos los valores entre 0 y 2000, es necesario sincronizar o coordinar su ejecución con la del proceso `prod`. Esto es posible a través del uso de semáforos. Se recordará que un semáforo tiene como atributo una cuenta asociada: un valor entero. En XINIX, el usuario especifica el valor inicial de esta cuenta al momento de crear el semáforo. El servicio `wait()` decrementa la cuenta de un semáforo, y si ésta resulta negativa, entonces bloquea al proceso que ejecutó `wait()`. `Signal()` realiza la operación contraria: si la cuenta de un semáforo es negativa desbloquea un proceso, e incrementa el valor la cuenta.

La sincronización de `prod` y `cons` necesita dos semáforos; en uno ha de esperar (bloquearse) `cons`, y en el otro `prod`. Los semáforos se crean dinámicamente con el servicio `screate()`, que recibe como argumento la cuenta inicial del semáforo, y devuelve un valor entero con el que se le identifica.

En el ejemplo 2 listado a continuación, el proceso `main` crea 2 semáforos, `consumed` y `produced`, cuya identificación la pasa como argumentos a los procesos que crea. Puesto que el semáforo `produced` inicia con una cuenta igual a 1, `wait()` no bloqueará la primera vez que es llamado en `cons`, constituido ahora por el procedimiento `cons2()`. Así que `cons` imprime el valor inicial de `n`. El primer `signal()` en `cons` encuentra un valor no negativo (0) en el semáforo `consumed`, por lo que no desbloquea a nadie, más si se incrementa su cuenta. El llamado a `wait()` en la segunda iteración en `cons` encuentra al semáforo `produced` en 0, por lo tanto, `cons` es bloqueado. En este momento la cuenta de `produced` es -1, y la de `consumed` es 1. El control pasa a `main`, que crea y pone en ejecución a `prod`, constituido ahora por procedimiento `prod2()`. `Prod` no es bloqueado en el primer `wait()` que realiza, pues `consumed` vale 1; así que incrementa a `n`. Sin embargo, al ejecutar `signal()`, `prod` encuentra un valor negativo en la cuenta del semáforo `produced`, por lo que desbloquea a `cons`. `Cons` continúa justo después del `wait()` que lo bloqueó, en la instrucción "`printf(...)`", la que ahora desplegará "`n` es 1". `Cons` se bloquea de nuevo en el `wait()` de la siguiente iteración, entonces `prod` incrementa `n`, desbloquea a `cons`, éste imprime, se bloquea, etcétera. Si se corre `ej2.c`, se verificará que ahora `cons` si imprime todos los valores de `n` entre 0 y 1999.

```
/* ej2.c; main, prod2, cons2 */
#include "xinxh.h"

int n = 0; /* variables globales son compartidas a todo proceso */

/*-----
 * main -- ejemplo de procesos productor - consumidor sincronizados
 *-----
 */

main()
{
    int prod2(), cons2();
    int produced, consumed;

    consumed = screate(0);
    produced = screate(1);
    resume(create(cons2, 200, 20, "cons", 2, consumed, produced));
    resume(create(prod2, 200, 20, "prod", 2, consumed, produced));
}

/*-----
 * prod2 -- incrementa n 2000 veces esperando a que se consuma
 *-----
 */
```

```

prod2(consumed, produced)
{
    int i;

    for ( i=1; i <= 2000; i++) {
        wait(consumed);
        n++;
        signal(produced);
    }
}

/*-----
 * consume -- imprime n 2000 veces, espera a que se produzca
 *-----*/

cons2(consumed, produced)
{
    int i;

    for ( i=1; i <= 2000; i++) {
        wait(produced);
        printf("n is %d\n", n);
        signal(consumed);
    }
}

```

EXCLUSION MUTUA

Los semáforos proveen otra función importante en XINIX: "exclusión mutua". Considérese la siguiente situación. Cuando un proceso quiere imprimir un archivo mete el nombre de éste en un arreglo de impresión especial. Otro proceso, el impresor, checa periódicamente si hay archivos para imprimir; si los hay, los imprime y borra sus nombres del arreglo. Tómese en cuenta que los procesos comparten al arreglo de impresión, y también a las variables `sale`, que apunta al siguiente archivo a imprimir, y `mete`, que apunta a la siguiente entrada libre en arreglo de impresión. En cierto momento, las entradas 0 a 3 están vacías (los archivos se han imprimido) y las entradas 4 a 6 están ocupadas (con los nombres de los archivos enlistados para imprimirse). Entonces, mas o menos simultáneamente, los procesos A y B quieren imprimir un archivo. Ver la figura 2.1.

Lo siguiente puede pasar. El proceso A lee a `mete` y almacena su valor, 7, en una variable local llamada `siguiente_entrada`. Justo entonces ocurre una interrupción de reloj; el SO detecta que se ha terminado el `quantum` del proceso A; así que pasa el control al proceso B. El proceso B también lee `mete` y obtiene un valor 7, así que guarda el nombre de su archivo en la entrada 7 y actualiza `mete` a 8; entonces procede a realizar cálculos. Eventualmente el proceso A corre de nuevo reiniciando donde se quedó: toma el valor de `siguiente_entrada`, un 7, y escribe su nombre de archivo en la entrada 7, borrando el nombre del archivo que el proceso B escribió anteriormente. Entonces calcula

siguiente_entrada + 1, lo que da 8, y deja a mete igual a 8. El proceso impresor no detecta nada erróneo, pero el proceso B nunca obtendrá su impresión.

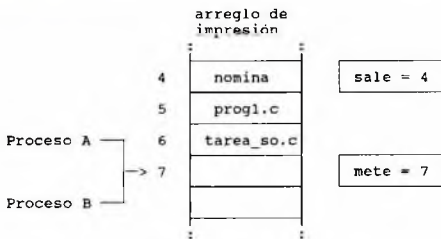


Figura 2.1. Dos procesos quieren utilizar memoria compartida al mismo tiempo.

Sincronización en este caso no es la respuesta, pues los procesos no necesitan alternar el acceso a las variables compartidas, sino excluir su acceso a éstas.

La exclusión se logra creando un semáforo S con una cuenta inicial igual a 1. Antes de utilizar las variables compartidas todo proceso ejecuta wait(S); y signal(S) cuando termine. El llamado a wait() y signal() puede colocarse al principio y al final, respectivamente, de un procedimiento que realiza la actualización. El código siguiente muestra un arreglo compartido y un procedimiento para poner datos en él en forma exclusiva: terminando la actualización de las variables compartidas.

```

/* ej3.c - additem */

int mutex; /* se asume inicializado asi: mutex = screate(1); */
int a[100]; /* las variables globales son compartidas */
int n = 0;

/*-----
 * additem - obtiene acceso exclusivo al arreglo 'a' y le escribe item
 *-----
 */

additem(item)
int item;
{
    wait(mutex);
    a[n++] = item; /* realmente es: a[n] = item; y n++; */
    signal(mutex);
}

```


COMUNICACION entre procesos, MENSAJES CORTOS

En XINIX varios procesos pueden comunicarse con "mensaje cortos", utilizando los servicios receive() y send(). Receive() provoca que el proceso que lo llama sea "bloqueado" en caso de no haber recibido un mensaje; de otra manera lo toma y continúa. El mensaje se recibe en el campo pmsg, en la entrada de cada proceso en la tabla de procesos (TP). Para enviar un mensaje, se llama a send(), pasándole como parámetros la identificación del proceso al que va dirigido el mensaje y el mensaje. El código que sigue es la versión con mensajes de los procesos productor y consumidor.

```
/* ej4.c; main, cons3, prod3 */
#include "xinix.h"

int n = 0;          /* variables globales son compartidas */
                   /* por todos los procesos */

/*-----
 * main -- procesos productor-consumidor sincronizados con mensajes
 *-----
*/

main()
{
    int id_cons;          /* identificacion del consumidor */
    int cons3(), prod3();

    resume( (id_cons = create(cons3, 200, 20, "cons", 0)) );
    resume( create(prod3, 200, 20, "prod", 1, id_cons));
}

/*-----
 * prod3 -- incrementa n 2000 veces, la envia como mensaje
 *-----
*/

prod3(id_cons)
{
    int i;

    for ( i=1; i <= 2000; i++) {
        while (send(id_cons, n) != OK)
            ;
        n++;
    }
}

/*-----
 * cons3 - imprime n 2000 veces, espera que la envíen
 *-----
*/
```

```

cons3()
{
    int i;

    for ( i=1; i <= 2000; i++)
        printf("n is %d\n", receive());
}

```

Primero se crea al proceso cons, ahora constituido por el procedimiento cons3(). Cuando cons ejecuta a receive() queda bloqueado, pues nadie le ha enviado un mensaje. Entonces main() crea al proceso prod, ahora constituido por prod3(), y le pasa como parámetro la identificación de cons, en id_cons. Cuando prod llama a send(), permanece en la instrucción while hasta que el envío sea correcto (== OK; OK es una constante definida en xinix.h). Si la identificación del proceso al que se envía un mensaje es errónea, o la entrada en la tabla de procesos que corresponde a id_cons está libre (no registra a un proceso), o ya existe un mensaje; send() regresa el valor SYSERR (también definido en xinix.h); de otra manera, send() deposita el mensaje en pmsg, y que este existe, lo manifiesta al poner el valor TRUE (también definido en xinix.h), en el campo phasmgs en la entrada en la TP del proceso que recibe el mensaje. Por último, send() revisa el estado del proceso al que envía el mensaje, si es PPRECV (esperando), entonces lo desbloquea: lo pone en estado READY y provoca cambio de contexto.

Cuando cons, bloqueado en receive(), es desbloqueado por el send() de prod, recibe el mensaje en el nombre de la función, por eso es que receive() está en la instrucción printf.

Obsérvese que receive() bloquea, si es el caso, pero send() no, por eso se deja en la instrucción while, para que salga de ella hasta que deposite bien el mensaje. Por otra parte, el usuario apreciará con la práctica la riqueza de este mecanismo, pues un proceso bloqueado con receive() puede condicionar su ejecución en base al mensaje recibido. Un ejemplo de esta situación corresponde a la implementación del manejador de diskette, cuando espera un mensaje, por parte del FDC, que le indica la completitud de una operación requerida. El mensaje también puede llegar, 3 segundos después, por parte de la rutina que atiende las interrupciones de reloj, en cuyo caso el mensaje, por tener otro valor, se intérpreta como "manejador de diskette no listo": se abrió la puerta del diskette y la transferencia de información no se completó; se pide entonces opción al usuario con el mensaje:

```

Not ready error reading drive A
Abort, Retry, Ignore?

```

... según la opción tecleada, el manejador de diskette procede a intentarlo de nuevo o a cancelar la operación.

Varios usuarios pueden objetar el comportamiento de los "mensajes cortos": el proceso que envía el mensaje consume su tiempo en revisar que éste llegue bien, de otra manera el mensaje se pierde; o, sólo puede enviarse un valor entero como mensaje. Además, si se deseará pasar más información a un proceso, lo conveniente sería enviar la dirección donde se encuentra el mensaje; más considerando que el modelo

de memoria utilizado al compilar XINIX, HUGE de TurboC, trabaja con apuntadores largos y el tamaño de éstos es el de dos valores enteros; pasar la dirección de un mensaje, con mensajes cortos, requeriría de dos eventos receive()-send().

Afortunadamente, esto no es necesario, pues XINIX tiene otro mecanismo para transferir información entre procesos, estos son los...

MENSAJES LARGOS, puertos

A diferencia de los mensajes cortos, los largos no se guardan en la tabla de procesos. En su lugar, se crea un puerto para transmitir información. Un puerto es una de estructura de datos, parecida a los semáforos, que tiene asociada una lista de hasta n mensajes. En un principio no hay mensajes, y se encolan conforme van llegando. El "ej5.c" aclara el uso de los "mensajes largos". El proceso printer desplegará, continuamente, los mensajes "msg_1" a "msg_8" declarados en main(), recibiendo únicamente su dirección.

```
/* ej5.c; main, printer */
#include "xinix.h"
#define TRUE 1

printer(portid)
int portid;
{
    while (TRUE)
        printf("recibi: %s\n", preceive(portid) );
}

main()
{
    int printer();
    int i;
    int portid; /* identificacion del puerto de mensajes */
    char *msgs[] = {"msg_1", "msg_2", "msg_3", "msg_4",
                   "msg_5", "msg_6", "msg_7", "msg_8"};

    portid = pcreate(5); /* se crea el puerto con capacidad de 5 msgs */
    resume( create ( printer, 1024, 5, "printer", 1, portid ));
    while ( TRUE )
        for ( i=0; i < 8; i++)
            psend(portid, msgs[i]);
}
```

Todo empieza en main. Primero se crea el puerto con capacidad de 5 mensajes, su identificación queda en portid. Portid se pasa como parámetro inicial al proceso printer.

Si se corre "ej5.c", se podrá comprobar que printer imprime los mensajes "msg_1", "msg2", ..., "msg_8", "msg_1", "msg_2",..., "msg_8",... continuamente, aún cuando en cada iteración printer, por el formateo y la escritura a dispositivo a través de printf(), consume más tiempo que main que sólo envía la dirección.

La sincronización se logra gracias a la manera como operan los puertos. Cuando un proceso llama a `preceive()` y existe un mensaje, `preceive()` regresa en seguida, devolviendo el mensaje en el nombre de la función (por eso se encuentra dentro del llamado a `printf` en `printer`); si por otra parte, no existe un mensaje, entonces `preceive()` bloquea al proceso que lo llamó. Si no hay mensajes y varios procesos leen de un puerto, quedarán bloqueados en el orden que llaman a `preceive()`.

`Printer` se bloquea la primera vez que llama a `preceive()`, pues `main` le pasó el control con `resume()` justo antes de enviarle un mensaje. El bloque de `printer` regresa el control a `main` en la instrucción `"while(TRUE)"`, donde éste empieza a enviar mensajes con `psend()`.

A diferencia de `send()`, `psend()` bloquea al proceso que lo llama cuando la capacidad del puerto (5 en este caso) se alcanza, esto evita usar al procesador en la revisión de un envío correcto. Un proceso bloqueado por `psend()` se desbloquea cuando, otro proceso lee un mensaje del puerto, hace espacio en el puerto, y así permite al proceso bloqueado poner su mensaje y continuar su ejecución. Al igual que en `preceive()`, varios procesos pueden enviar mensajes a un mismo puerto, y si no hay espacio, quedar bloqueados según el orden en que lo intentaron. El primer proceso que se bloquee será el primero que se libere una vez que haya espacio en el puerto.

Cabe mencionar que en el ejemplo anterior se pasa la dirección de un arreglo de caracteres a desplegar, con el único fin de que el usuario aprecie la operación de los puertos, más el mensaje puede ser el apuntador a una estructura más compleja que contenga otros apuntadores, arreglos, variables etcétera.

PROCESOS DORMILONES, servicios de reloj

Por una razón particular, un proceso puede necesitar que se le suspenda "temporalmente". El manejador de diskette XINIX, al realizar una transferencia de información, debe asegurarse que el motor del diskette está girando; si no es el caso, lo pone a andar, y espera 3 décimas de segundo a que el motor alcance su velocidad de trabajo. Tal espera la realiza con una "suspensión temporal", utilizando el servicio `sleep10(3)`.

XINIX provee los servicios `sleep()` y `sleep10()` para retardar la ejecución del proceso que los llame. Ambos servicios reciben como parámetro un valor entero, que especifica cuantas unidades de tiempo será retardado el proceso que les llama. Las unidades que `sleep()` maneja son segundos, `sleep10()` maneja décimas de segundo.

Para terminar la parte que corresponde al "manejo y coordinación de procesos", se mencionará que existen otros servicios que complementan los ya ejemplificados. Por ejemplo, `get_priority()` y `get_status()` nos devuelven, respectivamente, la prioridad y `status` actual de un proceso; ambos reciben como parámetro la identificación del proceso: `scout()` devuelve la cuenta asociada a un semáforo, y `pcount()` la asociada a un puerto (número de mensajes que hay). En el capítulo 3, manual del usuario, se da la lista de servicios por orden alfabético y por tipo de servicio.

2.3.2 Servicios para el manejo de memoria

MANEJO SIMPLE, getmem y freemem

OBTENCION de memoria

Muchas aplicaciones se diseñan para manejar dinámicamente la memoria que utilizan. En XINIX, un proceso puede solicitar y liberar memoria durante su ejecución, usando los servicios 'getmem' y 'freemem' respectivamente. El código que sigue muestra como utilizar 'getmem':

```
char *blkp;
*
if ( (blkp = getmem(nbytes)) == SYSERR) {
    "ACCION EN CASO DE NO OBTENER MEMORIA"
}
*
```

Getmem() recibe el número de bytes que el usuario desea, y en caso de existir memoria, devuelve la dirección de un bloque con capacidad igual o poco mayor; de otra manera, devuelve el valor SYSERR, definido en el archivo xinix.h que el usuario debe incluir en sus aplicaciones. La cantidad de bytes a pedir no tiene límite, y en caso de pasarlo en una variable, puede utilizarse un entero tipo long. Es conveniente que el usuario "siempre" cheque por la dirección que getmem() devuelve, tal como se muestra en el código anterior. Si así lo hace, al compilar su programa, el usuario obtendrá los siguientes mensajes por parte del compilador:

```
Suspicious pointer conversion in function X, y
Non portable pointer comparison in function X
```

El primero se debe a que getmem() está definido para devolver un apuntador tipo entero, y la variable blkp es definida como un apuntador, pero tipo char. El segundo mensaje es por la comparación del apuntador blkp con la constante simbólica (tipo entero) SYSERR. Si el usuario desea eliminar estos mensajes, puede codificar la petición así:

```
if ( (blkp = (char *)getmem(nbytes)) == (char *)SYSERR) {...
    ..haciendo un "cast" del valor que regresa getmem y de la
    constante SYSERR, al tipo de apuntador que recibirá la dirección del
    bloque de memoria. Si el apuntador lo es a una estructura, digamos
    "nodo", la codificación queda:
if ((blkp =(struct nodo *)getmem(nbytes)) ==(struct nodo *)SYSERR) {...
```

LIBERACION de memoria

Para liberar memoria obtenida, el usuario utiliza a `freemem()` de la siguiente manera:

```
freemem(blkp, nbytes);
```

`Freemem()` recibe la dirección y longitud del bloque de memoria a devolver, por lo que es conveniente que el usuario solicite la memoria así:

```
char *blkp, *blkpr; /* apuntador al bloque, apuntador de respaldo */
*
if ( (blkpr = blkp = getmem(nbytes)) == SYSERR) {
    "ACCION EN CASO DE NO OBTENER MEMORIA"
}
*
freemem(blkpr, nbytes);
*
*
```

De esta manera, ambos, `blkpr` y `blkp` contienen la dirección del bloque de memoria obtenido, y `blkp` (o `blkpr`) puede modificarse, y ser el apuntador de trabajo que se mueva a través del espacio del bloque.

MANEJO CONTROLADO, despensas (pools) de memoria

Una pool es un conjunto de bloques contiguos de memoria. Al crear un pool se especifica el número de bloques y el tamaño, en bytes, de cada bloque; los bloques quedan ligados como se muestra en la figura 2.2.

```
int bufsize;
int numbufs;
int pool_id;
*
*                               /* CHK64KB */
if ( (pool_id = mkpool(bufsize, numbufs, NCHK64KB)) == SYSERR) {
    ACCION EN CASO DE NO CREARSE EL POOL
}
*
*
```

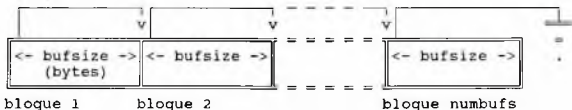


Figura 2.2. Pools de memoria.

Es conveniente que el usuario revise si fué o no creado el pool, tal como se muestra en el código, pues si no existe la memoria suficiente el pool no se crea. En cuanto a los parámetros que recibe `mkpool()`, `bufsize` y `numbufs` deben ser tipo entero, el primero debe ser mayor o igual que 2 (bytes) y menor o igual que 4608; mientras que el número de bloques debe ser mayor o igual a 1 y menor o igual a 100. El tercer parámetro, `CHK64KB` o `NCHK64KB`, es el valor verdadero (1) y falso (0) respectivamente, definidos como constantes simbólicas en el archivo `xinxix.h`, que debe incluir el usuario. Los primeros 2 parámetros son obvios, más el tercero requiere una explicación adicional. La causa es la arquitectura de la PC. La PC utiliza un chip especial (el DMA: Direct Memory Access) para transferir datos desde la memoria a las unidades de disco y viceversa. La dirección inicial de memoria que recibirá, o de la cual se leerán los datos, se carga en registros de control del DMA. La dirección en los registros se incrementa con cada byte transferido, apuntando así al siguiente byte de memoria a transferir. También en un registro del DMA se escribe el número de bytes a transferir. Las direcciones en la PC son de 20 bits de longitud, con los cuales se puede direccionar hasta 1 Megabyte de memoria:

$$2^{20} = 1\ 048\ 576$$

La dirección inicial que el DMA recibe para realizar la transferencia es de 20 bits, sin embargo, el DMA la maneja internamente según la figura 2.3.

Parte fija	Parte que se incrementa por c/byte transferido
bit 20 19 18 17	16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

Figura 2.3. Dirección escrita en el DMA.

Con 16 bits se pueden direccionar 64K bytes ($2^{16} = 65536$), por lo tanto, los bits 20-17 nos especifican el número de bloque, de 64K, en el que nos encontramos dentro de todo el Mega que puede direccionar la PC. La figura 2.3 ayuda a comprender esto.

¿Qué va a pasar cuando al DMA se le especifique una dirección (hexadecimal) tal como: A-F-F-F-0, y se le indique transferir 512 bytes (digamos a un sector físico del diskette) ?

Bueno, después de transferir el 1er byte, la dirección de transferencia se incrementa en 1, quedando: A-F-F-F-1. Después de transferir 13 bytes más, la dirección de transferencia apunta al byte A-F-F-F-F en memoria. Al transferir este byte, uno espera que la dirección de transferencia quede B-0-0-0-0: apuntando al siguiente byte. Sin embargo, no es así, pues como se mencionó, el DMA no actualiza los últimos 4 bits de la dirección de transferencia; o lo que es lo mismo, no existe acarreo entre los primeros 16 bits y los últimos 4. La dirección B-0-0-0-0 esperada, realmente llega a ser A-0-0-0-0, al inicio del bloque de 64K con el que se empezó.

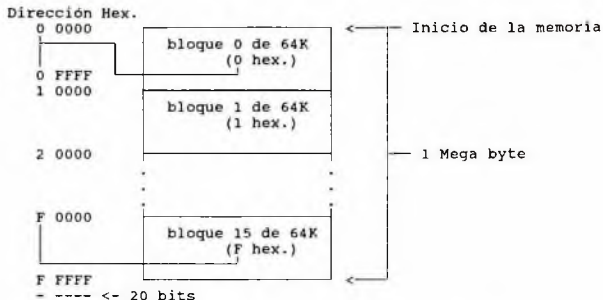


Figura 2.3. Direcciones de memoria en la PC.

El usuario puede imaginar las inconsistencias que provocaría ignorar esta situación.

XINIX, su manejador de disco, utiliza 3 pools de memoria para realizar operaciones con la unidad de diskette de la PC. Dos de ellos los utiliza para transferir datos, por lo tanto, estos 2 pools (los bloques que los componen) no deben quedar en la frontera de 2 bloques físicos de 64K en el espacio de memoria de la PC.

Al crear un pool, `mkpool()` recibe el tercer parámetro con valor falso o verdadero. Si el valor es falso (`NCHK64KB`), el pool se crea sin revisar si la memoria que forma al pool atraviesa una frontera de 64K; más si el tercer parámetro es `CHK64KB` (un valor verdadero), entonces `mkpool()` crea al pool contenido "todo" en un bloque de 64K del espacio de memoria.

Según la aplicación del usuario, puede ser conveniente manejar o no un pool en la frontera de un bloque físico de 64K. Sólo cabe mencionar que existen chips para transmisión de datos que se comportan exactamente igual que el DMA, y que en tal caso, el usuario cuenta con la manera de resolver el problema, con un sólo parámetro.

Por último, `mkpool()` solicita a `getmem()` la memoria que utilizará un pool, por lo que es conveniente que antes de hacer cualquier cosa que consuma memoria, el usuario cree los pools que va a utilizar en su aplicación.

