Desarrollo de Aplicaciones de Tiempo Real bajo Sistema Operativo QNX

Mag. Guillermo Friedrich - UTN - FRBB

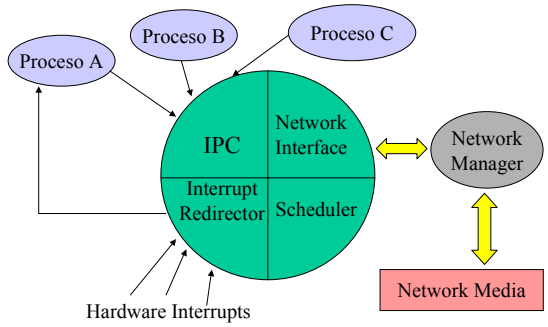
Características principales

- Arquitectura microkernel
- Comunicación entre procesos basada en pasaje de mensajes.

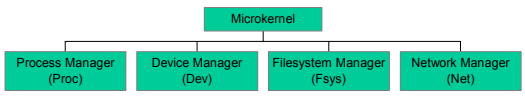
Funciones del microkernel

- Pasaje de mensajes
  - maneja el ruteo de todos los mensajes entre todos los procesos del sistema.
- Planificación de tareas (scheduling)
  - el scheduler es invocado cada vez que un proceso cambia de estado debido a un mensaje o a un interrupción.

Funciones del Microkernel



Arquitectura Microkernel



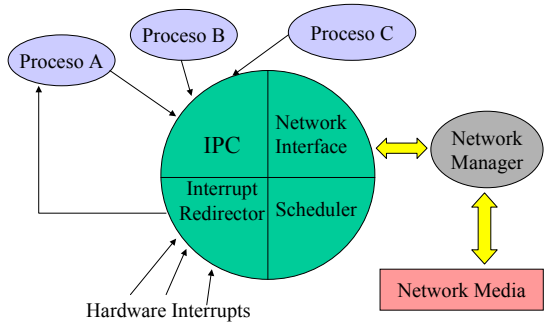
Procesos del Sistema

- Los procesos del sistema prácticamente no se diferencian de los procesos del usuario.
  - Process Manager (Proc)
  - Filesystem Manager (Fsys)
  - Device Manager (Dev)
  - Network Manager (Net)

Device drivers

- Son procesos que aislan al sistema operativo de tener que tratar con los detalles que requiere el soporte de un hardware específico.
  - Luego de completada su inicialización pueden:
    - desaparecer como procesos, convirtiéndose en extensiones del proceso del sistema con el que están asociados.
    - Retienen su identidad como procesos estándar.

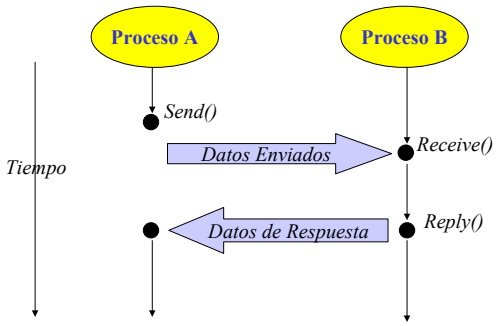
Funciones del Microkernel



IPC *(comunicación entre procesos)*

- Pasaje de Mensajes
  - Bloqueantes
  - Permiten intercambiar datos
  - Comunic. sincrónica entre emisor y receptor
- Proxies
  - Tipo especial de mensajes NO bloqueantes
  - Notifican eventos
  - Asincrónicos
- Señales
  - Requieren un manejador para su atención (si no matan al proceso al cual va dirigida).

IPC mediante Mensajes



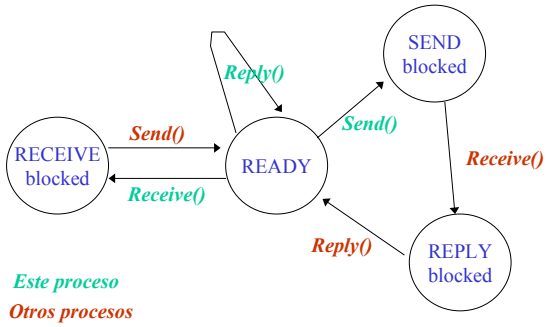
IPC mediante Mensajes

El proceso que hace ***Send()*** queda bloqueado (Send-blocked) hasta que el proceso destinatario efectúe ***Receive()***.

Luego de ello continúa bloqueado (Reply-blocked) hasta que el proceso destinatario ejecute ***Reply()***.

Por otra parte, si un proceso hace ***Receive()***, quedará bloqueado (Receive-blocked) hasta que otro proceso le efectúe un ***Send()***.

Estados de los procesos



Pasaje de Mensajes

Send(pid, smsg, rmsg, smsg_len, rmsg_len)

pid: process ID (PID) del proceso destinatario del msg

smsg: puntero al buffer que contiene el mensaje

rmsg: puntero al buffer donde se espera la respuesta

smsg_len: longitud del mensaje a enviar

rmsg_len: longitud máxima aceptable de la respuesta

Send() retorna 0 (cero) si fue exitosa, o -1 en caso de error (p.ej. si no existe el proceso de destino)

Pasaje de Mensajes

pid = Receive(wpid, msg, msg_len)

wpid: process ID (PID) de quien se espera recibir mensajes (0 = recibir de cualquiera)

msg: puntero al buffer donde se almacenará el mensaje recibido

msg_len: longitud máxima aceptable del mensaje a recibir

Receive() retorna el PID del proceso que envió el mensaje

Pasaje de Mensajes

Reply(pid, reply, reply_len)

pid: process ID (PID) del destinatario de la respuesta (se obtiene de Receive())

reply: puntero al buffer donde se encuentran los datos a enviar como respuesta.

reply_len: longitud del mensaje de respuesta.

Reply() retorna 0 en caso de éxito (-1 en caso de error)

Pasaje de Mensajes (ejemplo)

```
void cliente( pid_t spid )
{
   static struct { int a, b } dato;
   int resul;
   while( 1 )
   {
      printf("Ingrese dos números: ");
      scanf("%d  %d", &dato.a, &dato.b );
      if(Send( spid, &dato, &resul, sizeof(dato), sizeof(resul))==0)
           printf("La suma es : %d\n", resul );
      else
           printf("Error en Send()\n" );
   }
}
```

Pasaje de Mensajes (ejemplo - cont...)

```
void servidor()
{
   static struct { int a, b } dato;
   int resul;
   pid_t cpid;

   while( 1 )
   {
      cpid = Receive( 0, &dato, sizeof(dato) );
      resul = dato.a + dato.b;
      Reply( cpid, &resul, sizeof(resul) );
   }
}
```

Pasaje de Mensajes

Pasaje de mensajes gobernado por Send()

Un proceso servidor está a la espera de requerimientos por parte sus clientes.

El cliente hace Send() para activar al servidor.

Luego que el servidor realiza su trabajo, responde al cliente mediante Reply()

Esta forma, típica del modelo cliente-servidor, es usual cuando el servidor debe realizar un cierto procesamiento y entregarle los resultados al cliente (como el ejemplo anterior, o por ej.: leer datos de un archivo).

Pasaje de Mensajes

Pasaje de mensajes gobernado por Reply()

Un proceso cliente hace Send() para avisarle al servidor que está listo para realizar una tarea, o bien para solicitarle un recurso.

El servidor memoriza el pedido del cliente, hasta que pueda atenderlo.

Cuando el servidor determina que el cliente ya puede trabajar, le hace Reply(). De esta manera el cliente se desbloquea y continúa.

Esta forma es apta cuando el servidor administra el acceso a recursos compartidos, y los clientes le solicitan el permiso para usarlos. (Se verá una implementación del problema de la cena de los filósofos con este estilo).