OBTENCIÓN y LIBERACIÓN de un bloque en un pool

Una vez creado el pool, el usuario puede solicitar o liberar un bloque de éste con `getbuf()` y `freebuf()` respectivamente; se recomienda hacerlo según el código a continuación:


```

char *blkp, *blkpr; /* apuntador al bloque, apuntador de respaldo */
int bufsize;
int numbufs;
int pool_id;
:
: /* CHK64KB */
if ( (pool_id = mkpool(bufsize, numbufs, NCHK64KB)) == SYSERR) (
    ACCION EN CASO DE NO CREARSE EL POOL
)
:
blkpr = blkp = (char *) getbuf(pool_id); /* OBTENCION */
:
freebuf(blkpr); /* LIBERACION */
:

```

El cast "(char *)" en la obtención, es para evitar el mensaje warning del compilador "Suspicious pointer conversion in function X". Nótese que no se revisa si se obtuvo o no un bloque. La revisión no es necesaria, pues el manejo de memoria en base a pools consiste en dar el bloque si es que hay; si no hay, el proceso que solicita es bloqueado, y liberado sólo cuando se recupera un bloque del pool: cuando otro proceso lo libere con freebuf(). Por lo tanto, el llamado a getbuf() siempre nos devuelve memoria.

Getbuf() recibe la identificación del pool del cual se desea un bloque, mientras que freebuf() recibe la "dirección inicial" del bloque a liberar; por eso se justifica obtener la dirección del bloque en 2 variables apuntador, una para trabajo y otra para el momento de liberarlo. En realidad, los bloques en un pool son un entero más grande que el tamaño especificado por el usuario al momento de crear el pool. Cuando éste es creado, en el entero adicional (el primero) de cada bloque, se guarda la identificación del pool al que pertenecen. Getbuf() devuelve en verdad la dirección del segundo entero de un bloque, mientras que freebuf() la disminuye, en la longitud de un entero, para obtener la identificación del pool al que pertenece el bloque devuelto y así encadenarlo correctamente.

2.3.3 Servicios para manejar dispositivos

Los dispositivos que manejan estos servicios son terminales y archivos, y el manejo comprende las operaciones de lectura, escritura y control. Estas últimas dependen del tipo de dispositivo, por ejemplo, es posible cerrar un archivo, pero no una terminal, o posicionarse en un byte específico en un archivo pero no en una terminal. A continuación se lista la implementación del comando CP, del intérprete de comandos XINIX, para ejemplificar el uso de estos servicios. CP copia el contenido de un archivo a otro.

```

/*-----
* x_cp - (copy command) copy one file to another
*-----
*/

```

```

COMMAND x_cp(stdin, stdout, stderr, nargs, args)
int stdin, stdout, stderr, nargs;
char *args[];
{
    char *buf;
    int from, to;
    int ret;
    int len;

    if (nargs != 3) {
        fprintf(stderr, "...usage: cp file1 file2\n");
        return(SYSERR);
    }
    if ( (from = open(DISK0, args[1], "ro")) == SYSERR) {
        fprintf(stderr, fmt, args[1]);
        return(SYSERR);
    }
    if( ( to = open(DISK0, args[2], "w")) == SYSERR) {
        xclose(from);
        fprintf(stderr, fmt, args[2]);
        return(SYSERR);
    }
    if ( ((long)(buf = (char *)getmem(512))) == SYSERR) {
        fprintf(stderr, "...no memory\n");
        ret = SYSERR;
    } else {
        while ( ( len = read(from, buf, 512)) > 0)
            write(to, buf, len);
        freecm(buf, 512L);
        ret = OK;
    }
    close(from);
    close(to);
    return(ret);
}

```

El comando CP recibe el nombre del archivo a copiar y del archivo destino en el elemento 1 y 2, respectivamente, del arreglo args[]. En realidad se recibe la dirección donde se encuentran las cadenas de caracteres.

El servicio open() abre un archivo, recibe la identificación del disco donde se encuentra, su nombre y el modo de abrirlo (ro: Read Only, w: Write). Si todo sale bien, open() devuelve la identificación asignada al archivo, SYSERR en caso contrario. La identificación obtenida se utiliza entonces en otros servicios para leer de, escribir en, o cerrar el archivo: read(), write() y close() respectivamente. Cabe mencionar que existen los servicios fprintf() y fscanf() para, respectivamente, realizar lectura y escritura formateada, con la sintaxis del lenguaje C, en archivos o terminales.

Por último, tómesese en cuenta que la identificación de un dispositivo en XINIX es un valor entero, tipo de dato de las variables from, to y stderr. Ver la sección 3.2 y el apéndice C.

2.3.4 Otros servicios

XINIX maneja 2 tipos de terminales, de memoria mapeada y RS232. Para las últimas se desarrollaron un conjunto de procedimientos para leer, escribir y controlar puertos de comunicación serie. Es posible programar un puerto (velocidad, paridad, etcétera) con una sola función: `initport()`; saber si se ha recibido o transmitido un carácter, con `readyr()` y `readyt()` respectivamente; deshabilitar la interrupción al sistema por recibir o transmitir caracteres, con `disablel()` y `disablet()` respectivamente, etcétera.

Estos procedimientos también están disponibles al usuario. El detalle de cómo y para qué usarlos se da en el apéndice C.

CAPITULO 3

UTILIZANDO A XINIX

Este capítulo es el manual del usuario XINIX dividido en 2 partes. En la primera se explica la manera de desarrollar una aplicación: se muestran los componentes necesarios, su interrelación y la manera de usarlos. La segunda parte es una lista de los comandos y servicios disponibles al usuario agrupada por el tipo de objeto al que se aplican, una breve descripción acompaña a cada uno; se recomienda al usuario leer esta parte antes de desarrollar cualquier aplicación, con el único propósito de conocer las herramientas con que cuenta. La lista de los comandos y servicios por orden alfabético, incluyendo para cada uno sinopsis de uso (tipo de los parámetros que recibe, como debe llamarse y el tipo del valor que devuelve), se encuentra en el apéndice C.

Por otra parte, cada que en este capítulo se hable de una aplicación, imagen o programa ejecutable, se referirá a un programa "que correrá en XINIX", a menos que se indique lo contrario.

3.1 MEDIO AMBIENTE DE PROGRAMACION XINIX

La figura 3.1 muestra los pasos necesarios para desarrollar y ejecutar aplicaciones XINIX.

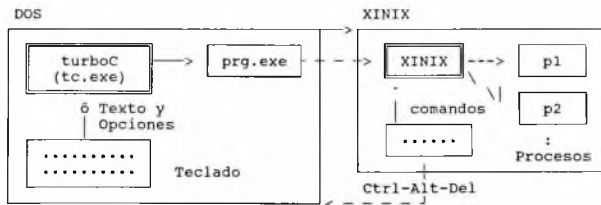


Figura 3.1. Medio ambiente de programación XINIX.

Todo empieza en DOS, el usuario captura su programa fuente, y crea el programa ejecutable con el compilador TurboC (TC) de Borland. Si son varios los programas que se van a correr, el usuario debe tener disponibles todas las imagenes ejecutables antes de pasar a XINIX, pues una vez en este no es posible ejecutar al compilador.

El usuario pasa de DOS a XINIX al ejecutar el programa xinx.exe, y de XINIX a DOS presionando, simultáneamente, las teclas Ctrl, Alt y Del (ver la figura 3.1).

Una vez en XINIX, el usuario puede crear procesos a partir de programas ejecutables DOS, los cuales correrán concurrentemente.

Para desarrollar y correr aplicaciones solo se necesita 2 diskettes de 360K bytes, que constituyen el medio ambiente de programación (versión austera; más adelante se señalan los requisitos para manejarlo en disco duro). A continuación se muestra el directorio del primer diskette, el etiquetado "XINIX A>":

```
Volume in drive A is XINIX A
Directory of A:\
```

```
INCLUDE      <DIR>          7-29-88   9:39a
ANSI         SYS          2716   10-08-85  12:08a
AUTOEXEC    BAT           41     7-29-88   9:38a
COMMAND     COM          23612   10-20-86  12:00p
CONFIG      SYS           43     12-09-88  11:52a
DISKCOPY    COM          4235   10-20-86  12:00p
TC          EXE         240996   1-25-88   1:05a
              7 File(s)      22528 bytes free
```

El directorio del segundo diskette, el etiquetado "XINIX B>", es:

```
Volume in drive A is XINIX B
Directory of A:\
```

```
LIB          <DIR>          7-29-88   9:45a
XINIX       EXE         91354   7-21-89   1:29p
TCCONFIG    TC           1711   8-18-88  11:48a
XINIXH      H            6435   6-27-89   5:28p
TCPICK      TCP           1196   1-26-89   7:16a
EJMSG       C            1529   4-20-89  12:36p
EJ1         C            1068   4-20-89  12:18p
EJ2         C            1434   4-20-89  12:13p
EJ4         C            1342   4-20-89  12:22p
EJ5         C             793   4-20-89  12:42p
EJ6         C             713   4-20-89  12:48p
EJ7         C            1235   4-20-89  12:34p
EJ1_2       C             699   4-20-89  12:20p
EJB         C             901   4-20-89  12:35p
              14 File(s)      102400 bytes free
```

INICIO DEL SISTEMA

Antes de encender la máquina, el usuario coloca el diskette "XINIX A>" en el manejador de disco A> (la entrada de arriba, o en algunas máquinas la de la izquierda); el diskette "XINIX B>" debe colocarse en el manejador de disco B> (la entrada restante). Con la aparición de los programas virus, se recomienda al usuario que siempre apague y encienda la terminal para iniciar el sistema.

En el diskette "XINIX A>", el archivo `command.com` es el intérprete de comandos de DOS; `ansi.sys` es el manejador de la terminal; `config.sys` contiene las características de DOS: cuántos archivos pueden estar abiertos al mismo tiempo, número de buffers de disco, la definición de otros manejadores para dispositivos físicos, etcétera; `diskcopy.com` permite copiar diskettes completos, se recomienda al usuario respaldar los discos XINIX antes de usarlos; `autoexec.bat` es un programa de comandos DOS. Cuando DOS termina de instalarse y configurarse, busca al archivo `autoexec.bat` en el diskette en A> y lo ejecuta. En XINIX, el contenido de `autoexec.bat` es:

```
echo=off
date
time
path a:\;b:\
b:
```

Al ejecutar un archivo de comandos (con extensión ".bat"), DOS despliega en la pantalla el comando que ejecuta; "echo=off", inhibe el desplegado. Los comandos `date` y `time` solicitan al usuario que introduzca la fecha y hora actual, respectivamente, para actualizar las del sistema. El comando `path` especifica a DOS en qué directorios buscar por un programa para ejecutarlo, en caso de que éste no se encuentre en el directorio actual del usuario. Por último, el comando `b:` define el prompt del sistema: B>; y además, el disco a considerar, en caso de omisión, cuando se efectuen operaciones con archivos.

PREPARANDO UNA APLICACION

Cuando DOS termina de ejecutar a `autoexec.bat`, el usuario se encuentra en el disco B. Suponiendo que su programa fuente se llama "ej1.c", y desea meter o modificar el texto y generar la imagen ejecutable correspondiente, entonces tendría que teclear:

```
B>tc ej1
```

Aun cuando el compilador TC, el archivo `tc.exe`, se encuentra en el disco A, DOS lo busca en los directorios definidos con el comando `path`, y al encontrarlo lo ejecuta. Si `ej1.c` (TC toma la extensión ".c" por omisión) ya existe, entonces lo carga; de otra manera, TC crea un nuevo archivo llamado "ej1.c". En realidad TC es un editor, un compilador y un encadenador integrados en un sólo programa; así que terminada la edición del programa fuente, la compilación y el encadenamiento se

realizan oprimiendo la tecla marcada F9 (opción make del compilador). Si no hay errores en el programa fuente del usuario, TC crea su imagen ejecutable: un archivo con el mismo nombre pero con extensión ".exe"; en nuestro ejemplo, TC crearía el archivo "ej1.exe". La imagen ejecutable queda en el diskette en B>.

Los archivos TCCONFIG.TC y TCPICK.TCP, en el disco B, son utilizados por TC. El primero especifica la configuración de TC: del compilador, del encadenador, y los directorios de trabajo. Estos últimos dicen a TC donde se encuentran las librerías del lenguaje C (subdirectorio LIB en el disco B); los archivos de macro-definiciones que acompañan a TC (subdirectorio INCLUDE en el disco A), etcétera. Por su parte, TCPICK.TCP contiene toda la información acerca de la última sesión; al punto en que el usuario puede teclear tan sólo:

B>tc

... y TC leerá primero a TCCONFIG.TC, luego a TCPICK.TCP, y de éste obtiene el nombre del último programa que el usuario editó, incluyendo las líneas que se estaban desplegando; así que lo carga y despliega al usuario las líneas del programa que aparecieron en pantalla por última vez. En [TCUG88] el usuario puede encontrar más acerca de como editar, compilar y encadenar programas con TC; también se explica ahí como manejar "proyectos" (útil cuando la aplicación del usuario es muy grande); y como definir los directorios de trabajo.

EL USUARIO NO DEBE MODIFICAR LA CONFIGURACION DE "TC" A MENOS QUE HAYA LEIDO Y COMPRENDIDO LA REFERENCIA CITADA.

EJECUTANDO UN PROGRAMA

Mientras un programa se encuentre en pruebas se recomienda ejecutarlo de la siguiente manera:

- 1)_No salir totalmente de TC. Oprimiendo la tecla F10, TC da el menu principal, con las teclas "flecha" colocarse en el rubro "File", oprimir la tecla ENTER, en este momento, TC desplegará una pantalla pequeña con varias opciones, con las teclas flecha colocarse en la opción "OS shell" y oprimir ENTER, en la pantalla aparecerá:

```
type EXIT to return to Turbo C . . .
```

```
Microsoft(R) MS-DOS(R) Version 3.20
```

```
(C)Copyright MicroSoft Corp 1981-1986
```

```
B>
```

Uno está de nuevo en DOS, pero TurboC permanece en memoria, ahora hay que...

- 2)_Copiar la imagen ejecutable del programa del usuario (mismo

nombre pero con extensión .exe) al disco A, por ejemplo:

B>copy xul.exe a:

Este paso es necesario, debido a que la versión actual de XINIX sólo maneja el diskette A. Una opción alterna es que el usuario intercambie la posición de los discos A y B. Se sugiere la primera.

3) Ejecutar XINIX:

B>xinix

En la pantalla aparecerá:

- The magic of XINIX

CINVESTAV-IPN_J.Buenabad (R) Time Sharing Operating System Version 1.0
Copyright (C) CINVESTAV_IPN. Julio 1988. All rights allowed.

CONSOLE

login: _

El usuario deberá teclear una identificación cualquiera para pasar al intérprete de comandos de XINIX. El que lo recibe con la siguiente leyenda:

Welcome to XINIX (type ? for help)

CONSOLE>>

CONSOLE>>

CONSOLE>>

La leyenda "CONSOLE>>" es el prompt de XINIX en la terminal de memoria mapeada, e indica al usuario que está listo para aceptar comandos. Ahora, para. . .

4) Ejecutar el programa del usuario, utilícese el comando:

CONSOLE>> execd ej1.exe

o bien, los comandos:

CONSOLE>> created ej1.exe

CONSOLE>> ps

CONSOLE>> resume id_proc

La primera forma crea del programa del usuario un proceso y lo

pone en ejecución. La segunda, con el comando `created`, crea el proceso, más lo deja suspendido. Con el comando `ps` se listan los atributos de los procesos en el sistema; así que el usuario reconocerá el nombre de su programa (ahora proceso), y tomará nota de que identificación le fue asignada (atributo `pid`), la que pasará al comando `resume`, que se encarga de reanudar a un proceso suspendido: lo pone en estado `READY`. Hay otros comandos para ejecutar programas, ver la parte 3.2 de este capítulo y el capítulo 2; sin embargo, la manera expuesta es la más fácil.

SALIENDO DE XINIX

Para salir de XINIX y volver a DOS sólo hay que oprimir, simultáneamente, las teclas `'Ctrl'`, `'Alt'` y `'Del'` en la terminal de memoria mapeada: la que despliega el prompt `"CONSOLE>> "`.

Ya en DOS, el usuario puede volver inmediatamente a TC con el comando `"exit"`, y modificar y compilar de nuevo su aplicación si acaso encontró errores. Si este es el caso, no debe olvidarse copiar otra vez, al disco `A>`, el nuevo programa ejecutable antes correr a XINIX.

XINIX VERSION DISCO DURO

Instalación

- 1)_Estando en DOS y en el disco donde XINIX se ha de transferir, crear un subdirectorio en el directorio raíz, `XINIXA` (XINIX Aplicaciones), si no hace conflicto con los del usuario, y colocarse en él:

```
C>cd\  
C>md xinixa  
C>cd xinixa  
C>
```

En lugar del disco duro `C>`, el usuario podría usar, si existen, al `D>` o al `E>`'. El comando `md` es la abreviación de `"make directory"`, y `cd` de `"change directory"`; `cd\
nos posiciona en el directorio raíz. El usuario ya debe tener instalado a TurboC en algún directorio de su sistema, y además, tener definida, con el comando path en el archivo autoexec.bat de su disco duro, una trayectoria al subdirectorio donde se encuentra TurboC, para así poder ejecutarlo desde el subdirectorio "xinixa".`

- 2)_Copiar todos los archivos del disco `"XINIX B>"` de la versión

económica, colocado en el manejador de disco A>, al subdirectorio "xinixa":

```
C>copy a:*.*
```

Deben copiarse los archivos

```
XINIX    EXE
TCCONFIG TC
XINIXH   H
TCPICK   TCP
EJMSG    C
EJ1      C
EJ2      C
EJ4      C
EJ5      C
EJ6      C
EJ7      C
EJ1_2    C
EJ8      C
```

Los que interesan son los 4 primeros, los archivos "ej*.c" son los fuentes de programas ejemplo en lenguaje C incluyendo los del capítulo 2; pueden borrarse, más se recomienda que primero los compilen y ejecuten.

- 3)_Crear el sub-subdirectorio LIB (dentro de xinix) y colocarse en él:

```
C>md lib
C>cd lib
```

- 4)_Copiar todos los archivos en el subdirectorio "lib" del disco "XINIX B>", al sub-subdirectorio "xinixa\lib" en el disco duro:

```
C>copy a:\lib\*.*
```

Deben copiarse:

```
COH      OBJ
CH       LIB
MATHH    LIB
EMU      LIB
```

- 5)_Reconfigurar al compilador TC:

```
C>cd..
C>tc
```

El comando "cd.." nos devuelve al subdirectorio padre, a "xinixa" en nuestro caso. En este se encuentra TCCONFIG.TC que copiamos anteriormente, así que TC tomará las opciones de éste.

Ya en TC oprimir la tecla F10 para obtener el menú principal. Con las teclas flecha elegir el rubro Options, oprimir la tecla ENTER, aparecerán nuevas opciones. Con las teclas flecha elegir la opción Directories, aparecerán la siguientes opciones:

```
Include directories: A:\INCLUDE
Library directories: B:\LIB
Output directory:
Turbo C directory: A:
Pick file name TCPICK.TCP
```

Actualizar "Include directories", con la identificación del directorio donde se encuentran los archivos ".h" del compilador TC (es muy probable que sea "c:\tc\include", verificarlo). En "Turbo C directory", hay que poner la identificación del directorio donde se encuentra el compilador (? "c:\tc" ?). Por último, en "Library directories", hay que poner "c:\xinixa\lib".

Recuérdese que en las especificaciones del manejador de disco (c:), letra c puede ser d o e, según el disco duro que el usuario haya elegido para instalar a XINIX. Por otra parte, las modificaciones a los directorios se realizan de la siguiente manera: posicionarse en el directorio a cambiar con las teclas flecha, oprimir la tecla ENTER, en este momento TC solicita nuevo directorio, teclear lo que corresponda y terminar con ENTER. Cabe mencionar que esto no afecta en nada el comportamiento del compilador TC del usuario, pues las modificaciones se registran, como se verá en el punto que sigue, sólo en el archivo TCCONFIG.TC.

6) Salvar la reconfiguración

Modificados los directorios de trabajo, lo que sigue es salvar o registrar los cambios: oprimir la tecla ESC, nos encontraremos en el menú anterior (donde elegimos la opción "Directories"), elegir ahora la opción "Store options". Al oprimir ENTER, TC desplegará:

```
Config File
C:\XINIXA\TCCONFIG.TC
```

Volver a oprimir la tecla ENTER, ahora TC desplegará:

```
Verify
Overwrite TCCONFIG.TC ? (Y/N)
```

...oprimir la tecla Y, TC salvará los cambios. Si se desea desarrollar una aplicación, oprimir la tecla F10 para volver al menú principal; en caso contrario, oprimir simultáneamente las teclas "Alt" y "x" para salir de turboc. Eso fué todo para instalar la versión disco duro de XINIX.

Preparación y ejecución de una aplicación

Supóngase ahora que el usuario va a modificar y correr el programa "ejl.c" en la versión disco duro. Ya en DOS y en el disco duro donde se encuentra XINIX, colocarse en el directorio "XINIXA", correr TC para realizar la edición y la compilación, salir de TC, copiar "ejl.exe" a un diskette en el disco A, correr XINIX, y ... dentro de "XINIX" ejecutarlo conforme a la versión austera:

```
C>cd\xinixa
C>tc xul
C>copy ejl.exe a:
C>xinix
:
:
CONSOLE>> execd ejl.exe
:
:
```

Acerca de la instalación

El usuario se preguntará porqué se copian sólo unos y no todos los archivos de la versión austera. Bueno, no se necesita el compilador TC ni los archivos "*.h" (de TC), a menos que el usuario no los tenga instalados en su disco duro, si este es el caso, copiarlos de la siguiente manera: colocar el disco "XINIX A>" en el manejador de disco A>, colocarse en el directorio donde se instaló XINIX, copiar TC:

```
C>cd\xinixa
C>copy a:tc.exe
```

..., crear el sub-subdirectorio "include", colocarse en él y copiar todos los archivos en el subdirectorio "include" del disco en A>:

```
C>md include
C>cd include
C>copy a:\include\*.*
```

..., volver al directorio padre, XINIXA, y reconfigurar de nuevo a TC, "sólo" en la parte correspondiente al directorio de trabajo "Include directories", ponerle "c:\xinixa\include", ver el punto 5 anterior.

El resto de los archivos en el diskette "XINIX A>", los relacionados con DOS, no se necesitan, pues se encuentran en el disco duro donde se instaló XINIX.

Por otra parte, respecto al diskette "XINIX B>", es obvio que para aplicaciones XINIX se necesitan los archivos xinix.exe y xinix.h. De este diskette no se necesitan los archivos TCPICK.TCP ni los de ejemplos, los "e*.c", más copiando todo se facilita la instalación. La configuración de TC, que se encuentra en el archivo TCCONFIG.TC, se ha actualizado en la parte relacionada a los directorios de trabajo; más la parte que corresponde al compilador y encadenador ha quedado intacta, y es ésta la que nos interesa conservar para desarrollar aplicaciones XINIX; de hecho es necesaria esta configuración. El

usuario puede familiarizarse con élla corriendo TC estando en el directorio "xinxix". Ya en TC, del menú principal elegir el rubro "Options", y de éste elegir las opciones "Compiler" y "Linker".

Por último, los archivos: c0h.obj y los "*.lib", que se copiaron al subdirectorio "xinxix\lib", son necesarios por las siguientes razones: el primero porque no es el archivo c0h.obj de TC, es uno especialmente desarrollado para correr aplicaciones XINIX (en el capítulo 4 se explica su función); mientras que los archivos "*.lib" se copian para facilitar la instalación y configuración de TC. Ver el manual "Turbo C User's Guide", la parte que corresponde a "instalación".

3.2 COMANDOS Y SERVICIOS XINIX SEGUN SU APLICACION. QUE SE OFRECE.

Esta parte agrupa, según su aplicación, a todos los comandos y servicios XINIX con los que el usuario cuenta para el desarrollo de sus aplicaciones. La manera de usar a cada uno de ellos se da en el apéndice C. Primero se listan los comandos del intérprete, luego los servicios internos. Cabe mencionar que el usuario puede utilizar toda la librería de TurboC, excepto aquellas funciones que tratan con dispositivos: terminal o archivos. Así, el usuario puede utilizar libremente, por ejemplo, todas las rutinas str.. para manejo de strings. En general, el usuario no deberá usar las rutinas cuya utilización requiere que se incluya el archivo "stdio.h" de TurboC.