Pasaje de Mensajes

Ejemplo: Pasaje de mensajes gobernado por Reply()

```
/* El cliente pide permiso p/entrar a una región crítica */
void cliente( pid_t  spid )
{       int entrar=1, salir=0;
        while( 1 )
        {       printf("Fuera de la región crítica\r\n");
                while(Send( spid, &entrar, 0, sizeof(int), 0));
                printf("Dentro de la región crítica\r\n");
                while(Send( spid, &salir, 0, sizeof(int), 0));
        }
```

Pasaje de Mensajes

```
...continuación  /* El servidor lleva una cola de procesos en espera */
void control_de_acceso( )
{       pid_t   p[16], pid;
        int prim=0, ult=0, msg, se_puede=1;
        while( 1 )
        {       pid = Receive( 0, &msg,  sizeof(msg) );
                if ( msg )              /* solicitud de entrada */
                {       if ( se_puede )
                        {       se_puede = 0;
                                Reply( pid, 0, 0 );  /* da el permiso */
                        }else{
```

continúa....

Pasaje de Mensajes

```
...continuación
                if ( msg )              /* solicitud de entrada */
                {       if ( se_puede )
                        {       se_puede = 0;
                                Reply( pid, 0, 0 );  /* da el permiso */
                        }else{
                                /* guarda la solicitud */
                                p[ult] = pid;
                                ult = (++ult) % 16;
                        }
                }else{
                        /* indicación de salida */
```

continúa...

Pasaje de Mensajes

```
...continuación
                }else{
                        /* indicación de salida */
                        Reply( pid, 0, 0 );
                        if ( prim != ult )
                        {
                                /* Hay requerimientos pendientes */
                                Reply( p[prim], 0, 0 );
                                prim = ++prim % 16;
                        }else
                                se_puede = 1;
                }
```

Pasaje de Mensajes

Detalles internos

- *Los datos del mensaje no se copian al microkernel, sino que se mantienen dentro del proceso que hace el Send(), hasta que el proceso destino esté listo para recibirlo. En ese instante se copian dentro del proceso que recibe. Como Send() es bloqueante no hay riesgo de que se alteren los datos.*
- *Los datos de Reply() son copiados del proceso que responde al proceso que está reply-blocked, en una operación atómica. Luego de esto el proceso que recibió los datos se desbloquea.*
- *Reply() no bloquea al proceso que la realiza.*

Pasaje de Mensajes

Detalles internos

• *Si el proceso al que se envía un Send() no está preparado para atender, el proceso que envía queda SEND-blocked.*

• *Si un proceso hace Receive() y no hay mensajes esperando ser atendidos, queda RECEIVE-blocked.*

• *Se puede enviar un mensaje de longitud cero, una respuesta de longitud cero, o ambas.*

```
Send( pid, 0, 0, 0, 0 );
Reply( pid, 0, 0, 0, 0 );
```

Pasaje de Mensajes

Detalles internos

• *Un proceso puede recibir mensajes de varios otros procesos.*

  • *El orden en que los mensajes son recibidos es, normalmente, de acuerdo al orden en que estos fueron enviados.*

  •*El proceso que recibe puede decidir que los mensajes sean recibidos en orden de acuerdo a la prioridad del proceso que los envió.*

Pasaje de Mensajes (ejercicio)

(a) *Implementar un proceso servidor que espere un mensaje consistente en una estructura que contiene dos miembros x e y de tipo int. Este proceso:*

• *al iniciar, debe presentar en pantalla su PID.*

• *por cada mensaje recibido debe imprimir el PID del cliente y los datos recibidos.*

• *debe retornar en el Reply una variable de tipo int conteniendo la suma de los dos números recibidos.*

(b) *Implementar un proceso cliente que le envíe mensajes al proceso servidor, e imprima el número obtenido en cada respuesta.*

Pasaje de Mensajes (ejercicio - cont...)

• *Ejecutar el proceso servidor, y tomar nota de su PID.*

• *Ejecutar una o más instancias del proceso cliente, desde distintas consolas virtuales, y ver la correspondencia entre los mensajes de cada cliente y el servidor.*

Nota: para que se le pueda pasar al cliente el PID del servidor hacer lo siguiente:

```
main( int argc, char *argv[] )
{
    pid_t servidor;
    if( argc > 1 )
        servidor = atoi( argv[1] );
```

Pasaje de Mensajes

Pasaje de mensajes a través de la red QNX

• *Las llamadas al sistema Send(), Receive() y Reply() pueden usarse también para el envío de mensajes a través de la red QNX.*

• *El PID del proceso de destino debe obtenerse mediante la función qnx_vc_attach:*

```
pid_t qnx_vc_attach( nid_t nid, pid_t pid, ... )
```

esta función devuelve un PID para hacer referencia a un proceso que está en el nodo ***nid*** *con un PID local* ***pid****.*

Pasaje de Mensajes

Facilidades avanzadas

• *Recepción condicional de mensajes.*

Cuando no sea factible utilizar Receive() para la atención de mensajes, debido a que es bloqueante, puede utilizarse

```
pid_t Creceive( pid, msg, nbytes)
```

Creceive() retorna:

-1 cuando no hay mensajes en espera,

el PID del proceso que lo envió, en caso de haber mensajes

Pasaje de Mensajes

... Creceive()

Creceive() puede ser útil, por ejemplo, cuando un proceso que básicamente trabaja como cliente puede esperar la llegada de mensajes o eventos -proxys-.

Tener en cuenta que al no ser bloqueante, el uso de Creceive() en lugar de Receive() produce una mayor utilización de la CPU.

Pasaje de Mensajes

... Creceive()

Ejemplo: el siguiente proceso, que trabaja como cliente espera la llegada de un proxy para finalizar.

```
proxy = qnx_proxy_attach(0,0,0,-1);
.....
while( 1 )
{       if ( Creceive( proxy, 0, 0 ) == proxy )
                exit(0);
        .....
        Send( ..... );
```

Pasaje de Mensajes *(facilidades avanzadas)*

• *Leer parte de un mensaje recibido (Readmsg()).*

La función Readmsg() le permite al receptor de un mensaje leer directamente del buffer en el área de memoria del proceso que efectuó el Send(), sin tener que consumir memoria para copiar todo el mensaje al área de memoria del receptor.

Esto es muy útil cuando no se puede saber de antemano la longitud de un mensaje, y la misma viene indicada en el encabezado del mensaje.

Pasaje de Mensajes *(facilidades avanzadas)*

• *Ejemplo del uso de Readmsg()*

```
void cliente( int spid )
{
        static struct { int longitud; char s[40]; } msg;
        int i, resp;
        strcpy( msg. s, "HOLA QUE TAL" );
        msg.longitud = strlen(msg. s);
        Send(spid, &msg, &resp, sizeof(msg), sizeof(resp));
}
```

Pasaje de Mensajes *(facilidades avanzadas)*

• *Ejemplo del uso de Readmsg()*

```
void servidor()
{   pid_t pid;
    int n, i;
    char c;
    pid = Receive( 0, &n, sizeof(n) );
    for(i=0; i < n; i++)  {
      Readmsg( pid, sizeof(n) + i * sizeof(c), &c, sizeof( c ) );
      putchar( c );
    }
    Reply( pid, 0, 0 );
}
```

Pasaje de Mensajes *(facilidades avanzadas)*

• *Ejemplo del uso de Readmsg()*

El servidor mediante Receive() lee en encabezado del mensaje (en este ejemplo este sólo contiene la longitud de los datos).

A continuación, mediante Readmsg() va obteniendo byte a byte los datos del mensaje y lo envía a la consola.

El proceso cliente permanece Reply-bloqued desde el Receive() hasta el Reply(), por lo que entre tanto no hay peligro de que los datos puedan ser alterados.

Pasaje de Mensajes *(facilidades avanzadas)*

• ***Ejercicio***

Desarrollar un proceso servidor que, a pedido de los clientes, realiza operaciones aritméticas sobre los datos que se le envían, y retorna el resultado en el Reply().

Las operaciones posibles son: SUMA y PRODUCTO

Los datos son de tipo float, pero la cantidad de ellos es variable.

En el comienzo del mensaje hay dos int, el primero de ellos para discriminar la operación a realizar y el segundo para indicar la cantidad de datos a operar.

Pasaje de Mensajes *(facilidades avanzadas)*

• ***Ejercicio (continuación)***

Usando Receive() leer el encabezado del mensaje y en función de la cantidad de datos que contiene el mismo, ir obteniendo cada uno de ellos mediante Readmsg().

Finalmente, en Reply() devolver el resultado en una variable de tipo float.

Sugerencia: el cliente podría usar una struct correspondiente al mensaje completo (con máx. de 20 datos), mientras que el servidor podría usar una struct para el encabezado del mensaje, y luego obtener cada dato.

Pasaje de Mensajes

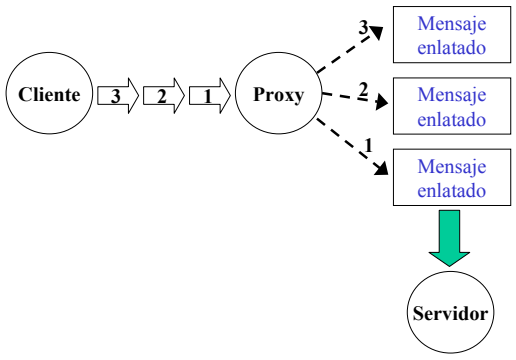
Facilidades avanzadas

• *Mensajes multiparte.*

Proxys

- *Un proxy es un mensaje NO bloqueante.*
- *El emisor no interactua sincrónicamente con el receptor.*
- *Se utilizan para la notificación de eventos.*
- *Su contenido es fijo (por ej.: el código del evento).*
- *Se usa qnx_proxy_attach() para asociar un proxy al proceso que lo recibirá.*
- *Se usa Trigger(proxy) para dispararlo.*
- *Se usa qnx_proxy_detach() para eliminar un proxy.*

Proxys



Proxys

Creación de un proxy:

pid_t ***qnx_proxy_attach****(pid, data, nbytes, priority)*

pid: PID del proceso al cual se le asocia el proxy
(0 : proceso actual)

data: puntero al mensaje contenido en el proxy

nbytes: longitud del mensaje

priority: prioridad (-1 : prioridad del proceso actual).

Retorna: el PID del proxy creado

Proxys

Disparo de un proxy:

*pid_t **Trigger**(proxy)*

proxy: PID del proxy que se desea disparar.

Retorna: el PID del proceso que posee el proxy (-1: error)

Proxys

Eliminación de un proxy:

*int **qnx_proxy_detach**(proxy)*

proxy: PID del proxy que se desea eliminar.

Retorna: 0 si hubo éxito (-1: error)

Errores: EPERM no es propietario del proxy a eliminar

ESRCH el proxy a eliminar no existe.

Proxys (ejemplo)

```
void main()
{       pid_t proxy;
        proxy = qnx_proxy_attach(0,0,0,-1);
        if( proxy != -1 )
        {       if( fork() )
                {       server();   /* El padre posee el proxy*/
                        qnx_proxy_detach( proxy );
                }else
                        client( proxy );   /* El hijo lo dispara */
        }
}
```

Proxys (ejemplo - cont...)

```
void client( pid_t proxy )
{
        int i;
        for( i=0; i<10; i++)
        {
                printf( "Trigger %d\n", proxy );
                Trigger( proxy );
        }
}
```

Proxys (ejemplo - cont...)

```
void server()
{       int i;
        pid_t pid;
        unsigned msg;
        for( i=0; i<10; i++)
        {
                pid = Receive( 0, &msg, sizeof(msg) );
                printf( "Received from: %d\n", pid );
        }
}
```

Proxys

Cuando una rutina de atención de interrupción debe notificarle un evento a una tarea (por ej.: que hay un dato disponible), la forma natural de hacerlo es mediante el disparo de un proxy.

En este caso la forma de ordenar el disparo es que la rutina de interrupción finalice retornando el PID del proxy que desea disparar.

Si retorna 0, no se efectúa ningún disparo.

(Más detalles se verán en el tema Interrupciones).

Señales (signals)

- *Son otra forma tradicional de comunicación asincrónica*
- *Una señal se parece a una interrupción por software.*
- *El proceso que recibe una señal debe tener un handler que determine la acción a tomar. En caso contrario, la acción por defecto es normalmente terminar el proceso.*
- *Desde el shell se usan los comandos kill o slay para enviar señales a un proceso*
- *Desde un programa se usan las funciones kill() o raise() para enviar una señal.*

Señales (ejemplo)

```
#include <signal.h>
volatile sig_atomic_t signal_count = 0;     /* es un int (typedef) */

void MyHandler( int signal_number )
{
        ++signal_count;
}

void main()
{
   signal( SIGFPE, MyHandler );  /* establece el handler */
   while( signal_count < 5 );          /* espera 5 veces */
   signal( SIGFPE, SIG_DFL );     /* restaura la acción por defecto */
   getch();       /* si se recibe otra vez la señal, el proceso se aborta */
}
```

Señales (signals)

Los parámetros de la función signal( ) son:

- *(int)* : Nº de señal
- *void (*func) (int)* : puntero al "handler" que atiende a la señal indicada.

El segundo parámetro determina la acción a tomar ante la llegada de una señal. Las posibilidades predefinidas son:

SIG_DFL *seguir la acción por defecto*

SIG_IGN *ignorar la señal*

Señales (ejemplo)

Para enviarle una señal al proceso del ejemplo anterior, se puede hacer:

kill –SIGFPE <pid>

donde: <pid> es el PID del destinatario.

O bien:

slay – s SIGFPE <proc_name>

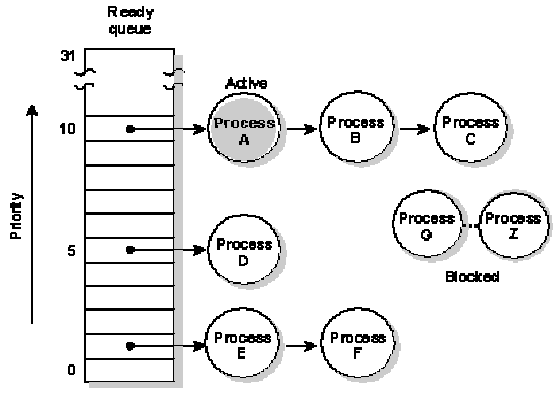
donde: proc_name es el nombre del programa.

Se pueden ver otras opciones de estos comandos haciendo:

use kill

use slay

Planificación de procesos (scheduling)



Planificación de procesos (scheduling)

El microkernel toma decisiones respecto a la ejecución de los procesos cuando:

- *Un proceso se desbloquea.*
- *Expira el "quantum" de tiempo de un proceso.*
- *Un proceso en ejecución es desalojado.*

Cada proceso tiene asignada una prioridad, que puede ir de 0 (la menor) a 31 (la mayor).

Planificación de procesos (scheduling)

- *Se ejecuta el proceso no bloqueado de mayor prioridad*
- *Si hay más de un proceso en condiciones de ejecutar, el microkernel tiene las siguientes opciones, de acuerdo a como haya sido establecida la forma de planificar cada proceso:*
  - *FIFO*
  - *Round-robin*
  - *Planificación adaptiva*

Planificación de procesos (scheduling)

- ***Planificación FIFO**: un proceso ejecuta hasta que:*
  - *Voluntariamente entrega el control (p.ej.: realiza una llamada al sistema).*
  - *Es desalojado por un proceso de mayor prioridad.*
- ***Planificación round-robin**: un proceso ejecuta hasta que:*
  - *Voluntariamente entrega el control*
  - *Es desalojado por una tarea de mayor prioridad*
  - *Consume su "quantum" de tiempo.*

Planificación de procesos (scheduling)

- ***Planificación Adaptiva**:*
  - *Si un proceso consume su slot de tiempo sin haberse bloqueado reduce su prioridad en un nivel. Cuando el proceso se bloquea, retorna a su prioridad original.*
  - *Solamente se reduce en un nivel la prioridad original.*
  - *Es el tipo de planificación por defecto de los procesos largados desde el shell.*

Planificación de procesos (scheduling)

- Consultar el nivel de prioridad de un proceso:

```
# include < sys/sched.h >

int getprio( int pid );
```

pid : PID del proceso cuya prioridad se quiere averiguar.
0 → averiguar la prioridad propia.

Retorna : nivel de prioridad (0 - 31)

Planificación de procesos (scheduling)

- Establecer el nivel de prioridad de un proceso:

```
# include < sys/sched.h >

int setprio( int pid,  int prio );
```

pid : PID del proceso cuya prioridad se quiere establecer.
0 → establecer la prioridad propia.

prio : nivel de prioridad deseado (0 - 31)

Planificación de procesos (scheduling)

- Consultar el tipo de planificación de un proceso:

```
# include < sys/sched.h >
int getscheduler( int pid );
```

pid : PID del proceso cuya planificación se quiere conocer.
0 → averiguar el tipo de planificación propio.