COMANDOS DEL INTÉRPRETE

Relacionados con procesos

chprio:

Cambia la prioridad de un proceso.

create:

Crea un proceso a partir de un archivo ejecutable en un diskette XINIX. El nuevo proceso queda suspendido.

created:

Crea un proceso a partir de un archivo ejecutable en un diskette DOS. El nuevo proceso queda suspendido.

exec:

Crea un proceso a partir de un archivo ejecutable en un diskette XINIX. El nuevo proceso queda en ejecución (READY).

execd:

Crea un proceso a partir de un archivo ejecutable en un diskette DOS. El nuevo proceso queda en ejecución (READY).

kill:

Termina a un proceso.

ps:

"Process Status", de cada uno de los procesos en el sistema, despliega sus atributos: identificación, nombre, prioridad, estado, etcétera. Otros comandos requieren que se ejecute primero a "ps", para así obtener la identificación del proceso que desean afectar.

resume:

Reanuda un proceso en estado suspendido.

send:

Envia un mensaje (un valor entero) a un proceso.

signal:

Incrementa en uno el contador de un semáforo. Si en éste hay un proceso bloqueado lo libera.

signaln:

Incrementa la cuenta asociada a un semáforo en un valor "n" (que recibe como argumento). Desbloquea hasta "n" procesos bloqueados si los hay.

sleep:

Retarda la ejecución del proceso intérprete de comandos (shell) de una terminal, por un tiempo especificado en segundos.

suspend:

Suspende la ejecución de un proceso. Éste debe encontrarse en estado CURRENT o READY.

Relacionados con archivos

cat:

Concatena (copia) uno o más archivos en uno sólo. También sirve para desplegar el contenido de un archivo por la terminal.

cdxf:

Copia un archivo en un diskette DOS a un archivo en un diskette XINIX.

close:

Cierra un archivo. Util cuando un proceso de usuario no lo cierra antes de terminar. Ver el comando "devs".

cp:

Copia un archivo a otro. No existe sobreposición: no debe existir el archivo destino.

devs:

Lista los dispositivos que maneja el sistema. Incluye el número de archivos que pueden estar abiertos simultáneamente y su identificación, "Num", para utilizarla con el comando close.

dir:

Lista el directorio de un diskette DOS.

fmt:

Formatea un diskette y le graba el sistema de archivos XINIX, lo hace un diskette XINIX.

ls:

Lista el directorio de un diskette XINIX.

mv:

Cambia el nombre de un archivo en un diskette XINIX (MoVe).

rm:

Borra un archivo de un diskette XINIX (ReMove).

vrf:

Verifica las pista de un diskette DOS o XINIX.

Relacionados con la terminal

echo:

Hace echo al texto que recibe, útil en versiones posteriores de XINIX, cuando se implemente la ejecución de archivos de comandos.

who:

Lista la identificación de los usuarios en cada terminal del sistema.

Relacionados con la memoria

bpool:

Lista los atributos de los pools (despensas) de memoria: identificación, tamaño de cada bloque, semáforo que lo controla y la cuenta de éste. Recuérdese que un pool es un área de memoria controlada; el control es por semáforos); y la cuenta del semáforo de un pool nos representa cuantos

bloques están disponibles.

mem:

Muestra la distribución de la memoria: cantidad (de bytes) libre al empezar; cantidad libre actual; cantidad ocupada para pila de procesos; cantidad de la memoria "heap"; y la lista de los bloques libres: su dirección y su longitud en bytes. La memoria "heap" es la que ocupan las instrucciones de los procesos, los pools y las petición dinámicas de memoria por parte de procesos de usuario.

Relacionados con los comandos

help y "?":

Listan los comandos disponibles.

Para terminar una sesión

exit y logout:

Terminan el procedimiento shell(): ya no es posible ejecutar comandos, hay que dar de nuevo una identificación.

SERVICIOS INTERNOS

Los servicios internos se utilizan desde un programa, como si se llamara a un procedimiento, los hay

Relacionados con el manejo procesos

create:

Crea un proceso a partir de un "procedimiento" en memoria. Lo deja en estado suspendido. Ver los ejemplos del capítulo 2.

resume:

Reanuda un proceso suspendido, lo pone en estado READY: compite por el procesador.

suspend:

Suspende a un proceso. Éste debe estar en estado READY o CURRENT.

chprio:

Cambia la prioridad a un proceso.

getpid:

Con este servicio, un proceso puede obtener su propia identificación: un valor entero que representa la entrada en la tabla de procesos asignada a un proceso.

kill:

Termina a un proceso. Si un proceso desea terminarse a si mismo, lo haria de la siguiente manera: `kill(getpid());`

get_priority:

Obtiene la prioridad de un proceso.

get_status:

Obtiene el estado (CURRENT, SUSPEND,...) de un proceso.

Relacionados con la sincronización de procesos

screate:

Crea un semáforo. Recibe el valor inicial del contador asociado a éste. Devuelve la identificación del semáforo.

wait:

Decrementa la cuenta asociada a un semáforo. Si la cuenta resulta negativa, el proceso que llamó a wait es bloqueado.

signal:

Revisa la cuenta asociada a un semáforo. Si esta es negativa, desbloquea al primer proceso bloqueado, entonces incrementa la cuenta asociada al semáforo.

signaln:

Incrementa la cuenta asociada a un semáforo en un valor "n" (que recibe como parámetro). Si existen procesos bloqueados desbloquea hasta "n".

reset:

Reestablece la cuenta de un semáforo a un nuevo valor (que recibe como parámetro). Si existen procesos bloqueados, todos son liberados.

scount:

Devuelve el valor de la cuenta asociada a un semáforo.

sdelete:

Borra un semáforo. Si existen, libera todos los procesos bloqueados.

Relacionados con la comunicación entre procesos

MENSAJES CORTOS: un valor entero de 16 bits.

receive:

Espera por un mensaje. El proceso queda bloqueado en caso de no existir uno.

send:

Envía un mensaje a un proceso. Si ya existe un mensaje, el envío actual NO se completa: no hay sobreposición. Si el proceso que ha de recibir el mensaje ya espera por él, entonces se desbloquea. El proceso que envía no se bloquea.

sendf:

Envía un mensaje a un proceso forzosamente. Si ya existe un mensaje, éste se pierde: se deposita (sobrepone) el actual. Si el proceso que ha de recibir el mensaje ya espera por él, entonces se desbloquea. El proceso que envía no se bloquea.

recvclr:

Si existe un mensaje lo obtiene y regresa inmediatamente, sino, también.

MENSAJES LARGOS: un valor de 32 bits (que puede ser la dirección de un objeto), a través de puertos.

pcreate:

Crea un puerto para enviar y recibir mensajes. Recibe el número de mensajes que pueden existir simultáneamente: la capacidad del puerto. Devuelve la identificación de éste.

psend:

Envía un mensaje a través de un puerto. Si no hay espacio para depositarlo, el proceso que llama a `psend()` queda bloqueado.

preceive:

Obtiene un mensaje de un puerto. Si no existe uno, el proceso queda bloqueado.

pcount:

Obtiene el número de mensajes que hay en un puerto.

pdelete:

Borra un puerto. Si hay procesos bloqueados los libera.

preset:

Borra los mensajes de un puerto. Si hay procesos bloqueados los libera.

Relacionados con el manejo de reloj**sleep:**

El proceso que le llama se "suspende temporalmente" (se duerme) por un tiempo especificado en segundos.

sleep10:

El proceso que le llama se "suspende temporalmente" (se duerme) por un tiempo especificado en décimas de segundo.

stopclk:

Detiene el manejo del reloj del sistema en tiempo real: los ticks son contabilizados pero no procesados; así que no hay cambio de contexto ni los procesos durmientes despiertan a tiempo. Según las veces que un proceso ejecute a `stopclk()`, las mismas veces debe ejecutar a `startclk()` para reanudar, efectivamente, el reloj del sistema. En otras palabras, los llamados a `stopclk()` son acumulativos.

strtclock:

Pone en marcha al reloj del sistema: se procesan los ticks que se habían contabilizado pero no procesado.

Relacionados con el manejo dinámico de memoria

MANEJO SIMPLE

getmem:

Obtiene memoria del sistema. Si hay disponible, devuelve su dirección, si no el valor "SYSERR", definido en el archivo xinxh.h que el usuario debe incluir en sus aplicaciones.

freemem:

Devuelve al sistema la memoria solicitada con getmem().

MANEJO CONTROLADO

mkpool:

Crea un pool (despensa) de bloques de memoria de tamaño fijo (mínimo 2 bytes, máximo 4096). El usuario especifica cuántos bloques manejará el pool.

getbuf:

Obtiene un bloque de un pool. Si no lo hay, el proceso es bloqueado hasta que haya uno disponible.

freebuf:

Devuelve un bloque al pool que pertenece.

Relacionados con Entrada/Salida de alto nivel

open:

Abre (establece conexión con) un dispositivo (o archivo).

close:

Cierra un dispositivo.

getc:

Obtiene un carácter de un dispositivo.

read:

Lee 1 o más caracteres de un dispositivo.

putc:

Escribe un carácter en un dispositivo.

write:

Escribe uno o más caracteres en un dispositivo.

seek:

Posiciona a un dispositivo. La siguiente operación de lectura o escritura será a partir de la posición obtenida. Si se trata de un archivo, el usuario especifica el número de carácter en el que se desea posicionar. Si se trata del disco, se refiere al número de sector físico. No tiene sentido para un dispositivo tipo "terminal".

control:

Controla a un dispositivo: programa o modifica sus características.

gets:**fgets:****scanf:****fscanf:****puts:****fputs:****printf:****fprintf:**

Todos estos servicios se utilizan tal y como se indica en "TurboC Reference Manual", con la única diferencia de que esta versión de "scanf", "fscanf", "printf" y "fprintf", no trabajan con valores reales: no es posible leer o escribir valores reales (con punto decimal o notación científica).

kprintf:

Todas las funciones anteriores utilizan al manejador de disco (el buffer que éste maneja) para leer o escribir caracteres en un dispositivo; mientras que "kprintf" escribe directamente en la memoria video RAM (ver apéndice 1) de la terminal de memoria mapeada: sin pasar por el manejador de terminal, por esta razón es más rápido. Se recomienda utilizarlo "sólo" con propósitos de revisión mientras se desarrolla una aplicación. Se utiliza exactamente igual que printf, con la misma restricción.

Relacionados con el manejador de interrupciones 8259 y terminales RS232 (puerto de comunicación serie).

enabdisp:

Habilita a un dispositivo a interrumpir al chip manejador de interrupciones 8259. Si el usuario conecta un dispositivo adicional y desea controlarlo por interrupciones, puede habilitarlo con este procedimiento. Ver [INTEL81] pag. 7-128.

disadisp:

Deshabilita a un dispositivo a interrumpir al chip manejador de interrupciones 8259. Ver procedimiento anterior.

init8259:

Establece el número de vector de interrupción inicial asociado al primer dispositivo que puede interrumpir. Son 8 los dispositivos que un 8259 puede controlar, a cada uno se asigna un vector de interrupción a partir del vector inicial. El 8259 pasa al 8086 el número de vector en el que se encuentra la dirección de la rutina que atenderá la interrupción provocada por un dispositivo. El usuario no debe utilizar este procedimiento, a menos que entienda y modifique la iniciación de los vectores de interrupción que XINIX lleva a cabo al empezar. Ver [INTEL81] pag. 7-130 y [TRIBM83] pag. A-9.

pol8259:

Pone al manejador de interrupciones en modo "polling": el 8259 no interrumpe al 8086, éste lee el estado del primero para ver si existe una interrupción por atender. Ver [INTEL81] pag. 7-131. El usuario debe tomar en cuenta las condiciones de uso del procedimiento init8259().

npol8259:

Quita al manejador de interrupciones del modo "polling": va a interrumpir al 8086 cada que un dispositivo interrumpa. Considerar lo escrito para el procedimiento pol8259().

intpend:

Se utiliza para saber si hay una interrupción por atender cuando el 8259 se encuentra en modo "polling" Devuelve la identificación (0 a 7) del dispositivo que interrumpió, el valor -1 si no hay interrupción por atender.

initport:

Programa un puerto de comunicación serie: velocidad, número

de bits a transmitir, número de bits de parada y tipo de paridad. El puerto queda sin poder interrumpir al 8259, ver los procedimientos que siguen para habilitarle a interrumpir por diferentes condiciones.

enablet:

Habilita la interrupción, por haber transmitido un carácter, de un puerto de comunicación serie.

enabler:

Habilita la interrupción, por haber recibido un carácter, de un puerto de comunicación serie.

enablem:

Habilita la interrupción, por variación en el estado del modem, de un puerto de comunicación serie. Ver [TRIBM83] pag. 1-240 a 1-245.

enables:

Habilita la interrupción, por variación en el estado de la línea de transmisión, de un puerto de comunicación serie. Ver Ver [TRIBM83] pag. 1-239 a 1-242.

disablet:

disabler:

disablem:

disables:

Deshabilitan la interrupción de un puerto de comunicación serie por haber transmitido o recibido un carácter, por variación en el estado del modem o de la línea de transmisión respectivamente.

readyt:

Al manejar un puerto de comunicación serie en forma "polling" (sin interrumpir al 8259), esta función nos devuelve TRUE en caso de que ya se pueda enviar otro carácter: la transmisión anterior se ha completado; FALSE en caso contrario.

readyr:

En modo "polling", esta función devuelve TRUE si se ha recibido un carácter a través de un puerto de comunicación serie; FALSE en caso contrario. Ver readyt().

erroronr:

Si al recibir un carácter por un puerto de comunicación serie hubo error, esta función nos devuelve TRUE; FALSE en caso contrario. El error puede ser por sobreposición de un

carácter a otro (llegó un carácter antes de leer el anterior), de paridad, de trama etcétera. Ver [TRIBM83] pag. 1-239.

breakt:

Deshabilita a un puerto de comunicación serie a transmitir.

nobreakt:

Restablece la transmisión de un puerto de comunicación serie.

Para leer o escribir datos en el espacio de puertos de la IBM-PC

readport:

Lee un carácter de un puerto. El equivalente del lenguaje TurboC es `inportb()`.

writport:

Escribe un carácter en un puerto. El equivalente del lenguaje TurboC es `outportb()`.

CAPITULO 4

XINIX - XINU

Se mencionó en el capítulo 2 que XINIX es una versión PC de XINU original. El cambio de nombre obedece a que XINIX utiliza parte del código fuente, del manejador de diskette, del sistema operativo MINIX. Así, el nombre XINIX se formó de las letras mayúsculas de XINU y minIX.

Este capítulo describe la organización lógica y física de XINIX, así como los cambios y adiciones sustanciales realizadas a XINU. El apéndice B describe en detalle que hacer para instalar, modificar y compilar a XINIX. En tal caso sería conveniente tener a la mano [COMER84], [COMER87] y [TANEN87].

Cabe mencionar que XINIX se constituyó a partir de [COMER84], sin considerar los capítulos 14 y 16: "El manejador de comunicaciones de datos"; de [COMER87] se tomó el intérprete de comandos; mientras que de [TANEN87] se tomó, del manejador de diskette, la parte que programa los chips Floppy Disk Controller (FDC) y Direct Memory Access (DMA), que realizan la transferencia de información entre memoria principal y diskette; y del manejador de terminal, la parte que convierte un código de tecla a su respectivo carácter ASCII.

Por "tomar" no se debería pensar en "añadir y compilar de nuevo"; en realidad es una tarea ardua y minuciosa,... y muy agradable al terminar.

4.1 ORGANIZACION LOGICA Y FISICA DE XINIX. INICIACION.

Organización Lógica

La organización lógica se refiere a las capas (o componentes) de XINIX por el tipo de servicios que ofrecen: manejo de memoria, manejo de procesos, sistema de archivos etc., la figura 4.1 muestra la organización lógica de XINIX. La organización física se refiere a la distribución de las capas lógicas en los archivos fuente de XINIX.

Cada capa constituye u ofrece servicios para las capas superiores incluyendo los programas del usuario. Por ejemplo, el intérprete de comandos utiliza el servicio create(), en la capa "manejador de procesos", para ejecutar (crear un proceso a partir de) un programa en

disco del usuario. El programa a su vez puede, utilizando también a create(), crear procesos a partir de procedimientos encadenados con él. Ver los ejemplos ej1.c y ej2.c del capítulo 2.

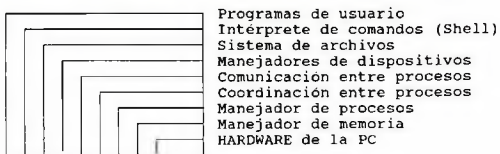


Figura 4.1. Organización lógica de XINIX. Capas.

Cada capa se forma de procedimientos y datos globales. Un servicio es un procedimiento en una capa, más no todos los procedimientos en esta son servicios; por ejemplo, create() llama a newpid(), otro procedimiento del manejador de procesos, para obtener la identificación que asignará al proceso a crear. Newpid() busca una entrada libre en el arreglo global proctab[] (donde se encuentran los atributos de cada proceso existente en el sistema), y si la encuentra devuelve su número. La capa "coordinación de procesos" utiliza al arreglo semaph[] para implementar los semáforos. Para crear un semáforo, screate() llama a newsem(), el equivalente de newpi(), para buscar una entrada libre en semaph[]. Las procedimientos en otras capas también utilizan variables y arreglos globales para manejar dispositivos, bloques de memoria, archivos, almacenar los nombres de los comandos, etcétera.

En general, los procedimientos en cada capa realizan una función definida sobre un tipo de objeto en particular: un proceso, un semáforo, un archivo, un diskette, una terminal, etcétera. Por su parte, los objetos, sus atributos, son los campos que forman una entrada de un arreglo global en una capa; mientras que los valores en estos campos constituyen el "estado" actual del objeto: un proceso suspendido, un archivo abierto, un mensaje enviado,...; son estados que se detectan, por los servicios de una capa, al preguntar por el valor de un campo-atributo.

Organización Física

La organización física de XINIX está en el archivo_proyecto xinx.prj, que indica al compilador TurboC (TC) los archivos a compilar y encadenar para obtener xinx.exe: el programa ejecutable XINIX que ha de correr en DOS. A continuación se lista xinx.prj.

```
xinx      (xconf.c, xconf.h, xkernel.h, xq.h, xproc.h, xsem.h,
          xmem.h, tty.h, xio.h, xmark.h, xbufpool.h, xdisk.h,
          xfdcdma.h, xfile.h, xiblock.h, xdir.h)
x2lqmani (xconf.h, xkernel.h, xq.h)
x2prmana (xconf.h, xkernel.h, xq.h, xproc.h, xsem.h, xmem.h)
x2prcoor (xconf.h, xkernel.h, xq.h, xproc.h, xsem.h)
```

```

x2ckmana (xconf.h, xkernel.h, xq.h, xproc.h, xsleep.h)
x2memana (xconf.h, xkernel.h, xmem.h)
x2bufpol (xconf.h, xkernel.h, xmem.h, xmark.h, xbufpool.h)
x2ports (xconf.h, xkernel.h, xmem.h, xmark.h, xports.h)
x2hlio (xconf.h, xkernel.h, xio.h)
x2ttyio (xconf.h, xkernel.h, tty.h, tty_ibm.h, xio.h)
x2hlmemn (xconf.h, xkernel.h, xmark.h)
x2dskdrv (xconf.h, xkernel.h, xproc.h, tty.h, xdisk.h,
          xfdcdma.h, xfile.h, xiblock.h, xdir.h, xio.h)
x2filsys (xconf.h, xkernel.h, xproc.h, xdisk.h, xfile.h,
          xiblock.h, xdir.h, xio.h, xfdcdma.h)
x2shell (xconf.h, xkernel.h, xproc.h, xshell.h, xcmd.h,
          tty.h, xbufpool.h, xmem.h, xdisk.h)
x2dosfls (xconf.h, xkernel.h, xproc.h, xmem.h, tty.h,
          xmark.h, xbufpool.h, xdisk.h, xfile.h, xiblock.h,
          xdir.h, xio.h, xfdcdma.h, xdosfls.h)
x2reloc (xconf.h, xkernel.h, xproc.h, xmem.h, xio.h, xreloc.h
          xmark.h, xbufpool.h, xdisk.h, xfile.h, xiblock.h,
          xdir.h, xfdcdma.h, xdosfls.h)
x2iofmt (xconf.h)
x2ctxsw.obj
x2clkint.obj
x2libasm.obj
x2emuxin.obj

```

Figura 4.2. Organización física de XINIX. Archivo xinix.prj.

Los nombres más a la izquierda designan módulos (archivos con instrucciones) XINIX. Los declarados entre paréntesis son archivos que contienen definiciones de macros, constantes simbólicas y estructuras. Los módulos con extensión ".obj", son los archivos objeto de fuentes en lenguaje ensamblador; los que no tienen extensión (como xinix, x2lqmani, ..., x2iofmt) son fuentes en lenguaje C, tienen extensión ".c". Los archivos designados entre paréntesis que siguen al nombre de un módulo, son incluidos por éste al momento de la compilación con la declaración "#include". De esta manera, las definiciones en un archivo pueden utilizarse en varios módulos sin tener que repetirlas, y lo más práctico y útil, es que en caso de alterar una definición, digamos una estructura, la modificación se realiza en un sólo archivo.

La mayor parte de cada capa lógica se encuentra en un archivo fuente en lenguaje C. La otra parte puede ser común a otras capas y por eso estar en un módulo aparte, o bien estar programada en lenguaje ensamblador. Por ejemplo, las capas que manejan y coordinan procesos, constituidas por los archivos x2prmana.c y x2prcoor.c respectivamente, crean y manejan listas de procesos con los procedimientos en el archivo x2lqmani.c (list queue manipulation). Mientras que x2ctxsw.obj, producto de ensamblar al archivo x2ctxsw.asm, contiene la rutina ctxsw() que forma parte del manejador de procesos.

La iniciación de XINIX

En el archivo `xinix.c` se encuentra el procedimiento principal, `main()`, que se encarga de la iniciación total del sistema XINIX, veamos en que consiste ésta:

- 1)_Inicializa las variables y arreglos globales del sistema.
- 2)_Obtiene de DOS la memoria que el manejador de memoria XINIX ha de administrar.
- 3)_Obtiene y guarda el contenido del vector de interrupción asociado al reloj, al teclado, al diskette, el 21H de DOS y el utilizado para la interfaz XINIX que da servicio a los programas de usuario. `Main()` restaura estos vectores antes de volver a DOS, poco después que el usuario oprima las teclas `Ctrl-Alt-Del` simultáneamente.
- 4)_Carga estos vectores con la dirección de los manejadores e interfaces XINIX.
- 5)_Inicializa los dispositivos: terminales (a cada una le crea su proceso intérprete de comandos), disco (crea el proceso que se encarga de manejarlo), y archivos.
- 6)_Entonces `main()` se convierte en el "proceso nulo" de XINIX:

```
xinix_on = TRUE;          /* variable global */
while( xinix_on == TRUE )
```

Cuando el manejador de terminal XINIX detecta la secuencia de caracteres `Ctrl-Alt-Del`, simultáneamente, pone el valor `FALSE` en la variable `xinix_on`; `main()` sale de la instrucción `while`, y

- 7)_Finalmente restaura los vectores de interrupción que modificó, devuelve la memoria que pidió y regresa a DOS.

4.2 CAMBIOS Y ADICIONES A XINU

Esta sección está organizada de la siguiente manera: primero se exponen los cambios y adiciones generales que afectan a todas las capas lógicas, para continuar, con cada capa, a explicar sus cambios y adiciones particulares.

4.2.1 Cambios y adiciones generales

Éstos se deben a las diferentes arquitecturas de las computadoras PC, donde corre XINIX, y LSI/11-02, donde corre XINU originari.

Apuntadores.

Las direcciones en la LSI son de 16 bits, con los que se puede direccionar un total de 64Kb ($= 2^{16}$). En la PC las direcciones son de 20 bits, con los que se puede direccionar un Mega byte ($2^{20} = 1048576$). El tamaño de los registros en ambas computadoras es de 16 bits, sin embargo, en la PC se utilizan (suman) 2 registros para formar una dirección, uno de los cuales se recorre 4 bits a la izquierda al momento de formarla: pasarla a la unidad de memoria. Ver la figura 4.1.