Retorna :	*SCHED_FIFO*	*(0)*	*FIFO*
	SCHED_RR	*(1)*	*Round Robin*
	SCHED_FAIR	*(2)*	*Adaptiva*

Planificación de procesos (scheduling)

• Establecer prioridad y tipo de planificación de un proceso:

 # include < sys/sched.h >

int setscheduler(int pid, int planif, int prio);

pid : PID del proceso que se desea configurar (0 = propio)

planif : SCHED_FIFO, SCHED_RR o SCHED_FAIR

prio : nivel de prioridad (0 - 31)

Retorna : 0 (no error) o -1 (error)

Planificación de procesos (scheduling)

• Ejemplo:

```
# include < stdio.h >
# include < sys/sched.h >
void main( )
{       printf ( "Prioridad = %d\r\n", getprio(0) );
        printf ( "Scheduler = %d\r\n", getscheduler(0) );
        setscheduler( 0, SCHED_FIFO, getprio(0)+1 );
        printf ( "Prioridad = %d\r\n", getprio(0) );
        printf ( "Scheduler = %d\r\n", getscheduler(0) );
}
```

El Administrador de Procesos

- *Es el único proceso que comparte el mismo espacio de direcciones que el microkernel.*
- *A pesar de ello, trabaja como un proceso más, es decir:*
  - *Su ejecución es planificada por el scheduler.*
  - *Se comunica por mensajes con los demás procesos.*
- *Es responsable de la creación de nuevos procesos, y del manejo de los recursos básicos asociados a un proceso.*
- *Si, por ejemplo, un proceso desea crear otro proceso, debe enviarle un mensaje a Administrador de Procesos, con los detalles correspondientes. Inclusive, se puede enviar tal solicitud al Administrador de Procesos de otros nodo.*

El Administrador de Procesos

- *QNX dispone de las siguientes funciones para la creación de nuevos procesos:*
- *fork()* *crea un nuevo proceso que es una imagen exacta del proceso que la invocó.*
  El nuevo proceso comparte el mismo código del creador, y recibe una copia de todos los datos de aquel.
  El proceso que la invoca (padre) recibe como retorno el PID del nuevo proceso (hijo).

El Administrador de Procesos (cont.)

- *fork()* *El proceso recién creado recibe 0.*
  Si no se pudo crear el proceso hijo, el padre recibe como retorno -1, y en errno queda el código de error.

El Administrador de Procesos (cont.)

• *Ejemplo: creación de un proceso usando la función fork()*

```
void main()
{
        pid_t   pid_hijo;
        if ( (pid_hijo = fork()) == 0 )
                hijo( );   /* por acá sale el nuevo proceso */
        else
                padre(  pid_hijo );  /* por acá sale el creador*/
}
```

El Administrador de Procesos (cont.)

```
void hijo( )
{       /* este proceso trabaja como servidor */
        pid_t pid;
        int msg;
        do {    pid = Receive(0, &msg, sizeof(msg) );
                printf("%d \r\n", msg++);
                Reply(pid, &msg, sizeof(msg));
        }while( msg < 10 );
        exit(0);   /* Finaliza */
}
```

El Administrador de Procesos (cont.)

```
void padre( pid_t  spid)
{       /* este proceso trabaja como cliente */
        int i = 0, status;
        while(  i<10 )
         {      Send(spid, &i, &i, sizeof(i), sizeof(i));
        }
        do {   /* espera la finalización del hijo */
                waitpid( spid, &status, 0 );
        } while( WIFEXITED( status ) == 0 );
}
```

El Administrador de Procesos (cont.)

- *Un proceso hijo que finaliza puede quedar en estado "zombie", si finaliza antes que el padre y aquel no tiene previsto esperar su finalización (p. ej.: usando waitpid( )).*

- *Si el padre termina, los procesos "zombies" son eliminados de la tabla de procesos.*

- *Un proceso "zombie" ya no existe como proceso, sino que permanece registrado en la tabla de procesos, con el PID que tenía.*

Si se hace kill al PID de un proceso zombie, la respuesta es que tal proceso no existe (a pesar de verlo con ps o sin)

El Administrador de Procesos (cont.)

- *La función waitpid() suspende la ejecución del que la invoca, hasta tener información acerca del proceso referido.*

*pid_t waitpid(pid_t pid, int * status, int options)*

pid = PID del proceso que se espera (-1: cualquier hijo)

En status retorna un código que se puede evaluar mediante alguna de las siguientes macros:

WIFEXITED(status)	*retorna True si el proceso terminó*
WIFSIGNALED(status)	*retorna True si el proceso terminó por una señal no esperada.*

El Administrador de Procesos (cont.)

WIFEXITED(status)	*retorna True si el proceso terminó*
WEXITSTATUS(status)	*retorna el valor de salida dado por el proceso que finalizó.*
WIFSIGNALED(status)	*retorna True si el proceso terminó por una señal no esperada.*
WTERMSIG(status)	*retorna el número de la señal que produjo la finalización del proceso*

El Administrador de Procesos (cont.)

- ***Ejercicio***

Desarrollar un programa que crea un proceso hijo (fork())

Luego el padre ordena finalizar al hijo de alguna de las siguientes maneras:

  - *Enviándole un mensaje.*
  - *Enviándole un proxy.*
  - *Enviándole una señal.*

El padre debe esperar la finalización del hijo y evaluar en que condiciones finalizó (WIFEXITED, WIFSIGNALED, etc)

El Administrador de Procesos (cont.)

• ***Asignación de nombres simbólicos a los procesos.***

Debido a que no se puede predecir el PID que tendrá un determinado proceso (normalmente servidor), QNX da la posibilidad de registrar un nombre simbólico a cada proceso. Para ello se utiliza la función:

```
int  qnx_name_attach( nid_t nid,  char * name )
```

nid : número de nodo (0 = nodo local)

name: cadena de char correspondiente al nombre del proc.

Retorna en un int el identificador que luego se usa cuando se desea des-registrar el nombre.

Se debe hacer: `# include <sys/name.h>`

El Administrador de Procesos (cont.)

... continuación

Los nombres pueden ser locales a un nodo, por ejemplo:

SumServer

miniacco/datalogging

miniacco/display

Los nombres también pueden ser globales a toda la red, en cuyo caso el nombre va precedido de '/', por ejemplo:

/miniacco/datalogging

El Administrador de Procesos (cont.)

• ***Asignación de nombres simbólicos a los procesos.***

```
void main( )
{       int     name_id, x[2], y;
        pid_t  pid;
        name_id = qnx_name_attach( 0,  "SumServer");
        if ( name_id == -1 )
        {       perror( "qnx_name_attach");
                exit(1);
        }
```

continúa ...

El Administrador de Procesos (cont.)

... continuación

```
        do                      /* bloque de trabajo del proceso*/
        {       pid = Receive( 0,  x, 2 * sizeof(int) );
                y = x[0] + x[1];
                Reply( pid, &y, sizeof(int) );
        } while( y > 0 );
        /* antes de finalizar des-registra el nombre */
        qnx_name_detach( 0, name_id );
        exit(0);
}
```

El Administrador de Procesos (cont.)

• ***Localización de un proceso por su nombre***

Se utiliza la función:

```
pid_t  qnx_name_locate( nid_t nid,
                        char *name,
                        unsigned size,
                        unsigned ncopies );
```

nid : número de nodo (0 = nodo local)

name : nombre del proceso que se desea localizar

size y ncopies se utilizan cuando se busca un proceso en la red; para buscar en el nodo local van en cero.

El Administrador de Procesos (cont.)

• ***Localización de un proceso por su nombre***

```
void main( )
{       int     x[2] = {15, 22},  y;
        pid_t  pid;
        pid = qnx_name_locate( 0, "SumServer", 0, NULL );
        if( pid == -1 )
        {       perror("qnx_name_locate");
                exit(1);
        }
        Send( pid,  x,  &y,  2 * sizeof(int),  sizeof(int) );
```

El Administrador de Procesos (cont.)

• ***Ejercicio: Uso de nombres para los procesos***

Implementar un proceso, y asignarle el nombre "FileServer"

Este proceso deberá trabajar como servidor, y espera un mensaje con la siguiente estructura:

```
typedef  struct
{       char   filename[40];   /* nombre de archivo */
        int    nro_renglon;   /* renglón solicitado */
}req_t;
```

El servidor debe abrir el archivo solicitado, leer el renglón pedido, retornar el mismo en el Reply y cerrar el archivo.

El Administrador de Procesos (cont.)

... continuación

Estructura para el Reply( )

```
typedef  struct
{       int    status;
        char   txt[80];
} resp_t;
```

Si status = -1 hubo error (p.ej.: el archivo no existe)

status >= 0 cantidad de caracteres en txt[].

El Administrador de Procesos (cont.)

... continuación

Por otra parte se deberá implementar otro programa (el que podrá ejecutarse en varias instancias), y que a pedido del usuario le solicite el envío de diferentes renglones de texto a "FileServer" y lo imprima en la pantalla.

Crear además dos o más archivos de texto diferentes, para ser usados por el servidor.

Temporizadores

• ***Facilidades simples para el temporizado***

Desde el shell: *sleep <nsegundos>*

suspende la ejecución durante nsegundos

Desde un proceso:

```
# include <unistd.h>
unsigned  sleep( unsigned segundos);
```

Suspende la ejecución del proceso durante la cantidad de segundos requerida.
Retorna -1 si hubo error (p.ej.: no hay timers disponibles)

Temporizadores

• ***Facilidades simples para el temporizado*** *(continuación)*

También desde un proceso:

```
# include <i86.h>
unsigned  delay( unsigned miliseg);
```

Suspende la ejecución del proceso durante la cantidad de milisegundos de tiempo real requerida.

Retorna: *0* *si completó el tiempo requerido*
-1 *si hubo error (p.ej.: no hay timers disponibles)*
>0 *si fue interrumpido por una señal; devuelve los milisegundos que faltan para completar.*

Temporizadores

• ***Facilidades simples para el temporizado*** *(continuación)*

También:

```
# include <unistd.h>
unsigned  alarm( unsigned  nseg);
```

Si nseg > 0 *se prepara el envío de SIGALRM para dentro de nseg segundos*

Si nseg = 0 *se suspende cualquier alarma activada.*

Retorna la cantidad de segundos que faltan hasta que sea enviada una señal SIGALRM al proceso que la invoca.
Retorna 0 si no hay alarmas previamente requeridas.
Retorna -1 si hubo error (p.ej.: no hay timers disponibles)

Temporizadores

• ***Ejemplo de alarm() y pause()***

```
# include <unistd.h>

void main()
{       unsigned  tiempo_restante;

        alarm( 10 );
        sleep( 5 );
        tiempo_restante = alarm(0);
        alarm( tiempo_restante );
        pause( );      /* el proceso aborta por que la señal  */
}                      /* no tiene un handler para esperarla */
```

Temporizadores

• ***Ejemplo de alarm() y pause()*** *(continuación)*

Para que la señala SIGALRM no finalice al proceso se debe contar con un handler.

```
# include <signal.h>
#include <unistd.h>

void alarm_handler( int sig_num )
{
        printf( "Handler, señal recibida: %d\r\n", sig_num);
}
```

(continúa)

Temporizadores

• ***Ejemplo de alarm() y pause()*** *(continuación)*

```
void main()
{       unsigned  tiempo_restante;

        signal( SIGALRM, alarm_handler );
        alarm( 10 );
        sleep( 5 );
        tiempo_restante = alarm(0);
        alarm( tiempo_restante );
        pause( );      /* el proceso se suspende hasta que
                          llega una señal */
        signal( SIGALRM,  SIG_DFL);
```

Temporizadores

• ***Manejo avanzado del tiempo: generación de eventos mediante temporizadores.***

Hay seis acciones posibles:

- *Crear un timer, y asociarle el proxy que debe disparar a su vencimiento.*
- *Poner en marcha el timer, fijándole el tiempo hasta su vencimiento y/o su período.*
- *Detener el timer, si no se desea que llegue a su vencimiento.*
- *Consultar cuanto falta para el vencimiento.*
- *Atender la llegada del evento, recibiendo o bien la señal o bien el mensaje del proxy disparado por el timer.*
- *Eliminar un timer que ya no se necesita.*

Temporizadores

• Creación de un timer: *timer_create( )*

```
# include <signal.h>
# include <time.h>

timer_t   timer_create (  clock_id  clk_id,
                          struct sigevent  *evp );
```

clk_id: el único valor soportado es CLOCK_REALTIME

Si evp → sigev_signo > 0 *Nº de señal a disparar cuando vence el timer.*

evp → sigev_signo < 0 *- PID del proxy a disparar.*

Temporizadores

• Ejemplo de creación de un timer (usando señales):

```
void signal_handler ( int signum )
{
}

void main( )
{       struct sigevent   ev;
        timer_t   t;
        /* Como ejemplo en el timeout dispara la señal SIGALRM */
        ev.sigev_signo = SIGALRM;
        t =  timer_create( CLOCK_REALTIME,  & ev );
        /* Instala un handler para esperar la señal SIGALRM */
        signal( SIGALRM, signal_handler );
```

Temporizadores

• Ejemplo de creación de un timer (usando proxy):

```
void main( )
{
        struct sigevent   ev;
        timer_t   t;
        pid_t   proxy;

        proxy = qnx_proxy_attach( 0, 0, 0, -1 );

        ev.sigev_signo = – proxy;

        t =  timer_create( CLOCK_REALTIME,  & ev );
```

Temporizadores

• Poner en marcha (armar) un timer: *timer_settime()*

```
# include <time.h>

int   timer_settime( timer_t   timer_id,    int  flags,
                     struct itimerspec * value,
                     struct itimerspec * ovalue  );
```

timer_id: ID del timer obtenido de timer_create()
flags: si es 0 ⇒ el tiempo es relativo.
value: valores de tiempo a establecer.
ovalue: valores de tiempo que había antes de settime()

Temporizadores

• Poner en marcha (armar) un timer: *(continuación)*

La estructura itimerspec está definida en time.h

```
struct itimerspec
{       struct                  /* tiempo al 1º disparo */
        {       long  tv_sec;     /* segundos */
                long  tv_nsec;   /* nanosegundos */
        }it_value;
        struct                  /* período p/ posteriores disparos*/
        {       long  tv_sec;     /* segundos */
                long  tv_nsec;   /* nanosegundos */
        }it_interval;
};
```

Temporizadores

• Ejemplo: armar un timer: *(continúa del ejemplo anterior)*

```
void start_timer( timer_t  tid, long periodo_seg )
{
        struct itimerspec  tiempo;

        /* El tiempo al 1º disparo y a los posteriores es igual */
        tiempo.it_value.tv_sec = periodo_seg;
        tiempo.it_value.tv_nsec = 0L;
        tiempo.it_interval.tv_sec = periodo_seg;
        tiempo.it_interval.tv_nsec = 0L;

        timer_settime( tid, 0, &tiempo, NULL );
}
```

Temporizadores

• Detener un timer:

La forma de detener un timer es mediante timer_settime(), fijándole el primer tiempo y el período en cero.
Por ejemplo:

```
        void  stop_timer( timer_t  tid )
        {       struct itimerspec  tiempo;

                tiempo.it_value.tv_sec = 0L;
                tiempo.it_value.tv_nsec = 0L;
                tiempo.it_interval.tv_sec = 0L;
                tiempo.it_interval.tv_nsec = 0L;
                timer_settime( tid, 0, &tiempo, NULL );
        }
```

Temporizadores

• Eliminar un timer:

La forma de eliminar un timer es mediante timer_delete().