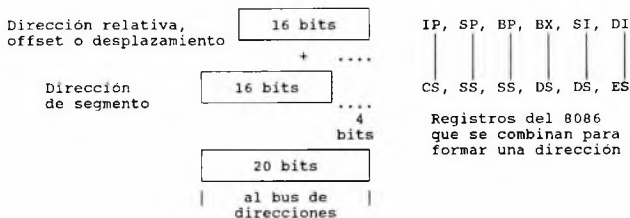


Figura 4.3. Formación de una dirección "física" en la PC.

Cuando en la PC, en el 8086, se utiliza una variable apuntador, su valor se carga en uno de los pares de registros de la figura 4.3, antes de poder direccionar datos o instrucciones. Un apuntador en la PC, entonces, es de 32 bits: 16 para el registro de segmento y 16 para el desplazamiento. Obsérvese que es posible definir un "desplazamiento" hasta de 64KB a partir de una dirección de segmento, pues los registros que se utilizan para tal fin son de 16 bits. De hecho, es posible obtener programas, para la PC, que manejan apuntadores de 16 bits, pero esto implica que no pueden direccionar todo el Megabyte disponible. Estos programas se compilan con el modelo SMALL de TC, y tal es el caso de la versión XINU_PC que se menciona en el capítulo 1. En la compilación de XINIX se utiliza el modelo HUGE, también de TC, que maneja apuntadores de 32 bits, y por lo tanto XINIX puede direccionar todo el megabyte disponible. El beneficio directo para el usuario es que puede correr, concurrentemente, más aplicaciones más grandes.

Sin embargo, manejar apuntadores de 32 bits presenta un problema: diferentes combinaciones de valores offset:segmento pueden constituir una misma dirección física. Véase la figura 4.4. Los valores que se muestran son valores hexadecimales: cada uno de 4 bits.

1) Off	+ 0 0 1 2	2) Off	+ 0 0 0 2
Seg	<u>1 0 0 0</u>	Seg	<u>1 0 0 1</u>
Dir. F.	1 0 0 1 2	Dir. F.	1 0 0 1 2
3) Off	+ 0 0 2 2	4) Off	+ f f e 2
Seg	<u>0 f f f</u>	Seg	<u>0 0 0 3</u>
Dir. F.	1 0 0 1 2	Dir. F.	1 0 0 1 2

Figura 4.4. Combinación de segmentos y offsets.

La situación anterior implica que no se pueden hacer comparaciones directas entre dos variables apuntadores de 32 bits, y tampoco, aplicar a éstos directamente la aritmética de apuntadores del lenguaje C. En ambos casos, antes es necesario obtener, de los 32 bits, la dirección física de 20 (bits) que constituyen. Ésta, como se puede apreciar en la figura 4.4, es la misma. Entonces, a partir de la dirección física, se forma de nuevo el apuntador de 32 bits; el nuevo offset queda sólo con el valor de los 4 primeros bits de derecha a izquierda, y el nuevo segmento con el resto (16 bits). Ver la figura 4.5.

offset inicial	0 0 2 2	
	+	
segmento inicial	0 f f f	
Dirección Física de 20 bits	<u>1 0 0 1 2</u>	
Nuevo offset	0 0 0 2	} Nuevo apuntador de 32 bits.
Nuevo segmento	1 0 0 1	

Figura 4.5. Normalización de un apuntador en XINIX.

A este proceso se le llama "normalización" de un apuntador, y como se puede observar, 2 apuntadores con la misma dirección, normalizados son iguales.

En XINIX, para manejar correctamente los apuntadores de 32 bits, se desarrollaron las rutinas para incrementar un apuntador, decrementarlo, normalizarlo, desnormalizarlo y comparar dos apuntadores. Estas rutinas se encuentran en el archivo x2memana.c, más se utilizan a través de las macros incpn(), decpn(), etc., que se encuentran en el archivo xmem.h.

Por último, todas las funciones para apuntadores mencionadas son obvias, excepto la "desnormalización". Un apuntador no normalizado está desnormalizado. ¿ No ? Sin embargo, en XINIX un apuntador desnormalizado tiene una característica especial: de la dirección física de 20 bits, los primeros 16 de derecha a izquierda forman el nuevo offset; mientras que los últimos 4 bits forman, en el valor de segmento, los 4 bits más significativos. Ver la figura 4.6.

Un apuntador desnormalizado es lo que devuelve el servicio getstk(), cuando, al crear un nuevo proceso, el servicio create() le llama para obtener memoria_pila. La dirección devuelta por getstk() es la dirección final de un bloque de memoria, y además, como ya se

mencionó, desnormalizada.

offset inicial	0 0 2 2		
	+		
segmento inicial	0 f f f		
Dirección Física de 20 bits		1 0 0 1 2	
Nuevo offset	0 0 1 2	}	Nuevo apuntador de 32 bits.
Nuevo segmento	1 0 0 0		

Figura 4.6. Desnormalización de un apuntador en XINIX.

¿ Porqué la dirección final ? Bueno, ésto obedece a que en el 8086, como en la mayoría de los procesadores recientes, la pila (espacio de memoria para guardar direcciones de retorno y variable temporales) crece (se guardan datos) hacia las direcciones bajas de la memoria, mientras que disminuye (se vacía) hacia las direcciones altas de la memoria.

¿ Y porqué la desnormalización ? Bueno, se ha mencionado que todas las variables apuntador, en el 8086, deben cargarse en un par de registros de la figura 4.3. La pila en particular se direcciona con los registros SS:SP, así que la dirección devuelta por getstk(), eventualmente, se ha de cargar en ellos. ¿ Qué pasaría si getstk() devuelve un apuntador normalizado ? La figura 4.7 muestra el comportamiento de una pila iniciada con tal apuntador. Se suponen los siguientes valores (hexadecimales) iniciales en el segmento y el offset respectivamente: 3333:0004.

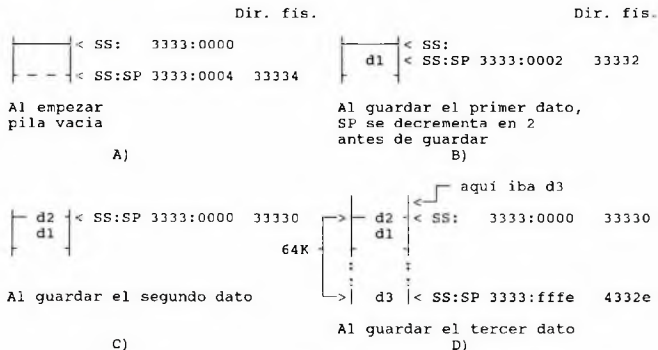


Figura 4.7. Comportamiento de pila iniciada con apuntador normalizado.

Cuando SP es igual a cero, ver C), la pila se ha llenado. Puesto que al guardar un dato el 8086 decrementa sólo al SP (sin acarreo para el SS), y además como un entero sin signo, el siguiente decremento a SP lo convierte en fffe (ffff + 2 = 0000); que junto con SS constituye una dirección 64K más alta (en valor, más abajo en la figura) para guardar el siguiente dato, en lugar de guardarlo encima del último (d2). Si la nueva dirección de SS:SP ya es utilizada, las inconsistencias no se harán esperar. Por otra parte, es obvio que una pila iniciada con un apuntador normalizado no requiere guardar más de 8 datos tipo entero (2 bytes), para empezar a direccionar inconsistentemente: $2 * 8 = 16$, valor máximo del offset a cargar en SP.

El mismo apuntador de nuestro ejemplo: 3333:0004, desnormalizado, evita este problema. Se muestra el valor de SS y SP al guardar datos según la figura 4.7.

	SS	SP	Dir. fis.
A) Al empezar	3000	3334	33334
B) Al guardar el primer dato	3000	3332	33332
C) Al guardar el segundo dato	3000	3330	33330
D) Al guardar el tercer dato	3000	3328	33328 (y no 4332e)

Para finalizar este apartado, no se debe olvidar que todos los apuntadores en XINIX ocupan 32 bits: el tamaño de 2 enteros o de un entero long; mientras que en XINU original ocupan 16 bits: 1 entero.

Acceso a dispositivos.

Los dispositivos como terminales, discos, impresoras, etc., que junto con un procesador (8086 o LSI-11) constituyen una computadora, tienen un conjunto de registros que forman el medio para programarles. Al escribir en estos registros una secuencia de bits particular, uno provoca la operación de lectura o escritura de caracteres en los dispositivos, leer el estado actual de éstos, programar sus características, etc.

En la computadora LSI-11, al conectar un dispositivo, sus registros quedan mapeados con el espacio de la unidad de memoria: los datos en un registro de un dispositivo se leen tal y como se lee de memoria, o se escribe en un registro de un dispositivo como se escribe a memoria. A continuación un fragmento, en lenguaje ensamblador, de la rutina ctxsw() de XINU original, que corre en el procesador LSI-11:

```

_ctxsw:      / el slash inicia un comentario
             mov r0, *2(sp) / salva el registro r0 de la UCP en la pila
             mov 2(sp), r0 / ...
             :

```

Lo importante de este código, es notar que la instrucción "mov" realiza las transferencias de información entre la memoria y los registros del procesador. Para escribir un valor en un registro de un

dispositivo, también se utilizaría la instrucción "mov":

```

;
mov DIR_REG, r0 / r0 = a la dirección del registro
mov VALOR, (r0) / mueve VALOR a donde apunta r0, al registro
;
```

El mapeo de memoria y registros de dispositivos puede entenderse también de la siguiente manera: consiste en utilizar, indistintamente, la misma instrucción "mov" para transferir información entre el procesador (sus registros), y la unidad de memoria o registros de dispositivos. Lo anterior implica que es posible, en la LSI, acceder directamente los registros de dispositivos desde un lenguaje de alto nivel, como C, que soporta manejo de apuntadores; pues como se mencionó anteriormente, el valor de un apuntador se carga, antes de realizar el direccionamiento "a memoria", en un registro del procesador. Obsérvense las últimas 2 líneas de código. A continuación la versión en lenguaje C de estas.

```

;
int *port; /* apuntador a un registro de un dispositivo */
;

port = DIR_REG;
*port = VALOR;
;
```

Así, XINU original puede programar dispositivos o transferir información con estos, a través de procedimientos en lenguaje C utilizando apuntadores. A continuación se muestra parte de las rutinas ttyoin() y ttyiin() de XINU original, que escriben y leen caracteres por un puerto de comunicación serie controlado por el manejador de terminal.

```

INTPROC ttyoin(iptr) /* transmite caracteres por puerto serie */
struct tty *iptr; /* apunta a los atributos de la terminal */
{
    struct csr *cptr; /* dirn. ler registro en el pto serie */
    ;
    cptr = iptr->ioadr;
    ;
    ; /* escribe sig. car. en buffer salida al */
    ; /* registro transmisor en el puerto serie */
    cptr->ctbuf = iptr->obuf[iptr->otail++];
    ;

INTPROC ttyiin(iptr) /* recibe caracteres por puerto serie */
struct tty *iptr;
{
    struct csr *cptr;
    int ch;
    ;
    ch = cptr->crbuf; /* lee carácter del reg. receptor de */
    ; /* caracteres en el puerto serie */
    ; /* procede a meterlo en el buffer de lectura */
}
```

En la computadora PC, que utiliza al procesador 8086, no existe el mapeo entre memoria y registros de dispositivos que existe en la LSI_11/02. Existe el espacio de memoria que contiene datos e instrucciones, y un "espacio de puertos" que se mapea con los registros de dispositivos. Para transferir información entre los registros del 8086 y la unidad de memoria, se utiliza también la instrucción "mov"; más para transferir información con registros de dispositivos (o puertos), se utilizan las instrucciones "in" y "out" para leer o escribir en ellos respectivamente. Por otra parte, los apuntadores siempre direccionan a la memoria, así que no es posible acceder, desde el lenguaje C, los registros de dispositivos. Por esta razón, en XINIX se utilizan rutinas en lenguaje ensamblador, llamadas desde C, que reciben la identificación del puerto donde hay que leer o escribir. Estas rutinas, en su interior, utilizan las instrucciones de máquina "in" o "out" para acceder los registros de dispositivos. A continuación parte de las rutinas, del manejador de terminal XINIX, que leen y escriben caracteres por un puerto de comunicación serie.

```

INTPROC ttyoin(iptr) /* transmite caracteres por puerto serie */
struct tty *iptr; /* apunta a los atributos de la terminal */
{
    int port; /* id reg. de trans. puerto serie */
    :
    port = iptr->ioaddr;
    : /* escribe sig. car. en buffer salida al */
    /* registro transmisor en el puerto serie */
    writport(port, iptr->obuf[iptr->otail++]);
    :

INTPROC ttyiin(iptr) /* recibe caracteres por puerto serie */
struct tty *iptr;
{
    int port; /* id reg. recepción puerto serie */
    int ch;
    :
    ch = readport(port); /* lee carácter del reg. receptor de */
    /* caracteres en el puerto serie */
    : /* procede a meterlo en el buffer de lectura */
}

```

Cabe mencionar que los otros manejadores XINIX de dispositivos, también utilizan writport() y readport() para transferir información a los dispositivos que controlan.

4.2.2 Cambios y adiciones particulares a cada capa.

MANEJADOR DE MEMORIA. Archivo x2memana.c

XINIX "mantiene bloques de memoria libre encadenados en una lista, con la variable global memlist apuntando al primer bloque libre" [COMER84]. Es posible obtener o liberar un bloque de memoria libre durante la ejecución de un programa. Los procedimientos servicio que lo realizan son getheapmem(), getstkmem() y freememory(), que se utilizan, a diferencia de XINU, a través de las macros getmem(), getstk() y freemem() respectivamente. Estas macros, junto con freestk(), se definen en el archivo xmem.h de la siguiente manera:

```
#define getstk(len) getstkmem((long)(len))
#define getmem(len) getheapmem((long)(len))
#define freemem(p,len) freememory( (p), (long)(len))
#define freestk(p,len) freememory(decptrn(p, (long)roundew(len) \
- sizeof(int)), (long)(len))
```

Los nombres de estas macros corresponden a los nombres de los procedimientos servicio en XINU, en los que, el parámetro "len" es de tipo entero (16 bits), y especifica, al solicitar o devolver un bloque de memoria, su tamaño; no mayor de 64Kb. En XINIX es posible pedir, o devolver, un bloque de memoria mayor de 64Kb; potencialmente se puede pedir poco menos que la capacidad de memoria de la PC (1 Mega). Por tal razón, el parámetro "len" en XINIX es de tipo "long" (32 bits), para poder especificar un tamaño de bloque mayor de 64Kb. Recuerdese que el valor máximo representable en 16 bits es 64K.

Todos los procedimientos servicio en el manejador de memoria XINIX esperan y utilizan al parámetro "len" como tipo "long". Veamos las implicaciones. Al llamar un procedimiento en el 8086, en lenguaje C, los parámetros y la dirección de retorno se colocan en la pila. A partir de la dirección de retorno, hacia las direcciones altas, se encuentran los parámetros, y hacia las bajas las variables locales o dinámicas (pues se crean al hacer el llamado a un procedimiento). Ver la figura 4.8.

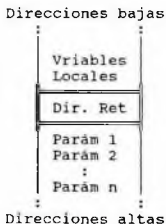


Figura 4.8. Paso de parámetros en la pila en el 8086, lenguaje C.

Antes de llamar a un procedimiento, el proceso que llama le coloca los parámetros en la pila. Una vez que el procedimiento empieza su ejecución, el primer argumento se encuentra siempre justo después de la dirección de retorno. El segundo parámetro se encuentra 2 bytes adelante si el primero es de tipo entero o char (en la pila del 8086 sólo se pueden guardar o recuperar 16 bits); 4 bytes si el primero es de tipo long o un apuntador de 32 bits. El tercer parámetro se encuentra tantos bytes adelante como bytes ocupen los dos anteriores, etcétera. El compilador, al procesar el llamado a un procedimiento, genera el código para poner en la pila tantos bytes por parámetro según su tipo; y al compilar al procedimiento, el código para, según el tipo y orden de los parámetros a recibir, determinar su dirección en la pila y su tamaño. Aclarando. Considerese el siguiente llamado al procedimiento proc().

```

:
long par1;
int par2, par3;
:
proc(par1, par2);      /* llamado */
:

proc(p1, p2)          /* definición de proc */
long p1;
int p2;
(
:
)

```

En el llamado, se guardan en la pila 2 bytes correspondientes al parámetro par2, 4 para par1. En proc(), el primer parámetro (p1), por su tipo es de 4 bytes; así que p2 (de 2 bytes) se encuentra, en la pila, 4 bytes después de la dirección de retorno. Todo esta bien, lo que se envía se ha puesto donde y como se espera recibir.

¿ Qué pasa si en el llamado se envían 2 parámetros tipo entero ?

En el llamado no hay problema, en la pila se colocan 2 bytes por cada uno, 4 en total. Sin embargo, en proc() se toman éstos 4 bytes para conformar a p1, mientras que para p2 se toman los siguientes 2 que nunca se colocaron en la pila. No se ha enviado lo que se espera recibir. Los inconsistencias no se harán esperar.

Los procedimientos servicio del manejador de memoria XINIX, como ya se mencionó, esperan recibir el tamaño de un bloque como un valor "long" de 4 bytes, pues es la única manera de poder manejar bloques mayores de 64Kb. Sin embargo, el usuario podría solicitar memoria de la siguiente manera:

```

getheapmem(30);      /* se solicitan 30 bytes */
:
getheapmem(sizeof(struct x));
:

```

En ambos casos se envía un valor entero de 2 bytes, sin embargo, getheapmem() tomará 4, a partir de la dirección de retorno para

conformar su argumento "len". Pueden ocurrir 2 cosas. Si los dos bytes adicionales que se toman (considerados como un entero que constituye la parte alta de "len"), tienen el bit más significativo prendido, se estará enviando un valor negativo que maneja como error. Más si no es el caso anterior, entonces se estará pidiendo una cantidad mucho mayor de memoria, tal vez más de un mega, en cuyo caso la petición también se rechazaría; a menos que estos 2 bytes adicionales estuviesen limpios (en cero), cosa que no es muy probable.

El problema puede resolverse indicando al usuario que en sus peticiones, el tamaño del bloque lo envíe como tipo de dato "long", o bien, decirle que los servicios se llaman getmem(), freemem(),...; las macros de XINIX, que se encargan de "preparar" y realizar el llamado a los servicios verdaderos, y así evitarle problemas al usuario, y además, asegurar el funcionamiento correcto del sistema. Se eligió la última opción.

MANEJADOR DE PROCESOS. Archivo x2prmana.c

Procesos internos y externos

Para correr una aplicación en XINU, los procedimientos_servicio que éste ofrece deben encadenarse al programa del usuario, pues no tiene un intérprete de comandos para solicitarle la ejecución de un programa. Este tipo de aplicaciones son estáticas, en el sentido de que no es posible crear más procesos de los que se especifican en el programa del usuario. En XINIX se pueden ejecutar programas "en disco" con el intérprete de comandos, así que el número de procesos en un momento dado no es predecible. La ejecución de un programa consiste en cargarlo a memoria, relocalizarlo y convertirlo en proceso. Por otra parte, XINIX también crea procesos a partir de procedimientos en alguna de las capas lógicas, algunos son el proceso "operador de diskette" y los procesos "shell" de cada terminal. Estos procesos, las instrucciones que ejecutan y los datos que utilizan, son parte del programa ejecutable XINIX. A éstos, XINIX les llama procesos "internos"; y "externos" a los generados a partir de programas en disco. Esto no existe en XINU, o podría pensarse que todos los procesos son internos.

Para ambos tipos de procesos, al crearles, se pide un bloque de memoria para el espacio pila; y sólo para los externos se pide un bloque para contener sus instrucciones y datos globales que se encuentran en disco. Cuando un proceso termina, se devuelve al sistema su bloque pila, y si es externo su bloque de datos e instrucciones. Para esto, en la tabla de procesos (proctab[]), en la entrada correspondiente a un proceso, se guarda su tipo en el campo ptype, mientras que la dirección y tamaño del bloque de datos e instrucciones en los campos pcode y pcodelen respectivamente. La dirección y tamaño del espacio pila se encuentra en los campos base y pstklen.

Cuando el servicio create() crea un proceso, hereda a éste, del proceso que le crea, los atributos ptype, pcode, pcodelen y ppctr. Los 3 últimos sólo tienen sentido si el primero identifica a un proceso externo. El primer proceso en XINIX se deriva del procedimiento main()

en `xinix.c`, que se convierte a sí mismo en el proceso "nulo" de tipo interno. Así que los procesos que éste crea, el operador de disco y los intérpretes de comandos de cada terminal, son de tipo interno también.

Al solicitar la ejecución de un programa en disco, el intérprete de comandos que recibe la petición ejecuta a `loadreloc()`, en el archivo `x2reloc.c`. `Load_reloc()` busca el nombre del programa en el directorio, y si lo encuentra, pide con `getmem()` la memoria suficiente para contener las instrucciones y datos del programa en disco; si hay memoria lo carga y relocaliza, entonces llama a `create()` para convertirlo en proceso. `Create()` le crea como un proceso interno, pues el intérprete de la terminal, quien llama a `create()`, es de este tipo. `Load_reloc()` recibe de `create()` la identificación del nuevo proceso, y con ésta obtiene la dirección de su entrada en `proctab[]`; entonces actualiza sus campos `pptype`, `pcode`, `pcodelen` y `ppctr`. Respectivamente les mueve los valores `PEXTRN`, la dirección que devolvió `getmem()`, la cantidad en bytes de memoria pedida, y la identificación de una entrada en la tabla `fpctrtab[]`. Ya es un proceso externo.

`Fpctrtab[]` es un arreglo de contadores tipo entero, cada uno de los cuales lleva la cuenta de cuantos procesos se han creado a partir del código de un programa (en disco). Solo al crear un proceso a partir de un programa se obtiene uno de éstos contadores inicializado a uno. Uno es el proceso que se crea a partir de un programa, el cual inicia su ejecución en el procedimiento `main()` del mismo. Eventualmente, este proceso puede crear otros procesos a partir de procedimientos en el mismo programa, veáanse los ejemplos del capítulo 2. Los cuales heredarán los atributos `pptype`, ..., `ppctr`; así que también serán externos. A continuación el código de `create()` que realiza el paso de la herencia.

```
SYSCALL create(procaddr,ssize,priority,name,nargs,args)
```

```

*
*
{
    int pid;                /* almacena ident del nuevo proces */
    struct pentry *pptr;    /* apuntadores a entradas en proctab[] */
    struct pentry *apptr;   /* del proceso creado y del que crea */
    .                      /* respectivamente */
    .
    pptr->pstate = PRSUSP; /* estado inicial */
    pptr->pptype = PINNER; /* internal process by default */

    if ((apptr = &proctab[getpid()])->pptype == PEXTRN) {
        pptr->pcode = apptr->pcode;
        pptr->pcodelen = apptr->pcodelen;
        pptr->pptype = PEXTRN;
        pptr->ppctr = apptr->ppctr; /* id cont. procesos */
        fpctrtab[pptr->ppctr]++; /* un proc. mas */
    }
    .
    .
}

```

Obsérvese que el contador, cuya identificación se encuentra en el campo `ppctr`, es incrementado al crear un proceso "a partir del código en un mismo archivo". Y decrementado cuando uno de ellos termina:

```
SYSCALL kill(pid)          /* kill termina un proceso */
    int pid;                /* proceso a terminar */
{
    struct pentry *pptr;    /* apunta a la entrada */
                          /* en proctab[] de pid */
    .
    freestk(pptr->ibase, pptr->pstklen); /* libera el espacio pila *
/* libera memoria de código si procede */
    if (pptr->ptype == PEXTRN && --fpctrtab[pptr->ppctr] == 0)
        freemem(pptr->pcode, pptr->pcodelen);
    .
}
```

De los ejemplos en el capítulo 2, tómesese en cuenta que el proceso `main` termina una vez que crea a los otros procesos. Su entrada en `proctab[]` es liberada, y posiblemente se asigne al crear otro proceso, es por eso que hereda su tipo, número de contador en `fpctrtab[]`, y dirección y tamaño del bloque de memoria donde se cargó el programa del cual deriva. El bloque se libera al terminar el último de los procesos creados con código del programa: cuando el contador asociado en `fpctrtab[]` sea igual a cero. No se puede liberar antes, digamos cuando termina `main`, pues si éste creó a otros procesos que continúan corriendo y se ejecuta otro programa, es posible que se cargue donde están los otros. De hecho sucedía durante el desarrollo de XINIX.