Es importante dar de baja un timer cuando no se lo requiera más, a fin de que dicho recurso esté disponible para otro proceso que lo pueda requerir.

```
int  timer_delete( timer_t  tid );
```

tid : identificador del timer obtenido de timer_create( )

Retorna: 0 si pudo eliminar el timer
-1 si hubo un error (p.ej.: tid incorrecto)

Temporizadores

• Consultar cuanto falta para el vencimiento de un timer:

*int timer_gettime(timer_t tid, struct itimerspec * tptr);*

tid : identificador del timer obtenido de timer_create( )

tptr: puntero a una variable tipo struct itimerspec, en la cual luego de llamarse a la función están disponibles:
tptr → it_value : tiempo restante al vencimiento
tptr → it_interval: valor repetitivo de recarga

Retorna: 0 si no hubo errores
-1 si hubo un error (p.ej.: tid incorrecto)

Temporizadores

• Consultar / modificar la resolución de los temporizadores

Desde el shell: *ticksize* *muestra el valor actual del período entre timer ticks*

ticksize N *establece un nuevo valor para la frecuencia de ticks*

N puede ser: .5, 1, 2, 5, 10, 25, 50 o 55 miliseg

Un valor pequeño aumenta la precisión de los timers, pero a costa de una mayor carga del sistema.

Temporizadores

• Consultar / modificar la resolución de los temporizadores

Desde un proceso:

*int clock_setres(CLOCK_REALTIME, struct timespec *res);*

res → tv_nsec : resolución deseada, medida en nanoseg.

Valores válidos son: .5, 1, 2, 5, 10, 25, 50 o 55 miliseg.

Ejemplo:
```
# define MSEC (1000000)
struct timespec res;
res . tv_nsec = 10 * MSEC;
clock_setres( CLOCK_REALTIME, &res );
```

Interrupciones

• Conceptos

Los manejadores de interrupciones (interrupt handlers) atienden los requerimientos de interrupción generados por el hardware de la computadora.

Si bien están fisicamente empaquetados junto con el resto del proceso al cual están asociados, corren asincronica-mente a dicho proceso.

Un interrupt handler es basicamente una función en C, asociada a una interrupción dada.

Interrupciones

• Manejadores de interrupción

No se ejecutan directamente desde la interrupción, sino que son llamados desde el kernel mediante un "far call"

Ejecutan en el contexto del proceso al cual están asociados, por lo tanto acceden a sus variables globales.

Pueden ser interrumpidos por otra interrupción de mayor nivel.

No deben interactuar directamente con el 8259.

Deben ser lo más breves que sea posible.

Interrupciones

• Manejadores de interrupción (*cont.*).

No deben llamar al kernel (p.ej.: imprimir o enviar mensajes)

La forma de comunicarse con un proceso es mediante un proxy:

- Retornando un número, se dispara el proxy cuyo PID es dicho número.

Ej.: return pid_pxy;

- Retornando 0 no se dispara ningún proxy.

Varios procesos pueden utilizar la misma interrupción.

Interrupciones

• Manejadores de interrupción *(Ejemplo).*

```
pid_t proxy;
volatile unsigned counter;

#pragma off (check_stack);

pid_t far handler( )
{
        if ( (++counter % 100 ) == 0 )
                return proxy;   /* dispara cada 100 ticks  */
        return 0;
}
#pragma on (check_stack);
```

Interrupciones

• Manejadores de interrupción *(Ejemplo - cont.)*

```
void main( )
{
        int id, i;

        /* Asocia un proxy que será disparado desde la
           interrupción */

        if ( (proxy=qnx_proxy_attach(0,0,0,0)) == -1 )
        {
                perror( "proxy attach");
                return;
        }
```

Interrupciones

•*(Ejemplo - cont.)*

```
        /* Asocia el handler a la interrupción del timer */
        if ( (id=qnx_hint_attach( 0, &handler,
                                  FP_SEG(&counter))) == -1)
        {
                perror("hint attach");
                return;
        }
        /* Espera 10 disparos */
        for (i=0; i<10; i++ )
        {       Receive( proxy, 0, 0 );
                printf ( "%d ticks\r\n", i * 100 );
        }
        qnx_hint_detach( id );   /* desactiva el handler */
```

Interrupciones

```
# include < sys/irqinfo.h >

int qnx_hint_attach( unsigned int num,
                     pid_t (* handler) (void),
                     unsigned ds );
```

num : número de IRQ (-1: timer fijo de 50 mseg)

handler: puntero a la función del handler de interrupción.

ds: valor del "data segment" de las variables globales accedidas por el handler.

Retorna: identificador del handler (usado para "detach")

Interrupciones

• Desinstalación de un handler de interrupción

Se utiliza la función qnx_hint_detach( )

```
# include < sys/irqinfo.h >

int  qnx_hint_detach( int iid );
```

iid : identificador del manejador de interrupción; es el valor obtenido de qnx_hint_attach( ) al instalarlo.

Retorna: 0 si se pudo desisntalar.
-1 si hubo algún error (p.ej.: identif. incorrecto)

Interrupciones

• Opciones para el compilador

Para compilar un programa que utiliza interrupciones se deben pasar los siguientes parámetros:

```
cc  –Wc  –s  –zu
```

La instalación de un handler de interrupción requiere del nivel de privilegio de superusuario, por lo tanto:

- El programa deberá tener como propietario a root
- Debera setearse: chmod a+s <programa>
a fin de que cualquiera lo pueda ejecutar con el nivel de privilegio de root.

Entrada / Salida

• Lectura de un puerto de 8 bits

La función para leer un byte de un puerto es inp( )

unsigned int inp *(int puerto);*

Ejemplo: leer el registro de recepción de COM2

```
# include <conio.h>

void main()
{
        printf ( "%c", inp( 0x2f8 ) );
}
```

Entrada / Salida

• Escritura en un puerto de 8 bits

La función para escribir un byte en un puerto es outp( )

unsigned int outp *(int puerto, int dato);*

Ejemplo: transmitir un dato por el puerto serie COM2

```
# include <conio.h>

void main()
{
        outp( 0x2f8,  getche( ) );
}
```

Entrada / Salida

• Lectura de un puerto de 16 bits

La función para leer un puerto de 16 bits es inpw( )

unsigned int inpw *(int puerto);*

• Escritura en un puerto de 16 bits

La función para escribir en un puerto de 16 bits es outpw( )

unsigned int outpw *(int puerto, int dato);*

Entrada / Salida

• Opciones para el compilador

Debido a que las operaciones de lectura y escritura de puertos implican la ejecución de instrucciones "privilegiadas", el proceso que las realiza debe tener un nivel de privilegio 1.

Debe ser compilado con la siguiente opción:

cc –T1

Asimsimo, deberá ejecutarse desde el superusuario.

Entrada / Salida con Interrupciones

• Ejemplo: Transmisión y Recepción de datos a través del puerto serie COM2

• Descripción:

El handler de interrupción analiza la causa de la interrupción, y si se debe a que se ha recibido un nuevo dato, lo guarda y recibido envía un proxy al proceso que luego lo imprimirá.

Por otra parte, los datos ingresados desde el teclado son enviados por el puerto serie.

Entrada / Salida con Interrupciones (cont.)

```
# include <stdio.h>
# include <stdlib.h>
# include <conio.h>
# include <ctype.h>
# include <signal.h>
# include <unistd.h>
# include <process.h>
# include <sys/irqinfo.h>
# include <sys/proxy.h>
# include <sys/kernel.h>
# include <sys/types.h>
# include <sys/wait.h>
```

Entrada / Salida con Interrupciones (cont.)

```
/* registros de la UART */
# define TXDATA        0x2F8  /* registro de Tx */
# define RXDATA        0x2F8  /* registro de Rx */
# define INTIDENT      0x2FA  /* reg.de identif. de interr*/
# define LINESTATUS    0x2FD  /* reg.de identif. de interr*/
# define MDMSTATUS     0x2FE  /* reg.de identif. de interr*/

/* causas de interrupcion */

# define NO_INT_PENDIENTE    0x01
# define MODEM_STATUS        0x00
# define TX_BUFFER_VACIO     0x02
# define RX_DATA_AVAILABLE   0x04
# define RX_LINE_STATUS      0x06
```

Entrada / Salida con Interrupciones (cont.)

```
pid_t  rx_proxy;   /* para el PID del proxy usado para
                      notificar el evento de dato recibido */

/* variables volatile (las modifica el interrupt handler) */

volatile char rxbuf[16];

volatile int head=0, tail=0;
```

Entrada / Salida con Interrupciones (cont.)

```
/* handler de interrupcion */
# pragma off ( check_stack );

pid_t far handler( )
{
        switch( intident = inp(INTIDENT) )
        {
        case RX_DATA_AVAILABLE:
                /* se ha recibido un byte; lo guarda y avisa
                   al proceso que lo usará */

                rxbuf [tail] = inp( RXDATA );
                tail = ++tail % 16;
                return rxproxy;
```

Entrada / Salida con Interrupciones (cont.)

```
        case TX_BUFFER_VACIO:
                /* terminó la transmisión de un byte */

                inp( LINESTATUS );   /* clarea la interrup.*/
                break;
        case MODEMSTATUS:
                inp( MDMSTATUS );
                break;
        case RX_LINE_STATUS:
                inp( LINESTATUS );
                break;
        }
        return 0;
}
```

Entrada / Salida con Interrupciones (cont.)

```
# pragma on (check_stack);

void main( )
{       pid_t hijo;
        int     iid, c=0, status;
        if ( (rxproxy = qnx_proxy_attach(0,0,0,-1)) == -1)
                exit(1);

        if ( ( iid=qnx_hint_attach(3, handler,
                                   FP_SEG(rxbuffer))== -1)
                exit(2);

        if ( (hijo = fork ()) == 0)
                transmision( );   /* el hijo transmite */
```

Entrada / Salida con Interrupciones (cont.)

```
        /* el padre se queda esperando recepción, y finaliza
           cuando reciba Escape */

        while ( c != 27 )
        {       pid = Receive( rxproxy, 0, 0 );

                while( head != tail )
                {
                        /* Lee todos los bytes guardados */

                        printf ( "%c", c = rxbuf[head] );
                        head = ++head % 16;
                }
        }
```

Entrada / Salida con Interrupciones (cont.)

```
        /* espera la finalización del hijo */

        do
        {       wpid = waitpid( hijo, &status, 0 );

        } while ( !WIFEXITED ( status ) );

        /* desinstala el handler de interrupción */
        qnx_hint_detach( iid );

        /* da de baja el proxy */
        qnx_proxy_detach( rxproxy );
}
```

Entrada / Salida con Interrupciones (cont.)

```
void transmision( )
{
        char   c=0;

        while( c != 27 )
        {
                c = getche( );

                outp( TXDATA, c );
        }
        exit( 0 );
}
```

Memoria compartida

• Por defecto cada proceso tiene su propio espacio de direciones.

• Desde un proceso no se puede acceder al espacio de direcciones de otro proceso.

• Esto favorece a la seguridad y confiabilidad del sistema.

• Si dos o más procesos necesitan compartir una región de memoria, como por ejemplo para definir semáforos y/o datos compartidos, deben seguirse una serie de pautas.

Memoria compartida

• Si dos o mas procesos deben compartir una zona de memoria, en primer lugar se debe crear un objeto de memoria y asignarle un nombre.

```
# include <fcntl.h>
# include <sys/mman.h>

int shm_open( char *nombre,  int flags,  mode_t modo );
```

nombre: nombre dado al objeto de memoria.
flags: combinación OR de uno o más flags que determinan el modo de apertura de la memoria compartida.

modo: permisos de acceso (similar a archivos, p.ej.: 0777)

Memoria compartida

• shm_open( ) continuación...

Flags :	O_RDONLY	*sólo lectura*
	O_RDWR	*lectura / escritura*
	O_CREAT	*si no existe lo crea*
	O_EXCL	*modo exclusivo*
	O_TRUNC	*si existe, y está combinado con O_RDWR,lo trunca a 0*

Memoria compartida

• shm_open( ) continuación...

Retorna:

> 0	si fue exitoso, ese valor es el descriptor del obj.
-1	si hubo algún error (en errno queda el código de error)

Ejemplo: *creación de un objeto de memoria compartido*

```
void main()
{       int   fd;
        fd = shm_open( "sh_vars",
                        O_CREAT | O_RDWR | O_TRUNC,
                        0777 );
```

Memoria compartida

• Uno de los procesos (normalmente el que crea el objeto) debe establecer el tamaño del área de memoria compartida.

```
# include <sys/types.h>
# include <unistd.h>

int  ltrunc( int fd,  off_t  tam,  SEEK_SET );
```

fd : descriptor (obtenido de shm_open( ))

tam : tamaño en bytes a fijar.

Memoria compartida

• Se debe mapear el objeto de memoria compartida dentro del espacio de direcciones del proceso, para poder accederlo

```
# include <sys/mman.h>

void * mmap( void * addr,  size_t  tam,  int  prot,  int  flags,
                int  fd,  off_t  off );
```

addr : dirección de destino (debe ser 0)
tam : tamaño en bytes del bloque a mapear.
prot : propiedades de acceso (detalla en hoja siguiente)
flags : MAP_SHARED para que se compartan los cambios
fd : descriptor (obtenido de shm_open())
off : offset desde el comienzo del objeto (En gral.: 0)

Memoria compartida

• mmap( ) continuación...

Retorna: dirección del objeto de memoria dentro del espacio de direciones del proceso.

-1 si hubo algún error (errno : cod. error)

Ejemplo: *mapeo de un objeto de memoria compartida.*

```
void main()
{       int   fd,  *sh_data;
        .....
        sh_data = mmap( 0, 10*sizeof(int),
                        PROT_READ | PROT_WRITE,
                        MAP_SHARED, fd, 0 );
```

Memoria compartida

• Luego de mapear el objeto de memoria compartida, ya no se utiliza el descriptor, y se puede hacer:

```
close ( fd );
```

Donde: fd es el valor obtenido de shm_open( ).

```
# include <unistd.h>

int close ( int fildes );
```

Retorna: -1 si hubo error (errno guarda el cód. de error)

• El objeto de memoria compartida permanece en el sistema luego de haber efectuado close( ).

Memoria compartida

• Para eliminar del sistema un objeto de memoria compartida se lo debe desvincular como si fuera un archivo:

```
shm_unlink( nombre );
```

Donde: nombre es la cadena de char correspondiente al nombre del objeto.

```
# include < sys/mman.h >

int   shm_unlink ( const char * name );
```

Retorna: 0 si tuvo éxito.
-1 su hubo error (errno guarda el cód. de error).

Memoria compartida

• Ejemplo: proceso que crea un objeto de memoria compartida, y luego almacena datos en el mismo.

```
# include <stdio.h>
# include <conio.h>
# include <string.h>
# include <unistd.h>
# include <limits.h>
# include <fcntl.h>
# include <sys/kernel.h>
# include <sys/mmap.h>

void main( )
{
```

Memoria compartida

Ejemplo: continuación...

```
void main( int argc, char *argv[ ] )
{       int       fd,  status;
        char    *shmp;
        pidt_t   pid;

        if ( argc > 1 &&  stricmp( argv[1], "CLOSE")==0 )
        {
                /* se ejecutó el programa para eliminar un
                   objeto de memoria compartida */

                shm_unlink( "shm_txt" );
                exit( 0 );
        }
```

Memoria compartida

Ejemplo: continuación...

```
        /* crea un nuevo objeto de memoria */

        fd = shm_open("shm_txt", O_RDWR | O_CREAT, 0777);

        if ( fd == -1 )
        {
                perror( "shm_open" );
                exit(1);
        }
```

Memoria compartida

Ejemplo: continuación...

```
        /* le asigna un tamaño de 80 bytes (Esto debería
           hacerse sólo al crear el objeto) */

        if ( ltrunc ( fd, 80, SEEK_SET ) == -1 )
        {
                perror( "ltrunc" );
                exit(1);
        }
```

Memoria compartida

Ejemplo: continuación...

```
        /* mapea la memoria compartida dentro del
           propio contexto */

        shm_p = mmap ( 0, 80, PROT_READ | PROT_WRITE,
                              MAP_SHARED, fd, 0);

        if ( shm_fd == -1 )
        {
                perror( "shm_open" );
                exit(1);
        }
        close( fd );   /* ya no se utiliza más a fd */
```

Memoria compartida

Ejemplo: continuación...

```
        /* ingresa una cadena de texto y la almacena en
           la memoria compartida */

        gets( shm_p );
}
```

Si se quiere que otro proceso acceda a estos datos y los imprima, podría reemplazarse esta última sentencia por;

```
puts( shm_p );
```

Asimismo, en ese caso no debería hacerse ltrunc( ), *ya que no se desea crear un objeto, sino acceder a uno existente.*

Semáforos

- Los semáforos son un caso particular de variables enteras.
- Deben estar ubicadas en una región de memoria compartida entre los procesos que los utilizan.
- Se dispone de las siguientes funciones :
  - sem_create( ) *para crear un semáforo.*
  - sem_destroy( ) *para eliminar un semáforo.*
  - sem_wait( ) *para testear y decrementar un semáforo (ponerlo en "rojo").*
  - sem_post( ) *para incrementar un semáforo (ponerlo en "verde").*

Semáforos

• Los semáforos son un caso particular de variables enteras.