Servicios adicionales

A esta capa se incluyeron también los servicios `get_priority()` y `get_status()` para obtener, respectivamente, la prioridad y el estado actual de un proceso.

COORDINACION DE PROCESOS. Archivo `x2prcoor.c`

Para esta capa sólo incluyeron los servicios `scount()` y `signaln()`, no incluidos en la referencia 1, pero sí utilizados en la implementación del manejador de terminal. Son muy sencillos, `scount()` devuelve la cuenta asociada a un semáforo, mientras que `signaln(s,c)` incrementa la cuenta del semáforo `s` en `c`, y libera "hasta" `c` procesos bloqueados si existen. `Signaln()` equivale a llamar `c` veces a `signal(s)`, pero optimizado: si existen varios procesos bloqueados en el semáforo `s`, y se llama a `signal(s)` varias veces, en cada llamado ocurrirá cambio de contexto si procede según prioridades de los procesos; mientras que `signaln(s,c)` pasa "todos" los procesos que han de desbloquearse al

estado "ready", y entonces provoca cambio de contexto. Esto es útil, pues realizar cambio de contexto es una operación que consume tiempo considerable del procesador.

COMUNICACION ENTRE PROCESOS. Archivo x2prcoor.c

Para esta capa sólo se incluyó el servicio sendf(), una variante del servicio send(). Send(pid, msg) envía el mensaje msg, del tamaño de un entero, al proceso cuya identificación es pid, si y sólo si, pid es un proceso activo y no se le ha enviado un mensaje por recibir. Sendf() realiza lo mismo, excepto que no toma en cuenta si existe otro mensaje, fuerza el envío.

MANEJADORES DE DISPOSITIVOS

Manejador de Reloj. Archivos x2ckmana.c y x2clkint.asm.

Los servicios sleep() y sleep10() suspenden, temporalmente, la ejecución del proceso que les llame. El proceso se pone en la lista de "durmientes" de acuerdo al tiempo que ha de dormir. Esta lista se maneja de manera incremental: si existe un solo proceso X que ha de dormir 5 segundos, y un proceso Y requiere dormir 7; entonces Y es enlistado después de X, pero con un tiempo por dormir igual a 2 (7-5). Más si Y requiere dormir 3 segundos con el mismo estado inicial de la lista, entonces es puesto antes de X con un tiempo por dormir igual a 3, y se actualiza el tiempo por dormir de X, se hace igual a 2 (5-3).

En general, el manejo de la lista XINIX de procesos "durmientes" tiene las siguientes implicaciones: al inicio de la lista se encuentra siempre el primer proceso que debe despertar; el tiempo total por dormir de un proceso, es la suma de los tiempos individuales por dormir de los procesos que le preceden en la lista, incluyendo el propio; al disminuir el tiempo por dormir del primer proceso en la lista, se disminuye el de todos los procesos durmientes. Esto es realmente eficiente, pues si el estado "durmiendo" (SLEEP) no se manejase en una lista incremental; entonces, cada que interrumpe el reloj del sistema (en XINIX es cada décima de segundo), habría que disminuir el tiempo por dormir individual de todos los procesos durmiendo. Si éstos son muchos, se utilizaría al procesador de manera muy poco eficiente.

No existen cambios para las rutinas en lenguaje C que manejan la suspensión temporal de un proceso: dormirlo y despertarlo; más si se cambió la rutina clkint() que atiende las interrupciones de reloj. En XINU, clkint() se limita a disminuir, cada décima de segundo, el tiempo por dormir del primer proceso a despertar (si existe), y si es igual a cero, despertarle. Clkint() también disminuye el tiempo que debe continuar en ejecución el proceso actual, su quantum, compuesto por 10 décimas de segundo, y si alcanza el valor 0, clkint() provoca cambio de

contexto. Este es todo el manejo de reloj que `clkint()` realiza en XINU.

En XINIX, además de lo anterior, `clkint()` contabiliza los ticks del reloj para manejar la hora del sistema, aun cuando no existen los servicios para obtenerla y actualizarla: `gettime()` y `settime()` serian los nombres de estos servicios. También maneja un temporizador asociado con los motores del manejador de diskette (MD); cuando el temporizador llega a cero, significa que no se han realizado operaciones con el MD desde hace 2.5 segundos; así que procede a apagar los motores para evitar desgaste prematuro a los diskettes. Con el mismo temporizador se realiza otra función importante cuando éste alcanza el valor cero: se envía un mensaje (`NOT_READY`) al proceso "disk_operator" encargado de realizar las operaciones con el MD. Normalmente, `disk_operator` espera el mensaje `GO_AHEAD` de parte de la rutina que atiende las interrupciones del MD, que ocurren cuando éste termina de realizar la operación que se le encomendó. Las operaciones del MD, leer o escribir sectores, pistas, etc., toman milisegundos para realizarse; más si se abre la puerta del MD las operaciones no terminan, y las interrupciones por completitud no ocurren. Así que se permiten 2.5 segundos para que el usuario cierre la puerta del diskette, en caso contrario, al consumirse éste tiempo y confirmar que `disk_operator` esta esperando - con `get_status()`-, `clkint()` le envía el mensaje `NOT_READY`. `Disk_operator` procede entonces a desplegar el mensaje:

```
Not ready error reading drive A
Abort, Retry, Ignore?
```

Por último, `clkint()` en XINIX permite a un usuario la ejecución periódica de un procedimiento, tal como en DOS. Para esto hay que cargar la dirección del procedimiento en el vector 1C hexadecimal. Si el procedimiento está escrito en lenguaje C su tipo debe ser "void interrupt"; si en lenguaje ensamblador, debe volver con la instrucción `IRET`.

Manejador de Terminal. Archivo `x2ttyio.c`

XINU maneja sólo terminales tipo RS232, XINIX maneja la terminal de memoria mapeada (TMM) de la PC, y tantas terminales RS232 como puertos de comunicación serie estén conectados. El código fuente sólo cambia al momento de leer o escribir un carácter. Si se trata de las terminales RS232, se utilizan las rutinas `readport()` y `writport()` respectivamente; más si se trata de la TMM se utilizan, respectivamente, los procedimientos `keyboard()` y `putchar_bios()`.

Los primeros leen o escriben un "carácter ASCII" en el espacio de puertos mapeado con los registros de dispositivos. `Keyboard()`, constituido de procedimientos del SO MINIX, también devuelve un carácter ASCII, derivado de la conversión que realiza al "scan code" obtenido al oprimir una tecla. Es decir, las terminales RS232, al oprimirles una tecla, convierten el número de ésta, considerando si "otras teclas" (`Shift`, `Ctrl`, `CapsLock`, `NumLock`) están activas, al código ASCII del carácter dibujado en la tecla; y es éste el código que envían a través del puerto serie. Por eso se recibe con `readport()`, sin ningún tratamiento adicional. Por otra parte, el chip que controla el teclado en la PC, e interrumpe al 8086 cuando se oprime una tecla, ofrece a la rutina que atiende la interrupción sólo el número de tecla (`scan code`). Así que tal rutina, `keyboard()` en XINIX, se encarga de la

conversión número_de_tecla -> código ASCII del carácter, que es lo que se devuelve al manejador de terminal, quién a su vez lo coloca en su buffer de lectura. Los programas que leen caracteres los toman de este buffer. La conversión sin embargo no es brumosa: con el scan code como índice, y considerando si "otras teclas" están activas, se obtiene el código ASCII del carácter de una de las siguientes tablas, parte del manejador de terminal XINIX (MTX):

```
/* Scan codes to ASCII for unshifted keys */
```

```
char unsh[] = {
0,033,'1','2','3','4','5','6','7','8','9','0','-','=','\b',
'\t','q','w','e','r','t','y','u','i','o','p','[',']',015,
0202,'a','s','d','f','g','h','j','k','l',';',047,0140,
0200,0134,'z','x','c','v','b','n','m','.',',','/',0201,'*',
0203,'-',0204,0241,0242,0243,0244,0245,0246,0247,0250,0251,0252,0205,
0210,0267,0270,0271,0211,0264,0265,0266,0214,0261,0262,
0263,'0',0177};
```

```
/* Scan codes to ASCII for shifted keys */
```

```
char sh[] = {
0,033,'!', '@', '#', '$', '%', '&', '*', '(', ')', '-', '+', '\b',
'\t','Q','W','E','R','T','Y','U','I','O','P','[',']',015,
0202,'A','S','D','F','G','H','J','K','L',';',042,'_',
0200,'|','Z','X','C','V','B','N','M','<','>','?',0201,'*',
0203,'-',0204,0221,0222,0223,0224,0225,0226,0227,0230,0231,0232,0204,
0213,'7','8','9',0211,'4','5','6',0214,'1','2',
'3','0',0177};
```

Esto en cuanto a la lectura de caracteres de la TMM. En cuanto a la escritura, putchr_bios() llama al servicio para desplegar caracteres en el BIOS de la PC (interrupción 10h servicio 14).

Manejador de Disco. Archivo x2dskdrv.c

En XINU, las operaciones para leer o escribir en un sector de diskette las realizan los procesos que las requieren. El manejador de disco (MD) de la LSI-11/02, al igual que el Floppy Disk Controller (FDC) en la PC, solo pueden realizar una transferencia a la vez. Puesto que la velocidad funcional de los dispositivos es mucho más lenta que la del procesador, puede ocurrir que varios procesos requieran operaciones con diskette cuando aún no termina la actual. Por esta razón se maneja una lista de operaciones para al MD.

En XINU, si la lista está vacía y un proceso requiere una operación, él mismo programa al MD para realizar la transferencia; más si la lista no está vacía, al final de ésta pone su petición, y procede a esperar por la completitud de la transferencia o continúa su ejecución. Cuando la transferencia actual termina, el MD provoca una interrupción; se ejecuta la rutina que la atiende (seguramente durante la ejecución de otro proceso), ésta checa por errores para determinar que se devolverá al proceso que solicitó la operación, entonces revisa

si hay peticiones en la lista, en cuyo caso inicia la siguiente petición (programa al MD para realizar la transferencia); finalmente vuelve al proceso que sufrió la interrupción. Nótese que la programación del MD la realizan varios procesos.

En XINIX también existe una lista de peticiones, pero sólo un proceso que las lleva a cabo: el que programa al FDC. Este proceso se llama `disk_operator`; y cuando no hay peticiones su estado es `SUSPEND`; `RECEIVE` cuando espera que el FDC termine la operación actual (el mensaje se lo ha de enviar la rutina que atiende las interrupciones del FDC, cuando éste termine de realizar la operación actual); `SLEEP` cuando espera que un motor de diskette, apagado, alcance su velocidad de trabajo antes de realizar la transferencia; y `READY`, cuando prepara y realiza la programación del FDC. Al requerir una operación con diskette, los procesos preguntan si la lista de peticiones está vacía; si es así enlistan la suya y reanudan la ejecución de `disk_operator` con el servicio `resume()`; sino, ponen su petición al final de la lista. `Disk_operator` lleva a cabo las peticiones una a la vez conforme llegaron a la lista, y cuando no hay más él mismo se suspende.

En XINU y XINIX las operaciones básicas son lectura y escritura de sectores físicos de 512 bytes c/u. En XINIX, además, existen las operaciones para leer y escribir una pista completa (9 sectores), formatear un diskette con el sistema de archivos XINU, verificar pistas o sectores, y, para cambiar, dinámicamente con una función de control, el tamaño del bloque lógico de transferencia en un diskette. Inicialmente éste es de 512 bytes al igual que el sector físico; sin embargo es posible manejar bloques lógicos de 1024, 2048 y 4096 bytes. Esto ha sido muy útil en el desarrollo de otra tesis construida sobre XINIX: "Un sistema de archivos tipo UNIX para XINIX".

SISTEMA DE ARCHIVOS. Archivo `x2filsys.c`

Son pocos los cambios a esta capa. Se implementaron los procedimientos para borrar y renombrar un archivo, y para listar el directorio de un sistema de archivos. Los tres se ejecutan por el intérprete de comandos en una terminal para llevar a cabo los comandos `"rm"`, `"mv"` y `"ls"` respectivamente. Otros comandos que tienen que ver con archivos (como `"cat"`, `"cp"`,...) utilizan los procedimientos básicos de XINU y XINIX: `open()`, `read()`, `seek()`, `write()`, etcétera.

Mientras que en XINU un archivo se utiliza en forma exclusiva por el proceso que le abre, en XINIX varios procesos pueden utilizar un archivo al mismo tiempo.

INTERPRETE DE COMANDOS (SHELL). Archivo `x2shell.c`

El intérprete de comandos XINIX deriva de la referencia 2, capítulos 16, 17 y 18. En su versión original, el intérprete trata con un "servidor de archivos" en lugar de un sistema de archivos

tradicional; así, los comandos que trabajan con archivos, como "cat", provocaron cambios sustanciales.

De todos los comandos XINIX (ver el apéndice C), los siguientes son adicionales: cdxf, chprio, created, dir, exec, execd, fmt, resume, send, signal, signaln, suspend y vrf. De éstos, los no subrayados son sencillos, t n solo preparan el llamado a los servicios en las capas precedentes. Sin embargo, para cdxf y dir se program  un m dulo para manejar (buscar, leer y escribir) archivos en diskettes con formato doble-lado doble-densidad de DOS, ver el archivo x2dosfls.c. Created y execd tambi n utiliza  ste m dulo para transferir un programa ejecutable desde DOS; pero adem s utilizan un m dulo relocalizador (archivo x2reloc.c) para ejecutarles una vez en memoria. Exec utiliza este  ltimo a partir de programas ejecutables en sistema de archivos XINU, los cuales se transfieren con el comando cdxf (Copy Dos to Xinu File). Los procedimientos que llevan a cabo los comandos fmt y vrf son parte del manejador de diskette, se encuentran en el archivo x2dskdrv.c.

Por  ltimo, todo el manejo de archivos, sean DOS o XINU, se hace a trav s del manejador de diskette XINIX: disk_operator.

OTROS.

Lectura y Escritura Formateada. Archivos x2iofmt.c

Leer y escribir informaci n a un dispositivo es tarea del Sistema Operativo. Es posible que antes de escribir, o despu s de leer, se d  un tratamiento a la informaci n tal como el formateo, m s la transferencia de  sta entre el dispositivo y la memoria la realiza el SO. Por ejemplo, si se llama a printf(), de la librer a de TurboC, para desplegar el valor de una variable entera de acuerdo con el formato "%d";  ste convierte el valor binario, en los 2 bytes de la variable, a una cadena hasta de 5 car cteres si el valor en la variable es positivo, y hasta de 6 car cteres si es negativo, pues se incluye el car cter "-" antes de los caracteres n mericos. Entonces printf() pide al SO que transmita la cadena a la terminal. La peticion que printf() hace es  nica para DOS en el caso de TC. Por esta raz n, XINIX provee el servicio printf() que permite formatear y escribir informaci n a la terminal. Fprintf() permite escribir a cualquier dispositivo. Ambos formatean y solicitan a XINIX, con putc(dev), que transmita la cadena obtenida a partir del formateo. Printf() y fprintf() acompa an a XINU, para XINIX se programaron kprintf(), scanf(), fscanf(), puts(), fputs(), gets(), y rutinas de soporte.

Interfaz XINIX. Archivo x2emuxin.asm.

En XINIX, a diferencia de XINU, es posible correr programas en forma interactiva con el int rprete de comandos. En XINU las

aplicaciones se encadenan con todos los procedimientos servicio del primero; en XINIX no, pues encadenarle a toda a aplicación se reflejaría en varias copias de XINIX en memoria al ejecutarles.

¿ Cómo entonces se obtiene acceso a los servicios ?

Bueno, todo programa en lenguaje TC se encadena, según el modelo de memoria con que se compile, al módulo COx.obj, y a la librería Cx.lib; donde la letra "x" identifica el modelo utilizado: h: huge, s:small, etcétera. COx.obj es realmente el punto inicial de ejecución de todo programa de usuario; entre otras, realiza las funciones de pasar los argumentos iniciales al programa, si los hay, al procedimiento main(); llama a main(); y cuando éste termina, COx.h pide a DOS termine el programa: todo inicia y termina en COx.obj. Todos los archivos COx.obj derivan de ensamblar al archivo CO.asm que TC provee. TC también provee el procedimiento de comandos "build-c0.bat" para ensamblarle según el modelo deseado.

En la compilación de XINIX se utiliza el modelo HUGE, por lo tanto se le encadena con el archivo c0h.obj de TC. Sin embargo, las aplicaciones XINIX se encadenan con un c0h.obj diferente, que deriva de la modificación y ensamblado de CO.asm de TC. La modificación consiste en incluir los procedimientos create(), resume(),..., writport(). Son 75 procedimientos: uno por cada servicio XINIX. A continuación varios de ellos:

```
public _create
_create proc far
    mov ax, CREATE
    int XINU_INTERRUPT
    ret
_create endp
```

```
public _resume
_resume proc far
    mov ax, RESUME
    int XINU_INTERRUPT
    ret
_resume endp
```

```
public _writport
_writport proc far
    mov ax, WRITPORT
    int XINU_INTERRUPT
    ret
_writport endp
```

El carácter "_" es necesario al nombrar rutinas en ensamblador a ser llamadas desde lenguaje C, pues éste, por convención, lo hace para todos los nombres de procedimientos o variables. Los nombres CREATE, RESUME,..., WRITPORT, son definiciones simbólicas, en el archivo xinu_emu.d, de valores enteros que van desde 1 hasta 75, e identifican cada uno de los servicios XINIX.

Al encadenar las aplicaciones de usuario con estos procedimientos, todas sus referencias a servicios XINIX se resuelven, y se obtiene el

programa ejecutable. Obsérvese que todas las rutinas generan la misma interrupción (XINU_INTERRUPT).

En la iniciación de XINIX se carga la dirección de xinixcall() en el vector XINU_INTERRUPT.

Un programa de usuario que solicita un servicio XINIX, ejecuta la rutina correspondiente en c0h.obj. Ésta mueve la identificación del servicio al registro AX, y genera la interrupción XINU_INTERRUPT. Se ejecuta entonces xinixcall(), que en base a la identificación de servicio que recibe en AX, ejecuta el procedimiento XINIX que lo constituye.

APÉNDICE A

ACERCA DE TERMINALES

Básicamente existen 2 tipos de terminales: las que se conectan a la computadora a través de un puerto serie, y transmiten o reciben un bit de información a la vez; y las de memoria mapeada, las cuales forman parte de la computadora.

TERMINALES RS232

Las terminales que se conectan a través de un puerto serie reciben el nombre de terminales RS232, y utilizan un conector de 25 contactos para comunicarse con la computadora. Por un contacto se recibe información, por otro se transmite, y un tercero es tierra, los 22 restantes sirven para manejar funciones de control, pero por lo regular no se utilizan. Es posible definir diferentes velocidades de transmisión, las más comunes son: 300, 1200, 2400, 4800, y 9600 bits por segundo (bps). Ya que las computadoras y las terminales se comunican por una línea serie, un bit a la vez, pero trabajan con caracteres (8 bits), se han desarrollado chips para realizar las conversiones carácter->serie y serie->carácter. Estos chips se llaman UART's (Universal Asynchronous Receiver Transmitters), y se proveen a la computadora dentro de una tarjeta RS232, insertada en el bus del sistema tal como ilustra la figura A.1. En la PC, los UART's quedan al alcance de UCP a través del espacio de puertos. En los puertos se leen caracteres recibidos o el status del UART; o se escriben caracteres a transmitir o bien la programación del UART: velocidad, paridad a utilizar, bits de arranque y de parada, etcétera.

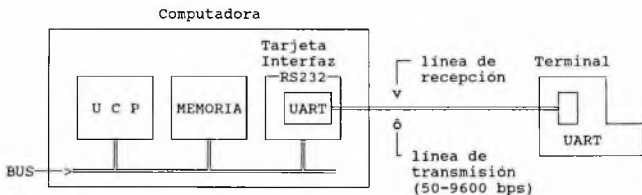


Figura A.1. Una terminal RS232 se comunica con la computadora por una línea de comunicación, un bit a la vez. La computadora y la terminal son independientes.

TERMINALES MAPEADAS EN MEMORIA

Las terminales de memoria mapeada forman parte de la computadora, y se implementan a través de una memoria especial llamada video RAM, que se integra al espacio de direcciones de la máquina a través del bus del sistema, y se utiliza de la misma manera que la unidad de memoria normal. La figura A.2 muestra la configuración de una terminal de memoria mapeada.

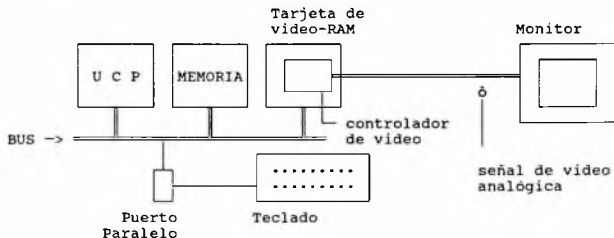


Figura A.2. Las terminales de memoria mapeada escriben directamente en la memoria video RAM.

También la tarjeta de memoria mapeada contiene un chip, el controlador de video, que se encarga de enviar, a partir de los caracteres en la video RAM, la señal analógica que controla al monitor. El monitor genera un rayo de electrones que corre horizontalmente a través de la pantalla pintando líneas sobre ésta. Generalmente, una pantalla tiene de 200 a 1200 líneas de arriba a abajo, con 200 a 1200 puntos por línea. Los puntos son llamados píxeles. La señal que envía el controlador determina si un pixel será luminoso u oscuro. Los monitores de color tienen 3 rayos para los colores rojo, verde y azul, los cuales se modulan en forma independiente.

Un carácter se puede definir en una caja de 9 píxeles de amplitud por 14 de altura, incluyendo el espacio entre caracteres y líneas de caracteres. Si se manejan 25 líneas de 80 caracteres cada una, se necesitan monitores con 350 líneas de 720 píxeles cada una. Los patrones de 9 por 14 bits que definen a los caracteres se almacenan en la memoria ROM (Read Only Memory) que utiliza el controlador de video.

En la IBM-PC, la video-RAM empieza en la dirección 0xB0000 para los monitores monocromáticos, y en la 0xB8000 para los de color. La figura A.3 ilustra la relación del contenido de la video RAM y la información desplegada en un monitor monocromático. Cada carácter en la pantalla ocupa 2 caracteres en la RAM. El de orden más bajo corresponde al código ASCII, y el de orden mayor es el byte de atributos, el cual especifica color, video inverso (carácter oscuro y caja luminosa), parpadeo, etcétera. La pantalla completa de 25 líneas de 80 caracteres cada una requiere $(25 \times 80 \times 2 = 4000 \text{ bytes})$ 4K de video RAM.

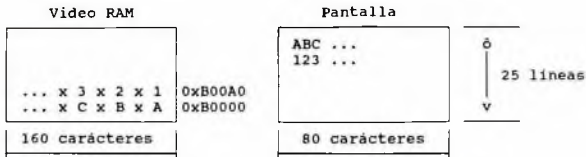


Figura A.3. Contenido de la video-RAM para una pantalla monocromática. Las x's son bytes de atributos.

El rayo de electrones recorre a la pantalla de 45 a 70 veces por segundo. Cuando un carácter se escribe en video RAM, aparece en la pantalla en el lapso en que ésta se despliega (1/50 seg. para monocromática, 1/60 seg para color). La CPU puede mover una pantalla predefinida de 4K a la video-RAM en 12 mseg. A 9600 bps, escribir 2000 caracteres a una terminal RS232 toma 2083 mseg; 174 veces más lenta.

En una terminal de memoria mapeada, el teclado está completamente desconectado de la pantalla. Se interactúa con él a través de un puerto paralelo o serie. Al oprimir una tecla se interrumpe a la UCP, y el manejador de la interrupción del teclado obtiene la identificación del carácter leyendo de un puerto. Con la identificación determina el código ASCII del carácter. En la IBM-PC el teclado interrumpe también al liberar una tecla, así que el manejador determina una "A" cuando se oprime una tecla "Shift", no se libera, y se oprime la tecla "A".

TERMINALES XINIX RS232, CARACTERISTICAS DE TRANSMISION.

- 19200 baudios,
- 8 bits por carácter,
- Paridad par y un
- 1 bit de parada.

Ver el servicio "initport()" en el apéndice C.

APÉNDICE B

XINIX PARA MODIFICACION. INSTALACION

El código fuente de XINIX se encuentra en dos diskettes de 360K en formato DOS. No deben confundirse con los 2 diskettes del capítulo 3, el medio ambiente de programación XINIX. El código fuente se compone de archivos en lenguaje C y en lenguaje ensamblador, archivos con definiciones y algunos otros. Los 2 diskettes están casi llenos. No hay lugar para los archivos objeto ni para el compilador TurboC (TC) ni para el sistema operativo DOS (recuérdese que XINIX se desarrolló en DOS). Considerando además, que trabajar con más de 2 diskettes es una tortura, se ha asumido que la persona interesada en modificar a XINIX cuenta con disco duro. En su defecto, corre por su cuenta implementar en diskettes la versión disco duro de "XINIX para modificación".

El disco 1 del código fuente, etiquetado "INSTALA Y FUENTES EN C", contiene, además de los fuentes en lenguaje C, los archivos:

```
README    COM
README
XRESP     BAT
XINSTALA  BAT
XINSTAL2  BAT
XHAZ-LIB  BAT
XASM      BAT
```

El archivo `readme` contiene todo lo que se necesita saber para instalar, modificar, ensamblar y compilar a XINIX, y algo más. Ejecutando el programa `readme.com`, asegurándose de estar en el manejador de disco donde se encuentra archivo `readme`, el usuario puede leerlo fácilmente: poner el disco "INSTALA Y FUENTES EN C" en el manejador A:, ponerse en el manejador A: y ejecutar `readme.com`:

```
?>a:
A>readme
```

`Xresp.bat` tiene los comandos DOS para construir (o respaldar), de los fuentes de XINIX en disco duro, los diskettes "INSTALA Y FUENTES EN C" y "OTROS FUENTES". Este último contiene los archivos en lenguaje ensamblador, todos los archivos de definiciones y los archivos para configurar al compilador TurboC.

`Xinstala.bat` tiene los comandos DOS para instalar, de los diskettes "INSTALA Y FUENTES EN C" y "OTROS FUENTES", la versión disco duro de

"XINIX para modificación"; realiza el proceso inverso que lleva a cabo xresp.bat. Xinstal2.bat es llamado por xinstala.bat: la instalación completa la realizan los dos. Además de realizar modificaciones a los fuentes de XINIX, el usuario puede interesarse en desarrollar aplicaciones que correrán en XINIX, así que ...

Xhaz-lib.bat crea la librería para encadenar aplicaciones XINIX.

Xasm.bat ensambla todos los módulos XINIX en lenguaje ensamblador. Los módulos en lenguaje C se compilan en forma automática a través del archivo_proyecto xinix.prj para TC, que también acompaña a los fuentes.

ANTES de realizar la intalación de XINIX, EL USUARIO DEBE LEER EL ARCHIVO README, y asegurarse de TENER YA INSTALADO el compilador TurboC versión 1.5 (o mayor) y un ensamblador. El ensamblador puede ser MASM de Microsoft, TASM de Borland o cualquier otro que el usuario desee; sin embargo, el compilador si debe ser TC. Además, TC y el ensamblador deben estar disponibles (poder ejecutarse desde cualquier directorio) a través de un "path" definido en el sistema.

APÉNDICE C

COMANDOS Y SERVICIOS XINIX EN ORDEN ALFABÉTICO. COMO USARLOS.

Los comandos son instrucciones que el usuario da a XINIX en forma interactiva, a través del intérprete de comandos, por medio del teclado de una terminal; mientras que los servicios internos se realizan al llamarles como procedimientos en un programa del usuario. Con el fin de evitar confusiones y facilitar al usuario las consultas al desarrollar sus programas, comandos y servicios se listan por separado en orden alfabético con un formato diferente en su explicación. A continuación el formato con el que se presenta la explicación de cada comando:

```
CONSOLE>> comando arg1 arg2... "comando"
```

...el comando se precede por el "prompt" del intérprete de comandos que corresponde a la terminal de memoria mapeada: "CONSOLE>> ", poniendo de nuevo, al final de la línea, el nombre del comando entre comillas ("comando"). La sintaxis del comando, lo que debe teclear el usuario, es lo que se encuentra entre el "prompt" y el nombre entre comillas, no más. Por otra parte, el formato para presentar de los servicios internos es:

servicio()	servicio
------------	----------

función general

sinopsis
tipo de parámetros que recibe

descripcion
qué hace en detalle, que regresa, tips, etcétera.

ver también
servicios relacionados.

SI SE OLVIDA LA SINTAXIS DE UN COMANDO, se recuerda al usuario que todos los comandos no informativos, excepto cat, despliegan la manera de usarlos en caso de no enviarles los parámetros necesarios, así:

CONSOLE>> cp

desplegará:

...usage: cp file1 file2

Ultimas indicaciones. A través de la explicación de comandos y servicios, la información contenida entre paréntesis cuadrados ([]) deberá interpretarse como opcional; mientras que los puntos suspensivos (...) indicarán que el elemento precedente puede repetirse una o más veces; además, las referencias a archivos deberán considerarse a archivos XINIX, a menos que se indique lo contrario.

COMANDOS DEL INTÉRPRETE

CONSOLE>> ? "?"

Despliega la lista de comandos disponibles al usuario. El comando help es equivalente.

CONSOLE>> bpool "bpool"

Despliega información acerca de los pools (despensas) de memoria controlada: número de bloques disponibles, tamaño de éstos y semáforo que los controla. No recibe parámetros.

CONSOLE>> cat file1 file2 ... > filer "cat"

Concatena (une o copia) uno o más archivos en uno sólo, el cual no debe existir. Si el usuario no especifica un archivo destino, con ">", entonces se asume la terminal, así: "cat file" despliega el contenido del archivo file en la pantalla de la terminal; mientras que, "cat >file" introduce datos desde el teclado de la terminal al archivo file. Para dejar de introducir al archivo, el usuario debe oprimir simultáneamente las teclas "Ctrl" y "x".

CONSOLE>> cdxfile dosfile xinxfile "cdxfile"

Copia (transforma) un archivo DOS a un archivo XINIX. La longitud del nombre del archivo xinx debe ser menor o igual a 9 caracteres.

CONSOLE>> chprio id_proceso newprio "chprio"

Cambia la prioridad de un proceso. La identificación de un proceso es un valor entero que puede conocerse con el comando "ps": process status. La nueva prioridad a asignarle, newprio, es también un valor entero entre 1 y 32767.

CONSOLE>> close dispositivo "close"

Cierra un dispositivo (archivo); "dispositivo" es un valor entero que puede conocerse con el comando "devs", se utiliza el campo "Num". Un proceso puede olvidar cerrar los archivos que utiliza.

CONSOLE>> cp fromfile tofile "cp"

Copia el contenido del archivo fromfile al archivo tofile. Este último no debe existir. Recuérdese que los nombres de archivo XINIX no deben tener más de 9 caracteres.

CONSOLE>> create file [tamaño_pila prioridad] "create"

Crea un proceso a partir de un programa en un archivo XINIX. Si no se especifica tamaño de la pila o prioridad, el proceso recibirá 2500 bytes para pila, y una prioridad de 5. El proceso queda en estado suspendido: no se ejecuta. Si desea cambiar sólo la prioridad, por ser éste el segundo argumento que create recibe, debe especificarse antes un tamaño de pila. Sin embargo, si es posible modificar sólo el tamaño de la pila. Ver los comandos exec y resume.

CONSOLE>> created file [tamaño_pila prioridad] "created"

Crea un proceso a partir de un programa en un archivo DOS. Se aplican las mismas observaciones que al comando create. Ver los comandos execd y resume.

CONSOLE>> devs "devs"

Despliega información acerca de los dispositivos del sistema: terminales, discos y archivos.

CONSOLE>> dir "dir"

Despliega el directorio de un diskette DOS. El diskette debe encontrarse en el manejador de disco A.

CONSOLE>> echo "leyenda" "echo"

Despliega el texto entre comillas, en este caso leyenda, en la pantalla de la terminal.

CONSOLE>> exec file [tamaño_pila prioridad] "exec"

Crea un proceso a partir de un programa ejecutable en un archivo XINIX. Si no se especifica tamaño de la pila o prioridad, el proceso recibirá 2500 bytes para pila, y una prioridad de 5. El

proceso queda en estado READY: empieza a ejecutarse. Ver el comando create.

CONSOLE>> `execd file [tamaño_pila prioridad]` "execd"

Crea un proceso a partir de un programa ejecutable en un archivo DOS. Se aplican las mismas observaciones que al comando `execd`. Ver el comando `created`.

CONSOLE>> `exit` "exit"

Termina la sesión del usuario. Ya no es posible seguir ejecutando comandos. El comando `logout` es equivalente.

CONSOLE>> `fmt` "fmt"

Formatea un diskette con la estructura del sistema de archivos XINIX. El diskette debe colocarse en el manejador de disco A.

CONSOLE>> `help` "help"

Despliega la lista de comandos disponibles al usuario. El comando "?" es equivalente.

CONSOLE>> `kill id_proceso` "kill"

Termina un proceso. La identificación del proceso a terminar es un valor entero que se obtiene con el comando "ps".

CONSOLE>> `logout` "logout"

Termina la sesión del usuario. Ya no es posible seguir ejecutando comandos. El comando `exit` es equivalente.

CONSOLE>> `ls` "ls"

Despliega el directorio de un diskette XINIX. El disco debe encontrarse en el manejador de disco A.

CONSOLE>> mem "mem"

Muestra la distribución de la memoria: cantidad (de bytes) libre al empezar el sistema; cantidad libre actual; cantidad ocupada por las pilas de los procesos; cantidad de la memoria "heap" (la que ocupan las instrucciones de los procesos de usuario, los pools y todo manejo dinámico de memoria); y la lista de los bloques libres: su dirección y su longitud en bytes.

CONSOLE>> mv file tofile "mv"

Cambia el nombre del archivo file a tofile.

CONSOLE>> ps "ps"

"Process Status", de cada uno de los procesos en el sistema, despliega sus atributos: identificación, nombre, prioridad, estado, etcétera. Otros comandos requieren que se ejecute primero "ps", para así obtener la identificación del proceso que desean afectar.

CONSOLE>> resume id_proceso "resume"

Reanuda la ejecución de un proceso en estado suspendido. La identificación del proceso a reanudar es un valor entero que puede conocerse con el comando "ps".

CONSOLE>> rm file "rm"

Borra (ReMueve) un archivo.

CONSOLE>> send id_proceso mensaje "send"

Envía un mensaje a un proceso. La identificación de éste es un valor entero que puede conocerse con el comando "ps". El mensaje a enviar es también un valor entero.

CONSOLE>> signal id_semáforo "signal"

Incrementa en uno al contador asociado a un semáforo. Si la cuenta era negativa antes del incremento, desbloquea al primer proceso bloqueado. La identificación del semáforo es un valor entero que puede obtenerse con el comando "ps"; pues éste, entre otras cosas,

despliega el estado de cada proceso, y si este es WAIT, incluye el número de semáforo que bloqueó al proceso.

CONSOLE>> signaln id_semáforo n "signaln"

Incrementa en "n" la cuenta asociada a un semáforo. Si hay procesos bloqueados desbloquea hasta 'n' si los hay. La identificación del semáforo es un valor entero que puede obtenerse con el comando "ps", pues éste, entre otras cosas, despliega el estado de cada proceso, y si éste es WAIT, incluye el número de semáforo que bloqueó al proceso.

CONSOLE>> sleep n "sleep"

Suspende la ejecución del proceso "intérprete de comandos" de una terminal por "n" segundos. Mientras dure la suspensión, el usuario puede utilizar la terminal para introducir o escribir datos hacia, o desde sus programas. Este comando es útil cuando las aplicaciones se encuentran aún en desarrollo; pues si el intérprete no es suspendido y un proceso lee o escribe datos, es imposible determinar quién leerá que datos; mientras que la "escritura" se mezcla de manera incongruente. La suspensión temporal permite al usuario tomar, posteriormente, de nuevo el control. Una vez terminada la aplicación del usuario, y si ésta lee o escribe datos a la terminal, se recomienda utilizar el comando . . .

CONSOLE>> suspend id_process "suspend"

Suspende indefinidamente la ejecución de un proceso, cuyo estado deber ser READY o CURRENT. La identificación del proceso a suspender es un valor entero que puede conocerse con el comando "ps". Si el usuario desea suspender al proceso intérprete de comandos de su terminal, y así utilizar a ésta para leer o escribir datos desde sus procesos, se le recuerda que los procesos intérpretes se nombran según el prompt que despliegan para aceptar comandos; así, cuando el usuario ejecute el comando "ps", podrá reconocer a su intérprete y obtener la identificación de proceso (pid: valor entero) que le corresponde. Ver los comando sleep y resume.

CONSOLE>> vrf

"vrf"

Verifica los sectores de un diskette DOS o XINIX. La verificación consiste en checar que se puedan leer. Si se encuentran sectores erroneos despliega la cantidad en bytes que constituyen.

CONSOLE>> who

"who"

Despliega la identificación de los usuarios en cada una de las terminales del sistema.

`breakt()``breakt`

Deshabilita la transmisión de un puerto de comunicación serie.

Sinopsis:

```
#include "xinixh.h"
```

```
int breakt(puerto)
int puerto;
```

Descripción:

Forza a un 0 lógico la transmisión de un puerto de comunicación serie, sin importar ninguna otra actividad del transmisor. Las funciones que XINIX provee para puertos serie asumen que se trata de tarjetas de comunicación asíncrona usando el chip 8250 (ver [TRIBMS3] pag. 1-223 en adelante). Cada tarjeta tiene varios registros para diferentes funciones. En la IBM-PC es posible conectar 2 de éstas tarjetas, sus registros quedan en los siguientes puertos en valor hexadecimal:

tarj-1 tarj-2 Registro seleccionado

3f8	2f8	Buffer de transmisión
3f8	2f8	Buffer de recepción
3f8	2f9	Divisor Latch LSB
3f9	2f9	Divisor Latch MSB
3f9	2f9	Interrupt enable register
3fa	2fa	Interrupt identification rg
3fb	2fb	Line control register
3fc	2fc	Modem control register
3fd	2fd	Line status register
3fe	2fe	Modem status register

La identificación del puerto que el usuario debe enviar a `breakt`, y a todos los procedimientos XINIX que trabajan con puertos serie, debe ser la del puerto que corresponde al primer registro de la

tarjeta: 3f8 o 2f8. El procedimiento en cuestión se encargará de acceder el registro involucrado para llevar a cabo la función deseada. Así, breakt() se encarga de acceder el registro de control correspondiente (3fb o 2fb) para deshabilitar la transmisión.

Ver también

clrints(), disablet(), disabler(), disablem(), disables(), enablet(), enabler(), enablem(), enables(), erroronr(), initport(), nobreakt(), readyr(), readyt().

chprio()	chprio
----------	--------

Cambia la prioridad de un proceso

Sinopsis:

```
#include "xinixh.h"
```

```
int chprio(pid, newprio)
int pid;
int newprio;
```

Descripción:

Chprio() cambia la prioridad de calendarización del proceso "pid" a "newprio". Las prioridades son enteros positivos. En todo momento, el proceso con prioridad más alta que está en estado READY será él que corra. Un conjunto de procesos con igual prioridad es calendarizado round-robin.

Si la nueva prioridad o la identificación del proceso es inválida, chprio() devuelve SYSERR; en caso contrario, la prioridad anterior del proceso. La prioridad del proceso "nulo" no puede cambiarse, siempre permanece en cero.

Ver también

create(), get_priority(), get_status(), resume().

close()	close
---------	-------

Cierra un dispositivo.

Sinopsis:

```
#include "xinixh.h"
```

```
int close(dev)
int dev;
```

Descripción:

Close() deshabilita las operaciones de lectura/escritura con el dispositivo dev. Close() devuelve SYSERR si dev es incorrecto o no está abierto; de otra manera devuelve OK. En esta versión de XINIX los dispositivos que se cierran y abren son archivos en disco. Las terminales no deberían ser abiertas ni cerradas. Una explicación más detallada acerca del manejo de dispositivos se da en el servicio control().

Ver también

control(), getc(), open(), putc(), read(), seek(), write()

clrints()	clrints
-----------	---------

Deshabilita todos los tipos de interrupciones que puede efectuar un puerto de comunicación serie.

Sinopsis:

```
#include "xinixh.h"
```

```
int clrints(puerto)
int puerto;
```

Descripción:

Un puerto de comunicación serie puede interrumpir por recibir un carácter, haber enviado un carácter, cambio en el estado de la línea o cambio en el estado del modem. Clrints() deshabilita todas las interrupciones. No devuelve ningún valor. La identificación del puerto debe enviarse conforme se explica en breakt().

Ver también

breakt(), disablet(), disabler(), disablem(), disables(), enablet(), enabler(), enablem(), enables(), erroronr(), initport(), nobreakt(), readyr(), readyt().

Controla un dispositivo.

Sinopsis:

```
#include "xinxh.h"
```

```
int control(dev, function, arg1, arg2)
int dev;
int function;
char *arg1;
char *arg2;
```

Descripción:

Control() es el mecanismo usado para enviar información de control a dispositivos y sus manejadores o para interrogar su estado. Los datos fluyen normalmente a través de getc(), putc(), read() y write().

Control() devuelve SYSERR si dev es incorrecto o si la función no puede ser ejecutada o no existe; de otra manera, el valor devuelto depende del dispositivo. Por ejemplo, para los dispositivos tipo terminal hay una función de control que devuelve el número de caracteres tecleados sin haberse leído: esperando en la cola de entrada. A continuación se listan fragmentos del archivo xinxh.h, que el usuario debe incluir en todas sus aplicaciones; en él se encuentran la definición de los dispositivos válidos que puede utilizar con el servicio control, así como las funciones permitidas con terminales y diskettes.

```
/* Device name definitions */
```

```
#define CONSOLE      0      /* type tty, terminal IBM */
#define OTHER_1     1      /* type tty, term. serie, pto. 3f8 */
#define OTHER_2     2      /* type tty, term. serie, pto. 2f8 */
#define DISKO       3      /* type dsk */
#define DISK1       4      /* type dsk, no se ve el disco B */
#define FILE1       5      /* type df */
#define FILE2       6      /* type df */
#define FILE3       7      /* type df */
#define FILE4       8      /* type df */

#define BADDEV      -1     /* used when invalid dev needed */
```

```
/* ttycontrol funtion codes */
```

```
#define TCSETBRK    1      /* turn on BREAK in transmitter */
#define TCRSTBRK    2      /* turn off BREAK " " */
#define TCNEXTC     3      /* look ahead 1 character */
#define TCMODER     4      /* set input mode to raw */
#define TCMODEC     5      /* set input mode to cooked */
#define TCMODEK     6      /* set input mode to cbreak */
```



```

#define TCICHARS      8      /* return number of input chars */
#define TCECHO       9      /* turn on echo */
#define TCNOECHO     10     /* turn off echo */
#define TCSTABS      11     /* set tab = # blanks */
#define TCINT        12     /* set pid to interrupt on 'Ctrl B' */

/* Disk control function codes. Several work over files */

#define DSKSYNC      0      /* synchronize (flush all I/O) */
#define DSKFMT       1      /* formatea las pistas de un disco */
#define DSKVRF       2      /* verifica las pistas de un disco */
#define DSK_FILE_RM  3      /* remove file from a specified disk */
#define DSK_FILE_MV  4      /* move (change name of) a file " " */
#define DSK_FILE_RDIR 5      /* read directory on a specified device */
#define DSK_FILE_LDIR 6      /* list directory on a specified device */

```

El usuario puede construir sus propias macros para realizar funciones específicas en un tipo de dispositivo. Por ejemplo, la siguiente macro borra un archivo en disco:

```
#define deletef(dev,namefile) control((dev),DSK_FILE_RM,(namefile))
```

Ver también

close(), getc(), open(), putc(), read(), seek(), write().

create()	-	create
----------	---	--------

Crea un nuevo proceso.

Sinopsis:

```
#include "xinixh.h"
```

```

int create(caddr, ssize, prio, name, nargs [,arguments])
char *caddr;
int ssize;
int prio;
char *name;
int nargs;
int arguments; /* argumento inicial en la lista de argmnts */

```

Descripción:

Create() crea un nuevo proceso que empezará su ejecución en la localidad "caddr", con una pila de "ssize" palabras, con prioridad inicial "prio" e identificado con el nombre "name". "Caddr" debe ser la dirección de un procedimiento o programa principal. Si la creación es exitosa, la identificación del proceso, un valor no

negativo, es devuelto al proceso que llama a create. El proceso creado se deja en estado suspendido: no empezará su ejecución hasta que sea iniciada con el servicio "resume". Si los argumentos son incorrectos o no hay una entrada disponible en la tabla de procesos, create() devuelve el valor SYSERR. El nuevo proceso tiene su propia pila para datos temporales, pero comparte datos globales con otros procesos de acuerdo a las reglas del lenguaje C. Si el proceso intenta regresar es terminado.

El proceso creador que llama a create() puede pasar un número variable de argumentos al proceso a crear, los cuales son alcanzados a través de los parámetros formales. El valor entero "nargs" especifica cuantos argumentos, de tipo entero, siguen. "Nargs" enteros, a partir de la lista de argumentos, se pasan al proceso creado.

Ver también

kill()

disablem()

disablem

Deshabilita la interrupción de un puerto de comunicación serie por cambio en el estado del modem. Sinopsis:

```
#include "xinixh.h"
```

```
int disablem(puerto)
int puerto;
```

Descripción:

Disablem() no devuelve valor alguno. La identificación del puerto debe enviarse conforme se explica en breakt(). Ver [TRIBM83] pag. 1-242, para conocer las causas que cambian el estado del modem.

Ver también

```
breakt(), clrints(), disabler(), disables(), disablet(), enablem(),
enabler(), enables(), enablet(), erroronr(), initport(),
nobreakt(), readyr(), readyt().
```

disabler()

disabler

Deshabilita la interrupción de un puerto de comunicación serie por recibir un carácter.

Sinopsis:

```
#include "xinixh.h"
```

```
int disabler(puerto)  
int puerto;
```

Descripción:

Disabler() no devuelve valor alguno. La identificación del puerto debe enviarse conforme se explica en break(). Después de ejecutar disabler() sobre un puerto de comunicación serie, el usuario puede utilizar readyr() y readport() para leer caracteres desde el puerto en forma polling.

Ver también

break(), clrints(), disablem(), disables(), disablen(), enablem(), enabler(), enables(), enablen(), erroronr(), initport(), nobreak(), readport(), readyr(), readyt().

disables()

disables

Deshabilita la interrupción de un puerto de comunicación serie por cambio en el estado de la línea de transmisión

Sinopsis:

```
#include "xinixh.h"
```

```
int disables(puerto)  
int puerto;
```

Descripción:

El cambio en el estado de la línea puede deberse a un error de sobreposición: llegó un carácter sin haber leído el anterior; un error de paridad; un error de trama o un error a causa de desconexión de la línea. Ver [TRIBM83] pag. 1-242. Disables() no devuelve valor alguno. La identificación del puerto debe enviarse conforme se explica en break().

Ver también

break(), clrints(), disablem(), disabler(), disablen(), enablem(),

enabler(), enables(), enablet(), erroronr(), initport(),
nobreakt(), readport(), readyr(), readyt().

disablet()

isablet

Deshabilita la interrupción de un puerto de comunicación serie por registro de transmisión vacío: listo para enviar otro carácter.

Sinopsis:

```
#include "xinixh.h"
```

```
int disablet(puerto)  
int puerto;
```

Descripción:

Disablet() no devuelve valor alguno. La identificación del puerto debe enviarse conforme se explica en breakt(). Después de ejecutar disablet() sobre un puerto de comunicación serie, el usuario puede utilizar readyt() y writport() para escribir caracteres hacia un puerto en forma polling.

Ver también

breakt(), clrints(), disablem(), disablel(), disables(), enablem(),
enabler(), enables(), enablet(), erroronr(), initport(),
nobreakt(), readyr(), readyt(), writport().

disadisp()

disadisp

Deshabilita a un dispositivo físico, no interrumpe más al manejador de interrupciones 8259.