• Si un proceso invoca a la función *sem_wait(&semaf)* y la variable (semáforo) tenía un valor mayor que cero, el proceso no se bloquea y la variable es decrementada.

• Si se hace s*em_wait( )* sobre una variable que está en cero, el proceso se "duerme", hasta que otro proceso levante dicho semáforo haciendo *sem_post( )*.

• Es válido que un signal handler intente esperar en un semáforo, y eventualmente mande a dormir al proceso al cual pertenece.

Semáforos

• Las variables que se utilizan como semáforos deben estar ubicadas en una región de memoria compartida entre los procesos que las utilizan.

• Se dispone de las siguientes funciones :

- sem_create( ) *para crear un semáforo.*
- sem_destroy( ) *para eliminar un semáforo.*
- sem_wait( ) *para testear y decrementar un semáforo (ponerlo en "rojo").*
- sem_post( ) *para incrementar un semáforo (ponerlo en "verde").*

Semáforos

• Creación de un semáforo:

 # include <semaphore.h>

*int sem_create(sem_t * semp, 1, unsigned valor_inicial);*

Donde: semp : puntero al semáforo

valor_inicial : debe ser ≥ 0

Semáforos

• Eliminación de un semáforo:

 # include <semaphore.h>

*int sem_destroy(sem_t * semp);*

Donde: semp : puntero al semáforo

Semáforos

• Esperar en un semáforo:

 # include <semaphore.h>

*int sem_wait(sem_t * semp);*

Donde: semp : puntero al semáforo

Si el semáforo tenía previamente un valor > 0, el proceso no se bloquea y continúa su ejecución.

Si el semáforo tenía previamente un valor = 0, el proceso es bloqueado (SEMAFORE_BLOCKED), hasta que otro proceso "levante" el semáforo en cuestión.

Semáforos

• "Levantar" un semáforo:

 # include <semaphore.h>

*int sem_post(sem_t * semp);*

Donde: semp : puntero al semáforo

El valor del semáforo indicado es incrementado en uno.

Si el semáforo estaba en cero, y había algún proceso bloqueado en ese semáforo, el scheduler lo pondrá nuevamente en "READY".

Semáforos

• Ejemplo:

```
# include <semaphore.h>

sem_t  * S1;    /* puntero a un semáforo; se lo hará apuntar
                   a un área de memoria compartida */
int * sh_data; /* puntero a datos en memoria compartida */

void main( )
{       ........
        sem_init ( S1, 1, 1 );   /* creación de un semáforo
                                    con valor inicial 1 */
        procesar( 10 );
        sem_destroy( S1 );
}
```

Semáforos

• Ejemplo *(continuación)* :

```
void procesar( int N )
{
        int i, x;
        for( i=0 ; i < N ; i++ )
        {       /* Región no crítica */
                x = getchar( );
                /* Región crítica */
                sem_wait( S1 );
                sh_data[i] = x;
                sem_post( S1 );
        }
}
```

Semáforos

• Ejemplo *(continuación)* :

La función *procesar( )* tiene una parte de su trabajo sobre una zona de memoria compartida, entonces :

• Antes de entrar a la región crítica intenta "bajar" el semáforo:
  • Si lo logra "bajar" ingresa.
  • Si el semáforo estaba "bajo" (en cero), se bloquea.

• Al salir de la región crítica "levanta" el semáforo. Si había algún proceso "durmiendo" en este semáforo, el scheduler lo "despertará".

APENDICE A

Comandos Básicos

Ing. Guillermo Friedrich - UTN - FRBB

Para obtener ayuda acerca de algún comando:

use <comando>

Por ejemplo:

use cc Muestra las opciones para invocar al compilador C.

Para conocer el directorio actual

pwd

Listar el contenido de un directorio

ls	Muestra la lista de archivos de manera breve
ls -l	Muestra detalles de los archivos/directorios

Cambiar de directorio

cd /home/curso	Cambia al directorio /home/curso
cd /	Cambiar al directorio raiz
cd ..	Cambiar al directorio padre

Crear un subdirectorio

mkdir <nuevo_subdirectorio>

Algunos ejemplos:

mkdir /home/curso/garcia

mkdir garcia

Eliminar un subdirectorio

rmdir <subdirectorio_a_eliminar>

Copiar un archivo

cp <origen> <destino>

Algunos ejemplos:

cp /home/curso/cliente.c /home/garcia

cp * /home/

cp /dos/a .

cp /usr/include/*.h /home/varios

Mover o renombrar un archivo

mv <origen> <destino>

Algunos ejemplos:

mv cliente.c cli.c

mv /home/curso/cliente.c /home/garcia

mv /dos/a .

mv /usr/include/*.h /home/varios

Montar las particiones MS-DOS

Dosfsys

Crea un conjunto de subdirectorios, por ej.:

/dos/c (Unidad C:)
/dos/a (Unidad A:)

Con mayor o menor cantidad de ellos, de acuerdo a las unidades encontradas.

Dosfsys a=/dev/fd0 solamente monta el drive A: en /dos/a

Dosfsys -x desmonta las unidades MS-DOS.

Compilar un programa en C

cc prog.c

compila el archivo prog.c y produce los siguientes archivos:

prog.o (objeto)

a.out (ejecutable)

cc -o prog prog.c

produce un ejecutable de nombre prog

Compilar un programa en C (continuación)

cc -o prog prog.c aux.c

compila los archivos fuente prog.c y aux.c, produciendo los siguientes archivos:

prog.o (objeto)
aux.o (objeto)

prog (ejecutable)

Para ejecutar el programa obtenido se debe hacer:
./prog

Cambiar de consola virtual

Se debe presionar Ctrl-Alt-<Nro>

Donde <Nro> es el Nº de consola a la que se quiera pasar.

Por ej.: Ctrl-Alt-1 pasa a la primera consola virtual
Ctrl-Alt-0 pasa a la décima consola virtual

Normalmente hay entre 6 y 10 consolas virtuales.

Ver el estado de los procesos

Hay dos comandos para ver el estado de los procesos:

ps

sin

Ambos muestran, con diferente presentación, la lista de procesos actual, con los siguientes datos, entre otros:

PID - Estado - Nombre del programa - Tipo de planificación

Shell Scripts

Son archivos de comandos (equivalentes a los *.BAT)
Se los puede ejecutar haciendo: sh scriptfile

O bien, si tiene establecido el permiso de ejecución, tipeando directamente:
./scriptfile

Para cambiar los permisos de un archivo se hace, por ejemplo, lo siguiente:
chmod 744 scriptfile

Esto asigna los siguientes permisos: rwx r-- r--

Shell Scripts (continuación)

Para tomar los parámetros que se le pasa a un script se usa "$1" para el primero, "$2" para el segundo, etc.

Por ejemplo, el archivo script llamado: compilar

tiene el siguiente contenido: cc -o "$1" "$1.c"

Si se lo invoca así: compilar hello

hace lo siguiente: cc -o hello hello.c

Utilitario make

La forma básica de invocarlo es:

```
make -f makefile
```

Si el contenido de <makefile> fuera, por ejemplo:

```
hello.o : hello.c
    cc -c hello.c
```

El resultado sería que si hello.c tiene una fecha-hora más reciente que la del archivo objeto, lo compila nuevamente.

Edición de archivos de texto

QNX provee dos editores de texto: el tradicional vi de Unix, y el vedit, que permite trabajar en modo pantalla.

La forma de invocarlo es:
vedit <archivo>

Una vez dentro del editor, presionando Escape da la opción de salir (grabando o no las modificaciones).

Presionando: F1 se obtiene la ayuda,
Alt-L se presenta la barra de menú