Sinopsis:

```
#include "xinixh.h"
```

```
int disadisp(id_dispositivo)  
int id_dispositivo;
```

Descripción:

Un 8259 maneja las interrupciones de 8 dispositivos. En la configuración normal de una IBM-PC la identificación de cada uno de ellos, al utilizar disadisp() o enabdisp(), es la siguiente:

```

id dispositivo
0 reloj del sistema
1 teclado
2 reservado
3 puerto de comunicación (puerto inicial 2f8)
4 puerto de comunicación (puerto inicial 3f8)
5 disco duro
6 diskette
7 impresora

```

Si el usuario implementa un manejador para un dispositivo y desea controlarlo por interrupciones, debe poner la dirección del procedimiento, que constituye al manejador, en el vector de interrupciones asociado al dispositivo; y habilitar éste a interrumpir al 8259. Si eventualmente el usuario ya no desea que el 8259 reconozca las interrupciones del mencionado dispositivo, lo deshabilitaría con `disadisp()`.

Ver también

```
break(), enabdisp(), init8259().
```

enabdisp()	enabdisp
------------	----------

Habilita a un dispositivo físico a interrumpir al manejador de interrupciones 8259.

Sinopsis:

```
#include "xinixh.h"
```

```
int enabdisp(id dispositivo)
```

```
int id dispositivo;
```

Descripción:

Supóngase que el usuario desea implementar un manejador para un puerto de comunicación serie controlado por interrupciones; ¿qué debe hacer? El código que se lista a continuación muestra los pasos a seguir:

```
main() {
```

```
int port, int manejador();
```

```
port = 0x3f8; /* puerto inicial del puerto de com. serie */
```

```
initport(port, BR_4800, BTR_8, SB_1, ParityNn);
```

```
disable();
```

```
load_vector(0x0C, manejador);
```

```
enabler(port); /* Habilita la interrupción por recibir cars. */
```

```
/* pues initport las deja deshb. */
```

```
enabdisp(4); /* hab al puerto a interrumpir. */
```

```
/* 4 es la id para el 8259, del puerto de */  
/* comunicación serie mapeado a partir */  
/* del puerto 0x3f8 */
```

```
enable();  
*  
*
```

Primero se programa el puerto con `initport()`: velocidad de transmisión (4800 bits/seg), 8 bits por carácter, 1 bit de parada y paridad non. La identificación del puerto a programar que recibe `initport()` es conforme se explica en `breakt()`. `Disable()` (del lenguaje turboC) inhibe la CPU para atender interrupciones; mientras que `load_vector()` carga la dirección del procedimiento "manejador" que las atenderá. El vector de interrupción 12 (0x0c), en la configuración normal de la IBM-PC, corresponde a la tarjeta de comunicación serie, cuyo primer registro queda en el puerto 3f8 hexadecimal. `Enabler()` habilita al puerto de comunicación para que interrumpa cada que reciba un carácter. `Enabdisp()` permite que el manejador de interrupciones 8259, reconozca las interrupciones que provoque el puerto de comunicación serie. Por último, `enable()` (del lenguaje turboC) permite que la CPU atienda de nuevo las interrupciones. Si se tratará de otro dispositivo, disco o reloj, la inicialización y programación cambiarían, más habilitarles para interrumpir al 8259 se haría con `enabdisp()`. Cabe mencionar que `load_vector()` es una macro definida en `xinixh.h`, así como todas las constantes simbólicas utilizadas por `initport()`: están disponibles al usuario.

Ver también

`breakt()`, `disadisp()`, `enabler()`, `init8259()`, `initport()`.

<code>enablem()</code>

<code>enablem</code>

Habilita la interrupción de un puerto de comunicación serie por cambio en el estado del modem.

Sinopsis:

```
#include "xinixh.h"
```

```
int enablem(puerto)  
int puerto;
```

Descripción:

`Enablem` no devuelve valor alguno. La identificación del puerto debe enviarse conforme se explica en `breakt()`. Ver [TRIBM83] pag. 1-242, para conocer las causas que cambian el estado del modem.

Ver también

break(), clrints(), disablem(), disabler(), disables(),
disablet(), enabler(), enables(), enablet(), erroronr(),
initport(), nobreakt(), readyr(), readyt().

enabler()

enabler

Habilita la interrupción de un puerto de comunicación serie por recibir un carácter:

Sinopsis:

```
#include "xinixh.h"
```

```
int enabler(puerto)  
int puerto;
```

Descripción:

Enabler() no devuelve valor alguno. La identificación del puerto debe enviarse conforme se explica en breakt().

Ver también

```
break(), clrints(), disablem(), disabler(), disables(),  
disablet(), enabledisp(), enablem(), enables(), enablet(),  
erroronr(), initport(), nobreakt(), readyr(), readyt().
```

enables()

enables

Habilita la interrupción de un puerto de comunicación serie por cambio en el estado de la línea.

Sinopsis:

```
#include "xinixh.h"
```

```
int enables(puerto)  
int puerto;
```

Descripción:

Enables() no devuelve valor alguno. La identificación del puerto debe enviarse conforme se explica en breakt().

Ver también

```
break(), clrints(), disablem(), disabler(), disables(),  
disablet(), enablem(), enabler(), enablet(), erroronr(),  
initport(), nobreakt(), readyr(), readyt().
```

enablet()	enablet
-----------	---------

Habilita la interrupción de un puerto de comunicación serie por registro de transmisión vacío: listo para enviar otro carácter.

Sinopsis:

```
#include "xinixh.h"
```

```
int enablet(puerto)
int puerto;
```

Descripción:

Enablet() no devuelve valor alguno. La identificación del puerto debe enviarse conforme se explica en breakt().

Ver también

```
breakt(),    clrints(),    disablem(),    disabler(),    disables(),
disablet(),  enablem(),    enabler(),    enables(),    erroronr(),
initport(), nobreakt(), readyr(), readyt().
```

erroronr()	erroronr()
------------	------------

Devuelve TRUE si hubo error al recibir un carácter por un puerto de comunicación serie, FALSE en caso contrario.

Sinopsis:

```
#include "xinixh.h"
```

```
int erroronr(puerto) int puerto;
```

Descripción:

La identificación del puerto debe enviarse conforme se explica en breakt(). Los valores TRUE y FALSE están definidos en el archivo "xinixh.h". Se recomienda al usuario checar por el estado de la recepción al recibir un carácter:

```
/* se ha recibido un carácter por interrupción */
if (erroronr(puerto)) {          /* error al recibir? */
    ACCION EN CASO DE ERROR
} else {
    car = readport(port);      /* lee el carácter */
}
*
```


Ver también

breakt(), clrints(), disablem(), disabler(), disables(),
disablet(), enablem(), enabler(), enables(), enablet(), initport(),
nobreakt(), readport(), readyr(), readyt().

fgetc()	fgetc
---------	-------

Ver getc().

fgets()	fgets
---------	-------

Ver gets().

fprintf()	fprintf
-----------	---------

Ver printf().

fputc()	fputc
---------	-------

Ver putc().

fputs()	fputs
---------	-------

Ver puts().

freebuf()	freebuf
-----------	---------

Devuelve un buffer al pool del cual fué obtenido. Ver mkpool().

freemem()	freemem
-----------	---------

Devuelve un bloque de memoria obtenido con getmem().

Sinopsis:

```
#include "xinixh.h"
```

```
int freemem(p, len)
char *buf;
long len;
```

Descripción:

Freemem() devuelve el bloque de memoria apuntado por "p" y con longitud "len". En realidad, freemem() es una macro definida en "xinixh.h" de la siguiente manera:

```
#define freemem(p,len) freememory((p), (long)(len))
```

...con el único fin de evitar que el usuario devuelva una cantidad tipo entero y no long; lo que provocaría un recorrimiento de los parámetros de acuerdo a como los espera freememory(). El cual devuelve OK si la dirección del bloque y su tamaño son correctos; SYSERR en caso contrario. Se recomienda al usuario no usar freememory directamente.

Ver también

getmem().

fscanf()	fscanf
----------	--------

Ver scanf()

getbuf()	getbuf
----------	--------

Obtiene un buffer de un pool. Ver mkpool().

getc()	getc
--------	------

fgetc()
getchar()

Leen un carácter de un dispositivo.

Sinopsis:

```
#include "xinxh.h"
```

```
int getc(dev)  
int dev;
```

Getc() lee el siguiente carácter del dispositivo identificado por "dev". En caso de éxito, getc() devuelve el carácter leído. Si "dev" es un archivo, getc puede devolver EOF, definido en xinxh.h, en caso de no existir más caracteres. En realidad, getc es una macro definida así:

```
#define getc xgetc
```

...con el fin de evitar que el usuario llame la rutina getc() del lenguaje turboC. "Xgetc()" es la rutina de XINIX que lee caracteres de un dispositivo. Por otra parte, fgetc() y getchar() son macros definidas de la siguiente manera:

```
#define getchar() xgetc(CONSOLE)  
#define fgetc(unit) xgetc((unit))
```

...con el único propósito de asumir la convención del lenguaje C. Se notará que getchar() lee caracteres sólo de la terminal de memoria mapeada.

Ver también

close(), control(), open(), putc(), read(), seek(), write().

getchar()	getchar
-----------	---------

Ver getc().

getmem()	getmem
----------	--------

Obtiene un bloque de memoria.

Sinopsis:

```
#include "xenix.h"
```

```
int *getmem(len)  
long len;
```

Descripción:

Getmem() obtiene un bloque de memoria con longitud de "len" bytes. En realidad, getmem() es una macro definida en "xenix.h" de la siguiente manera:

```
#define getmem(len) getheapmem((long)(len))
```

... con el único fin de evitar que el usuario envíe una cantidad tipo entero y no long; lo que provocaría que getheapmem() reciba una longitud errónea. Si hay memoria disponible, getheapmem() (getmem) devuelve la dirección donde empieza; SYSERR en caso contrario. Se recomienda al usuario no usar getheapmem() directamente.

Ver también

freemem().

getpid()	getpid
----------	--------

Obtiene la identificación del proceso que corre actualmente.

Sinopsis:

```
int getpid()
```

Descripción:

En ocasiones, es necesario que un proceso conozca su identificación para realizar ciertas operaciones. Por ejemplo, si un proceso desea cambiar su prioridad, lo haría de la siguiente manera:

```
chprio(getpid(), newprio);
```

gets()	gets
--------	------

fgets

Obtienen una cadena de caracteres de un dispositivo.

Sinopsis:

```
#include "xinxh.h"
```

```
char *gets(s)
char *s;
```

```
char *fgets(s, n, dev)
char *s;
int n, dev;
```

Descripción:

Gets() lee una cadena de caracteres en "s" desde la terminal de memoria mapeada CONSOLE. La cadena termina al recibir un carácter NEWLINE ("\n"), el cual es reemplazado en "s" por un carácter nulo (el valor 0). gets devuelve su argumento: "s".

Fgets() lee n-1 caracteres o hasta leer un carácter NEWLINE, lo que ocurra primero, del dispositivo dev, en "s". Después del último carácter leído, fgets pone un carácter nulo (0). Fgets() devuelve su primer argumento, "s", si lee correctamente; en caso contrario, la constante apuntador NULL definida en xinxh.h.

Ver también

control() para conocer la identificación de los dispositivos XINIX, gets(), puts(), scanf().

Consideraciones

Gets() borra un NEWLINE, fgets() lo mantiene. Este manejo se conforma a las funciones del lenguaje TurboC.

get_priority	get_priority
--------------	--------------

Obtiene la prioridad de un proceso.

Sinopsis:

```
int get_priority(pid)
int pid;
```

Descripción:

get_priority devuelve la prioridad del proceso pid.

get_status

get_status

Obtiene el estado de un proceso.

Sinopsis:

```
#include "xinixh.h"
```

```
int get_status(pid)
int pid;
```

Descripción:

Get_status() devuelve el estado del proceso pid: PCURR, PRFREE, PRREADY, PRRECV, PRSLEEP, PRSUSP o PRWAIT, definidas en xinixh.h.

init8259

init8259

Programa al manejador de interrupciones 8259.

Sinopsis:

```
int init8259(vector)
int vector;
```

Descripción:

Programa al 8259 tal y como lo hace el BIOS de la IBM-PC, y opcionalmente, establece el número de vector de interrupción inicial asociado al primer dispositivo que puede interrumpir. Son 8 los dispositivos que un 8259 puede controlar, a cada uno se asigna un vector de interrupción a partir del vector inicial. El 8259 pasa al 8086 el número de vector en el que se encuentra la dirección de la rutina que atenderá la interrupción provocada por un dispositivo. El usuario no debe utilizar este procedimiento, a menos que entienda y modifique la inicialización de los vectores de interrupción que XINIX lleva a cabo al empezar. Ver [INTEL81] pag. 7-130 y [TRIBM83] pag. A-9. Init8259() no devuelve valor alguno.

Ver también

```
intpend(), pol8259(), npol8259().
```

Programa un puerto de comunicación serie.

Sinopsis:

```
#include "xinixh.h"
```

```
int initport(puerto, vel, bpc, bp, p)
int puerto;
int vel;
int bpc;
int bp;
int p;
```

Descripción:

Programa un puerto de comunicación serie para transmitir a velocidad "vel", caracteres con "bpc" bits, "bp" bits de parada y paridad "p". La identificación del puerto debe enviarse conforme se explica en break(). Los valores que el usuario debe enviar para cada uno de los parámetros de initport() ya están definidos como constantes simbólicas en "xinixh.h". A continuación se lista parte de éste, ver la ejemplificación de uso:

/* Constantes utilizadas por la versio'n IBM PC. En particular para la programacio'n del puerto de comunicacio'n asincrona. Ver Technical Reference Manual IBM PC, p 1-230 -> */

```
#define BR_19200 0x0006 /* Baud Rate */
#define BR_9600 0x000C
#define BR_7200 0x0010
#define BR_4800 0x0018
#define BR_3600 0x0020
#define BR_2400 0x0030
#define BR_2000 0x003A
#define BR_1800 0x0040
#define BR_1200 0x0060
#define BR_600 0x00C0
#define BR_300 0x0180
#define BR_150 0x0300
#define BR_134 0x0359
#define BR_110 0x0417
#define BR_75 0x0600
#define BR_50 0x0900

#define BTR_5 0x0000 /* # de Bits a Transmitir y Reci */
#define BTR_6 0x0001
#define BTR_7 0x0002
#define BTR_8 0x0003

#define SB_1 0x0000 /* # de Stop Bits */
#define SB_2 0x0001
```

```
#define NoParity 0x0000 /* Tipo de paridad a manejar */
#define ParityPr 0x0018 /* paridad par */
#define ParityNn 0x0008 /* paridad non */
```

```
/* Ejemplo de utilizacio'n:
   initport(port, BR_9600, BTR_7, SB_1, ParityNn); */
```

Ver también

```
break(), clrrints(), disablem(), disabler(), disables(),
disablet(), enabler(), enables(), enablet(), erroronr(),
initport(), nobreakt(), readyr(), readyt().
```

intpend

intpend

Devuelve la identificación (0-7) del dispositivo que interrumpió al manejador de interrupciones 8259.

Sinopsis:

```
#include "xinixh.h"
```

```
int intpend()
```

Descripción:

Cuando el 8259 no esta habilitado para interrumpir a la CPU (se encuentra en modo "polling"), el usuario puede utilizar a intpend() para saber si hay una interrupción pendiente por atender. Intpend() devuelve la identificación del dispositivo que interrumpió; SYSERR en caso contrario.

Ver también

```
pol8259(), npol8259().
```


kill	kill
------	------

Termina un proceso

Sinopsis:

```
int kill(pid)
int pid;
```

Descripción:

Kill() para el proceso pid y lo remueve del sistema, devolviendo SYSERR si la identificación del proceso es inválida; OK en caso contrario. Kill termina un proceso inmediatamente. Si el proceso se encuentra encolado en un semáforo es removido de la lista asociada a éste, la cuenta asociada al semáforo también se incrementa en 1: como si el proceso nunca hubiera estado ahí. Procesos esperando enviar un mensaje a un puerto desaparecen sin afectar a éste. Si el proceso está esperando por lectura o escritura, ésta se detiene si es posible.

Uno puede matar a un proceso en cualquier estado, incluso a uno en estado suspendido. Una vez terminado con kill(), un proceso no puede ser recuperado.

kprintf	kprintf
---------	---------

Ver printf.

mkpool	mkpool
--------	--------

```
freebuf
getbuf
```

Servicios para el manejo de pools de buffers: conjuntos de bloques de memoria controlados.

Sinopsis:

```
int freebuf(buf)
char *buf;

char *getbuf(poolid)
int poolid;
```

```
int mkpool(bufsiz, numbufs)
int bufsiz, numbufs;
```

Descripción:

Mkpool() crea un pool con "numbufs" buffers, cada uno con tamaño "bufsiz" bytes. Mkpool() devuelve un entero identificando al pool. Si no se puede crear el pool por falta de memoria o los argumentos son incorrectos, mkpool() devuelve SYSERR.

Una vez que se ha creado un pool, getbuf() obtiene un buffer libre del pool identificado por poolid y devuelve la dirección de la primera palabra del buffer. Si todos los buffers en el pool especificado están en uso, el proceso que llama a getbuf() es bloqueado hasta que uno este disponible. Si el argumento poolid no especifica un pool válido, getbuf() devuelve SYSERR.

Freebuf() devuelve un buffer al pool del cual fué tomado. Freebuf() devuelve OK si la devolución del buffer es correcta; SYSERR si buf no apunta a un buffer válido en un pool.

Deficiencias:

Actualmente no hay manera de liberar el espacio ocupado por un pool, aún cuando sus buffers ya no se utilicen.

nobreakt()

nobreakt

Reestablece la transmisión de un puerto de comunicación serie.

Sinopsis:

```
#include "xinixh.h"
```

```
int breakt(puerto)
int puerto;
```

Descripción:

Constituye la operación contraria del servicio breakt(). La identificación del puerto debe enviarse como se especifica en breakt().

Ver también

```
clrints(), disablem(), disabler(), disables(), disablet(),
enablem(), enabler(), enables(), enablet(), erroronr(), initport(),
breakt(), readyr(), readyt().
```

npol8259()

npol8259

Quita al manejador de interrupciones 8259 del modo 'polling'. Ver pol8259.

open()

open

Abre un dispositivo.

Sinopsis:

```
int open(dev)
int dev;
```

Descripción:

Open() establece una conexión para realizar operaciones de lectura/escritura con el dispositivo "dev". Open() devuelve SYSERR si "dev" es incorrecto o no puede ser abierto; en caso contrario, la identificación del archivo. No tiene sentido, no es necesario, en la versión actual de XINIX, abrir un dispositivo tipo "terminal" o "disco". El servicio control() explica con más detalle el manejo de los dispositivos.

Ver también

close(), contro(), getc(), putc(), read(), seek(), write().

pcount()	pcount
----------	--------

Devuelve el número de mensajes esperando en un puerto a ser leídos.

Sinopsis:

```
int pcount(portid)
int portid;
```

Descripción:

En realidad, pcount() devuelve la cuenta asociada a un puerto si la identificación de éste es correcta; SYSERR en caso contrario.

Una cuenta positiva "p", significa que hay "p" mensajes disponibles para leerse. Una cuenta negativa "p" significa que hay "p" procesos esperando que llegue un mensaje. Una cuenta igual a cero significa que no hay mensajes ni procesos esperando por mensajes.

Ver también

pcreate(), pdelete(), preceive(), preset(), psend().

Deficiencias

En esta versión de XINIX SYSERR tiene el valor -1, que corresponde a una cuenta válida asociada a un puerto. Pcount() debe modificarse para devolver cuentas negativas como enteros positivos muy grandes, o para devolver un valor que no sea igual a SYSERR.

pcreate()	pcreate
-----------	---------

Creación de un nuevo puerto para enviar mensajes largos.

Sinopsis:

```
int pcreate(count)
int count;
```

Descripción:

Pcreate() crea un puerto con "count" localidades para almacenar mensajes largos.

Pcreate() devuelve un entero identificando al nuevo puerto si la creación es correcta. Si no hay espacio para crear al puerto o "count" no es positiva, pcreate() devuelve SYSERR. En esta versión de XINIX el número total de localidades para mensajes en todos los puertos es 100.

Los puertos son manejados con psend() y preceive(). Recibir de un

puerto devuelve un apuntador a un mensaje que fué previamente enviado al puerto.

Ver también

pcount(), pdelete(), preceive(), preset(), psend().

pdelete()

pdelete

Borra un puerto para manejo de mensajes largos.

Sinopsis:

```
int pcreate(portid)
int portid;
```

Descripción:

Pdelete() borra el puerto "portid" y devuelve OK si todo sale bien; SYSERR si portid no es válido o no está creado.

Si al borrar un puerto hay procesos esperando por mensajes en él, estos procesos son puestos en estado READY para obtener SYSERR al volver del llamado a preceive(). Si hay mensajes en el puerto, las localidades que ocupan se devuelven a la lista de libres: los mensajes son borrados. Si hay procesos que esperan colocar un mensaje en el puerto, son puestos en estado READY para obtener SYSERR al volver del llamo a psend(): como si el puerto nunca hubiera existido. Pdelete() realiza la misma función de limpiar y liberar procesos de un puerto tal y como lo hace preset(), excepto que pdelete() también libera al puerto.

Ver también

pcount(), pcreate(), preceive(), preset(), psend().

pol8259()

pol8259

npol8259()

Servicios para establecer el manejo por interrupciones o polling del manejador de interrupciones 8259.

Sinopsis:

```
int pol8259()
int npol8259()
```

Descripción:

Cuando el 8259 se programa con `init8259()` o con el BIOS al encender la máquina para utilizarla con DOS, el 8259 queda en "modo de interrupción": cada que un dispositivo lo interrumpe, él a su vez interrumpe al procesador 8086. El 8086 ejecuta entonces la rutina que atiende la interrupción del dispositivo que interrumpió. En modo "polling" no se ejecutan tales rutinas al ocurrir una interrupción de dispositivo, lo que permite al usuario tener más control sobre la ejecución de los procesos, al menos durante la fase de desarrollo de sus aplicaciones.

`Pol8259()` pone al manejador de interrupciones en modo "polling": el 8259 no interrumpe al 8086, éste lee el estado del primero para ver si existe una interrupción por atender. Ver [INTEL81] pag. 7-131. Nótese que `pol8259()` provoca que no se atiendan las interrupciones del reloj, y por lo tanto no habrá cambio de contexto periódicamente. Pueden existir otras implicaciones.

`Npol8259()` quita al manejador de interrupciones del modo "polling": el 8259 interrumpe al 8086 cada que un dispositivo interrumpe.

Ver también

`init8259()`, `readyr()`, `readyt()`.

<code>preceive()</code>

<code>preceive</code>

Obtiene un mensaje largo de un puerto.

Sinopsis:

```
char *preceive(portid)
int portid;
```

Descripción:

`Preceive()` recupera el siguiente mensaje en el puerto `portid`; si todo sale bien, `preceive()` devuelve el mensaje (puede ser la dirección de una estructura); `YSERR` si "portid" es inválido. Ambos procesos, el que envía y el que recibe, deben estar de acuerdo en la tipo de apuntador que envían y reciben.

El proceso que llama es bloqueado si no hay mensajes disponibles, y desbloqueado tan pronto como un mensaje llega. La única manera de ser liberado de la lista de procesos esperando por un mensaje, es que algún otro proceso envíe un mensaje al puerto con `psend()`, o que el puerto sea reinicializado o borrado con `preset()` y `pdelete()` respectivamente.

Ver también

`pcount()`, `pcreate()`, `pdelete()`, `preset()`, `psend()`.

<code>preset()</code>

<code>preset</code>

Reinicializa un puerto de mensajes largos.

Sinopsis:

```
int preset(portid)
int portid;
```

Descripción:

`Preset()` borra todos los mensajes en el puerto "portid", y libera a todos los procesos que esperan enviar o recibir un mensaje del puerto. `Preset()` devuelve `YSERR` si `portid` es inválido.

Si al reiniciar un puerto hay procesos esperando por mensajes en él, estos procesos son puestos en estado `READY` para obtener `YSERR` al volver del llamado a `preceive()`. Si hay mensajes en el puerto, las localidades que ocupan se devuelven a la lista de libres: los mensajes son borrados. Si hay procesos que esperan colocar un mensaje en el puerto, son puestos en estado `READY` para obtener `YSERR` al volver del llamado a `psend()`: como si el puerto nunca hubiera existido. `Pdelete()` realiza la misma función de limpiar y liberar procesos de un puerto tal y como lo hace `preset()`, excepto que `pdelete()` también libera al puerto.

Los efectos de `preset()` son los mismos que provocaría ejecutar `pdelete()` seguido de `pcreate()`, excepto que el puerto no es liberado: la identificación del puerto y la cuenta máxima de mensajes permanece como antes.

Deficiencias

No hay manera de cambiar la cuenta de mensajes máxima al momento de reinicializar el puerto.

Ver también

`pcount()`, `pcreate()`, `pdelete()`, `preceive()`, `psend()`.

```
printf()
```

```
printf
```

```
fprintf()  
kprintf()
```

Servicios para escribir información formateada a dispositivos.

Sinopsis:

```
printf(format [, arg]...)  
char *format;
```

```
fprintf(dev, format [, arg]...)  
int dev;  
char *format;
```

```
kprintf(format [, arg]...)  
char *format;
```

Descripción:

Printf() y kprintf() escriben información formateada hacia el dispositivo CONSOLE (terminal de memoria mapeada). Ellos difieren solamente en que kprintf() escribe directamente a la memoria VIDEO-RAM, mientras que printf() utiliza al manejador de terminal a través de putc(). Fprintf() escribe información formateada hacia el dispositivo "dev", que puede ser un archivo o cualquiera de las terminales.

Cada una de estas funciones convierte, formatea y escribe sus argumentos bajo el control del argumento "format". La manera de interpretar y realizar el formateo indicado en un argumento format es exactamente igual a como lo hace el lenguaje turboC, excepto que las rutinas de XINIX no contemplan la conversión y formateo de datos reales: notación decimal (con punto) o científica. Ver en [TCRfG88] la función "printf()". De esta referencia sólo debe considerarse la especificación de formatos y calificadores que le afecten; mientras que el uso debe conformarse a lo dado en la parte sinopsis. El usuario no debe confundirse con el tipo del primer parámetro de la función fprintf() en la referencia citada.

Ver también

control() para conocer la identificación de los dispositivos XINIX y su tipo, scanf(), y putc().

Deficiencias

Campos grandes (mayores de 80 caracteres) como resultado de la conversión y formateo no se imprimirán completos.

psend()	psend
---------	-------

Envía un mensaje largo a un puerto.

Sinopsis:

```
int psend(portid, message)
int portid;
char *message;
```

Descripción:

Psend suma el apuntador message al puerto portid. Si todo sale bien, psend() devuelve OK; SYSERR si portid es inválido. Nótese que sólo el apuntador al mensaje, y no todo el mensaje es encolado. Psend() puede regresar al procedimiento que le llamó antes de que otro proceso consuma el mensaje.

Si el puerto está lleno al momento de llamar a psend(), el proceso que envía es bloqueado hasta que haya espacio en el puerto para depositar el mensaje.

Ver también

pcount(), pcreate(), pdelete(), preceive(), preset().

putc()	putc
--------	------

```
fputc()
putc()
```

Escriben un carácter a un dispositivo.

Sinopsis:

```
#include "xinixh.h"
```

```
int putc(dev, ch)
int dev;
char ch;
```

Descripción:

Putc() escribe el carácter "ch" en el dispositivo identificado por "dev". Putc() devuelve OK si todo sale bien, SYSERR si "dev" es inválido. Por convención, printf() llama a putc() para escribir en el dispositivo CONSOLE la salida formateada. CONSOLE es el dispositivo cuya identificación es cero. En realidad, putc() es una macro definida en el archivo xinixh.h de la siguiente manera:

```
#define putc xputc
```

... con el único fin de evitar que el usuario llame la rutina `putc()` del lenguaje turboC. `Xputc()` es la rutina de XINIX que escribe caracteres a un dispositivo. Por otra parte, `fputc()` y `putchar()` son macros definidas de la siguiente manera:

```
#define putchar(ch)      xputc(CONSOLE, (ch))
#define fputc(unit,ch)  xputc((unit), (ch))
```

... con el único propósito de asumir la convención del lenguaje C. Se notará que `putchar()` escribe caracteres a la terminal de memoria mapeada.

Ver también

`close()`, `control()`, `getc()`, `open()`, `read()`, `seek()`, `write()`.

<code>putchar()</code>

<code>putchar</code>

Ver `putc`.

<code>puts()</code>

<code>puts</code>

`fputs()`

Escriben una cadena de caracteres a un dispositivo.

Sinopsis:

```
#include "xinix.h"
```

```
puts(s, dev)
char *s;
```

```
fputs(s, dev)
char *s;
int dev;
```

Descripción:

`Puts()` escribe la cadena "s" terminada con un carácter nulo (0), hacia la terminal de memoria mapeada (CONSOLE), y finalmente escribe un carácter NEWLINE ('\n').

`Fputs()` escribe la cadena "s" terminada con un carácter nulo (0), hacia el dispositivo (archivo o terminal) dev; no escribe un NEWLINE.

Ver también

gets(), putc(), printf(), read(), write().

Consideraciones

Puts() suma un NEWLINE, fputs() no; no hay una buena razón para esta convención.

read()	read()
--------	--------

Lee uno o más caracteres de un dispositivo.

Sinopsis:

```
#include "xinxh.h"
```

```
int read(dev, buffer, numchars)
int dev;
char *buffer;
int numchars;
```

Descripción:

Read() lee hasta "numchars" caracteres del dispositivo identificado por "dev". Read() devuelve SYSERR si dev es incorrecto; en caso contrario, el número de caracteres leídos. El número de caracteres leídos depende del tipo de dispositivo. Por ejemplo, cuando se lee de una terminal, cada lectura devuelve, normalmente, una línea. Más si se trata de un archivo se leen "numchars" caracteres, a menos que se encuentre antes el fin de archivo: EOF.

Ver también

close(), control(), getc(), open(), putc(), seek(), write().

readport()	readport
------------	----------

Lee un carácter de un puerto de hardware.

Sinopsis:

```
#include "xinxh.h"
```

```
int readport(port)
int port;
```

Descripción:

Readport() lee un carácter del puerto físico "port". El procesador 8086 tiene un espacio de puertos, como el de memoria,

pero más pequeño, destinado para comunicarse con dispositivos externos, tales como: puertos de comunicación serie, manejadores de disco y diskette, relojes, etcétera. La comunicación se lleva a cabo a través de los registros de éstos dispositivos, los cuales quedan mapeados en el espacio de puertos; así que existe un puerto para cada registro en cada dispositivo. Hay registros para programación, para conocer el estado, para leer o escribir información. `Readport()` y `writport()` son utilizados por el manejador de disco y terminal (las tipo RS232) de XINIX, para controlar tales dispositivos y leer o escribir datos hacia ellos.

`Readport()` devuelve el contenido del puerto al momento de hacer la lectura.

Ver también

`breakt()`, `clrints()`, `disablet()`, `disabler()`, `disablem()`, `disables()`, `enablet()`, `enabler()`, `enablem()`, `enables()`, `erroronr()`, `initport()`, `nobreakt()`, `readyt()`, `writport()`.

<code>readyr()</code>	<code>readyr</code>
-----------------------	---------------------

Revisa si se ha recibido un carácter por un puerto de comunicación serie.

Sinopsis:

```
#include "xinixh.h"
```

```
int readyr(port)
int port;
```

Descripción:

`Readyr()` devuelve TRUE si se ha recibido un carácter por un puerto de comunicación serie; FALSE en caso contrario. `Readyr()` recibe la identificación del puerto según se explica en `breakt()`. `Readyr()` es útil cuando un puerto de comunicación serie se controla en modo polling: sin interrupciones; a continuación un ejemplo:

```
char ch;
int port;
*
*
*   port = 0x3f8;
*   initport(port,....
*
*
*   while ( !readyr(port)) /* espera a que llegue un carácter */
*       ;
*   ch = readport(port); /* lo lee */
*
*
```

Ver también

`breakt()`, `clrints()`, `disablet()`, `disabler()`, `disablem()`,
`disables()`, `enablet()`, `enabler()`, `enablem()`, `enables()`, `erroronr()`,
`initport()`, `nobreakt()`, `readport()`, `readyt()`, `writport()`.

<code>readyt()</code>	<code>readyt</code>
-----------------------	---------------------

Revisa si se ha enviado el último carácter escrito a un puerto de comunicación serie.

Sinopsis:

```
#include "xinixh.h"
```

```
int readyt(port)  
int port;
```

Descripción:

`Readyt()` devuelve `TRUE` si se ha enviado el último carácter escrito a un puerto de comunicación serie; `FALSE` en caso contrario. `Readyt()` recibe la identificación del puerto según se explica en `breakt()`. `Readyt()` es útil cuando un puerto de comunicación serie se controla en modo polling: sin interrupciones; a continuación un ejemplo:

```
char ch;  
int port;  
*  
*  
    port = 0x3f8;  
    initport(port,....  
*  
*  
    while ( !readyt(port)) /* espera a que se envíe el último */  
        ; /* carácter */  
    writport( port, ch); /* envía el actual */  
*  
*
```

Ver también

`breakt()`, `clrints()`, `disablet()`, `disabler()`, `disablem()`,
`disables()`, `enablet()`, `enabler()`, `enablem()`, `enables()`, `erroronr()`,
`initport()`, `nobreakt()`, `readport()`, `readyr()`, `writport()`.

receive()	receive
-----------	---------

Recibe un mensaje del tamaño de un entero.

`recvclr()`

Limpia los mensajes, devolviendo el que exista.

Sinopsis:

```
#include "xinixh.h"
```

```
int receive()
```

```
int recvclr()
```

Descripción:

`Receive()` devuelve el mensaje, del tamaño de un entero, enviado a un proceso con `send()`. Si no hay mensaje alguno, `receive()` bloquea al proceso que le llamó hasta que un mensaje sea recibido.

`Recvclr()` devuelve un mensaje, del tamaño de un entero, si existe; OK en caso contrario. A diferencia de `receive()`, `recvclr()` nunca bloquea al proceso que llama.

Ver también

`send()`, `signal()`, `wait()`.

resume()	resume
----------	--------

Reanuda a un proceso suspendido.

Sinopsis:

```
#include "xinixh.h"
```

```
int resume(pid)
```

```
int pid;
```

Descripción:

`Resume()` reanuda la ejecución del proceso, en estado suspendido, cuya identificación es "pid". Si "pid" no es válida o el proceso no está suspendido, `resume()` devuelve `YSERR`; en caso contrario, devuelve la prioridad con la cual el proceso "pid" reanuda su ejecución. Solamente procesos en estado `PRSUSP` (valor (definido en `xinixh.h`), pueden reanudarse con `resume()`.

Ver también

get_status(), sleep(), suspend(), send(), receive().

scanf()

scanf

fscanf()

Servicios para leer información de dispositivos, formatearla y convertirla a diferentes tipos.

Sinopsis:

```
scanf(format [, pointer]...)
char *format;
```

```
fscanf(dev, format [, pointer]...)
int dev;
char *format;
```

Descripción:

Scanf() lee del dispositivo CONSOLE, fscanf() del dispositivo "dev". Ambos leen caracteres, los interpretan y convierten a otros tipos de datos de acuerdo al formato definido en "format", y almacenan los resultados en la dirección contenida en los argumentos "pointer". Cada uno espera como argumentos, un apuntador a la cadena de caracteres que especifica el formato de control, y un conjunto de argumentos (apuntadores) que indican donde se almacenará la entrada convertida.

La manera de interpretar y realizar el formateo indicado en un argumento format, es exactamente igual a como lo hace el lenguaje TurboC, excepto que las rutinas de XINIX no contemplan la conversión y formateo de datos reales: notación decimal (con punto) o científica. Ver en [TCRfG88] la función "scanf()". De esta referencia sólo debe considerarse la especificación de formatos y calificadores que le afecten; mientras que el uso debe conformarse a lo dado en la parte sinopsis. El usuario no debe confundirse con el tipo del primer parámetro de la función fscanf() en la referencia citada.

Ver también

control() para conocer la identificación de los dispositivos XINIX y su tipo, printf(), getch().

scout()	scout
---------	-------

Devuelve la cuenta asociada a un semáforo.

Sinopsis:

```
scout(sem)
int sem;
```

Descripción:

Scout devuelve la cuenta actual asociada al semáforo "sem". Una cuenta negativa "s", significa que hay "s" procesos esperando en el semáforo; una cuenta positiva "s", significa que a lo más "s" llamados a wait(sem) ocurrirán sin bloquear a un proceso.

Ver también

screate(), sdelete(), signal(), sreset(), wait().

Deficiencias

En esta versión de XINIX SYSERR tiene el valor -1, que corresponde a una cuenta válida de semáforo. Ver pcount().

screate()	screate
-----------	---------

Crea un nuevo semáforo.

Sinopsis:

```
scount(count)
int count;
```

Descripción:

Screate() crea un semáforo e inicializa su cuenta a "count", la cual debe ser mayor o igual a cero. Si todo sale bien, screate() devuelve el valor entero que identifica al semáforo; SYSERR si el semáforo no puede crearse.

Los semáforos son manejados con wait() y signal() para sincronizar procesos. Wait() provoca que la cuenta asociada a un semáforo se decremente, y si resulta negativa, el proceso que ejecuto a wait() es bloqueado. Send() provoca que la cuenta del semáforo se incremente, liberando un proceso bloqueado si existe.

Ver también

scount(), sdelete(), signal(), sreset(), wait().

sdelete()	sdelete
-----------	---------

Borra un semáforo.

Sinopsis:

```
sdelete(sem)
int sem;
```

Descripción:

Sdelete() borra el semáforo "sem" del sistema, y pasa los procesos que esperaban en él al estado PRREAY (definido en el archivo 'xinixh.h'). Sdelete() devuelve SYSERR si "sem" no identifica a un semáforo válido; OK en caso contrario.

Ver también

scount(), screate(), signal(), sreset(), wait().

seek()	seek
--------	------

Servicio para posicionamiento directo en un dispositivo.

Sinopsis:

```
int seek(dev, position)
int dev;
long position;
```

Descripción:

Seek() posiciona la dispositivo "dev" en el carácter "position". Seek() devuelve SYSERR si "dev" es inválido o si no es posible el posicionamiento tal como se especifico.

Seek() no debe usarse con dispositivos tipo terminal. Más bien con archivos.

Ver también

close(), control(), getc(), open(), putc(), read(), write().

send()	send
--------	------

sendf()

Envían un mensaje del tamaño de un entero a un proceso.

Sinopsis:

```
int send(pid, msg)
int pid;
int msg;
```

```
int sendf(pid, msg)
int pid;
int msg;
```

Descripción:

Ambos envían el mensaje "msg" al proceso cuya identificación es "pid". Un proceso puede recibir a lo más un mensaje.

Send() devuelve SYSERR si "pid" es inválida o si el proceso pid ya tiene un mensaje esperando a ser obtenido con receive(). De otra manera, deposita el mensaje y devuelve OK.

Sendf() difiere de send tan sólo en que fuerza el envío del mensaje "msg", aún cuando esto signifique destruir un mensaje existente que no se ha recibido.

Ver también

receive(), signal(), wait().

sendf()	sendf
---------	-------

Fuerza el envío de un mensaje a un proceso. Ver send().

signal()	signal
----------	--------

signaln()

Señalizan un semáforo.

Sinopsis:

```
int signal(sem)
int sem;
```

```
int signaln(sem, count)
int sem;
int count;
```

Descripción:

Ambos señalizan al semáforo "sem" y devuelven OK; SYSERR si "sem" no existe. Signal() incrementa la cuenta del semáforo en 1, y si existen procesos esperando, libera al primero que fue bloqueado. Signaln() incrementa la cuenta del semáforo en "count", y libera hasta "count" procesos esperando si existen. Signaln(sem,n) equivale a ejecutar signal(sem) "n" veces; sin embargo, signaln() es más eficiente.

Ver también

scount(), screate(), sdelete(), sreset(), wait().

signaln()	signaln
-----------	---------

Ver signal()

sleep()	sleep
---------	-------

sleep10()

Suspenden "temporalmente" (duermen) a un proceso.

Sinopsis:

```
int sleep(segundos)
int segundos;

int sleep10(decimas)
int decimas;
```

Descripción:

Ambos provocan que el proceso que les llama sea suspendido por un tiempo especificado, para entonces reanudar su ejecución. Sleep() recibe el tiempo de retardo como un entero que especifica segundos; útil para retardos grandes. Mientras que sleep10(), recibe el tiempo de retardo como un entero que especifica décimas de segundo; para retardos cortos.

Ambos devuelven SYSERR si el argumento es negativo o si XINIX no está manejando el reloj en tiempo real. De otra manera, retardan la ejecución por el tiempo especificado y devuelven OK.

El estado de un proceso que ejecuta a sleep o sleep10 no es PRSUSP, sino PRSLEEP (ambos valores definidos en xinix.h). Un proceso en estado PRSLEEP no puede reanudarse en cualquier momento, XINIX, el manejador de reloj, le despierta después de transcurrido el tiempo que solicitó dormir.

Ver también

suspend().

Consideraciones

El máximo número de segundos es 32767 (alrededor de 546 minutos o 9.1 horas). Sleep() y sleep10() garantizan un retardado mínimo por el tiempo especificado, pero ya que el sistema puede retardar la atención de las interrupciones del reloj, no se puede garantizar un retardo máximo.

sreset()	sreset
----------	--------

Reinicializa la cuenta de un semáforo.

Sinopsis:

```
sreset(sem, count)
int sem;
int count;
```

Descripción:

Sreset() libera los procesos esperando en la cola del semáforo "sem", y actualiza la cuenta de éste a "count". Esto equivale a las operaciones:

```
sdelete(sem);
sem = screate(count);
```

...excepto que sreset() garantiza que la identificación del semáforo no cambia. Sreset() devuelve SYSERR si "sem" no es una identificación de semáforo válida, OK en caso contrario. La cuenta actual de un semáforo no afecta la reinicialización.

Ver también

scount(), screate(), sdelete(), signal(), wait().

stopclk()	stopclk
-----------	---------

Suspende el procesamiento de las interrupciones del reloj.

strtclock()

Reanuda el procesamiento de las interrupciones del reloj.

Sinopsis:

```
stopclk();
strtclock();
```

Descripción:

Stopclk() provoca que las interrupciones de reloj sean acumuladas más no procesadas: los procesos durmientes retardan su despertar y además, no existe cambio de contexto.

Strtclock() reanuda el procesamiento de las interrupciones de reloj considerando las acumuladas.

¿ Qué ocurre si un proceso A solicita dormir durante 5 segundos, un proceso B durante 10, y otro proceso ejecuta stopclk() y strtclk() 7 segundos después ? Bueno, el proceso A despertará 2 segundos más tarde de lo previsto. Pero el proceso B despertará a tiempo por dos razones: los ticks del reloj son acumulados aun cuando no se procesan, y porqué la restauración del reloj ocurre antes de que se cumpla su tiempo para despertar.

Ver también

sleep(), sleep10().

strtclk()

strtclk

Ver stoptck()

suspend()

suspend

Suspende un proceso por tiempo indefinido.

Sinopsis:

```
#include "xinix.h"
```

```
int suspend(pid);  
int pid;
```

Descripción:

Suspend() coloca al proceso "pid" en un estado inactivo, latente, en el que no espera por un evento ni compite por el procesador. Si "pid" es inválida o el estado del proceso a suspender no es PRCURR o PRREADY (ambas definidas en xinix.h), suspend() devuelve SYSERR. De otra manera, devuelve la prioridad del proceso suspendido. Un proceso puede suspenderse a sí mismo, en cuyo caso el llamado devuelve la prioridad con la cual el proceso reanuda su ejecución.

Nótese que un proceso puede poner a otro proceso en estado inactivo (PRUSP), pero sólo puede ponerse a sí mismo a dormir.

Ver también

get_status(), sleep(), send(), receive(), resume().

wait()	wait
--------	------

Bloquea y espera hasta que un semáforo sea señalizado.

Sinopsis:

```
#include "xinixh.h"
```

```
int wait(sem)  
int sem;
```

Descripción:

Wait() decrementa la cuenta asociada al semáforo "sem". Si la cuenta resulta negativa, el proceso que llamo a wait() es bloqueado y encolado en la lista de procesos asociada a "sem". La única manera de que un proceso se libere de la cola de un semáforo, es que otros procesos señalicen al semáforo "sem" con signal() o signaln(), o bien que lo borren o reinicien con sdelete() y sreset() respectivamente. Wait() y signal() son las dos primitivas de sincronización básicas del sistema XINIX.

Wait devuelve SYSYERR si "sem" es inválido. De otra manera, devuelve OK una vez que el proceso es liberado de la cola.

Ver también

scount(), screate(), sdelete(), signal(), sreset().

write()	write
---------	-------

Escribe una secuencia de caracteres desde un buffer a un dispositivo.

Sinopsis:

```
#include "xinixh.h"
```

```
int write(dev, buff, count)  
int dev;  
char *buff;  
int count;
```

Descripción:

Write() escribe "count" caracteres al dispositivo "dev" desde localidades secuenciales del buffer "buff". Write() devuelve SYSERR si "dev" o "count" son inválidos, OK si todo sale bien. La validez de "count" depende del tipo de dispositivo, si se trata de terminales o archivos debe ser un valor igual o mayor a 1; más si se trata de un diskette, "count" representa el número de sector 0 a

719. Write() generalmente regresa cuando el usuario puede ya modificar a buff. Para algunos dispositivos (terminales), esto significa que write() esperará hasta que termine la escritura antes de regresar. En otros (disco: al escribir sectores), los datos son copiados a un buffer del sistema, y write() regresa mientras ellos son transferidos.

Ver también

close(), control(), getc(), open(), putc(), read(), seek().

Deficiencias

Write no puede tener uso exclusivo de un dispositivo, así que la escritura de otros procesos se puede mezclar.

writport()

writport

Escrib un carácter hacia un puerto de hardware.

Sinopsis:

```
#include "xinixh.h"
```

```
int writport(port, ch)
int port;
char ch;
```

Descripción:

Writport() escribe un carácter hacia el puerto físico "port".

Writport no devuelve valor alguno. Ver readport().

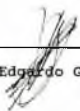
Ver también

```
breakt(), clrints(), disablt(), disabler(), disablem(),
disables(), enablet(), enabler(), enablem(), enables(), erroronr(),
initport(), nobreakt(), readyt().
```

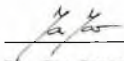

REFERENCIAS

- [COMER84] Douglas Comer.
OPERATING SYSTEM DESIGN. THE XINU APPROACH.
Prentice-Hall Inc., 1984.
- [COMER87] Douglas Comer.
OPERATING SYSTEM DESIGN. VOLUME II.
INTERNETWORKING WITH XINU.
Prentice-Hall Inc., 1987.
- [INTEL81] COMPONENT DATA CATALOG.
Intel, 1981.
- [TANEN87] Andrew S. Tanenbaum.
OPERATING SYSTEMS. Design and Implementation. (MINIX).
Prentice-Hall Inc., 1987
- [TCRfG88] TURBO C REFERENCE GUIDE.
Borland International, 1988.
- [TCUsG88] TURBO C USER'S GUIDE.
Borland International, 1988.
- [TRIBM83] TECHNICAL REFERENCE IBM PC
International Business Machines Corp., 1983.

El jurado designado por la Sección de Computación del Departamento de Ingeniería Eléctrica, del Centro de Investigación y de Estudios Avanzados del I.P.N., aprobó esta tesis el 25 de Septiembre de 1989.



Dr. Manuel Edgardo Guzmán Rentería



Dr. Jan Janeček Hyan



Dr. Hugo César Coyote Estrada

El lector está obligado a devolver este libro
antes del vencimiento de préstamo señalado
por el último sello.

10 ABR. 1990 12 OCT. 1995
5 MAR. 1991 12 OCT. 1995
17 JUN. 1991 3 NOV. 1995
15 OCT. 1991 966 OND. E -
28 MAYO 1992 24 FEB. 1997
12 ABR. 1993 25 ABR. 1997
28 ABR. 1993
- 8 ABR. 1994
28 SET. 1995
- 9 OCT. 1995

1 ENE. 2003
1 OCT. 2003

