IPC

Sistemas Operativos II

DC - FCEFyN - UNC

Morales, Julian (jmorales@unc.edu.ar)

Abratte, Diego (?)

Actualización: 06/04/2020

Introducción

1. Fork

La creación de procesos consiste en: crear un proceso en un nuevo espacio de direcciones, leerlo y ejecutarlo. La mayoría de los sistemas operativos implementan un mecanismo *spawn* que se encarga de estas tareas, mientras que Unix es el único que divide este proceso en tres conjuntos de funciones: *fork()*, *exec()* y clone().

La función *fork*[1]*()* crea una copia del proceso actual, que se denomina proceso hijo. Esta difiere del padre, únicamente en su PID. El proceso padre se bifurca comenzando un nuevo proceso y haciendo una copia de todos los datos y contexto del proceso viejo. La segunda función exec(), que en realidad es una familia de funciones, es la encargada de de cargar el nuevo ejecutable en el espacio de direcciones y comenzar la ejecución. Llamadas a execve causan que la imagen del programa sea reemplaza (con un nuevo stack, heap y un nuevo segmento de datos, liberando los viejos)[2]

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main()
{
   pid_t pid;
   int rv;

   switch(pid=fork()) {
       case -1:
           perror("fork"); /* algo salió mal */
           exit(1); /* el padre termina */
       case 0:
           printf(" HIJO: este es el proceso hijo!\n");
           printf(" HIJO: Mi PID es %d\n", getpid());
           printf(" HIJO: El PID de mi padre es %d\n", getppid());
           printf(" HIJO: Ingrese mi estado de salida (hágalo corto): ");
           scanf(" %d", &rv);
           printf(" HIJO: Aquí esta mi estado de salida!\n");
           exit(rv);
       default:
```

[1] Ver página 2 del manual de fork, ya que la explicación expuesta acá no es lo que realmente pasa aunque sí una aproximación conceptual.

[2] https://www.hackerearth.com/practice/notes/memory-layout-of-c-program/

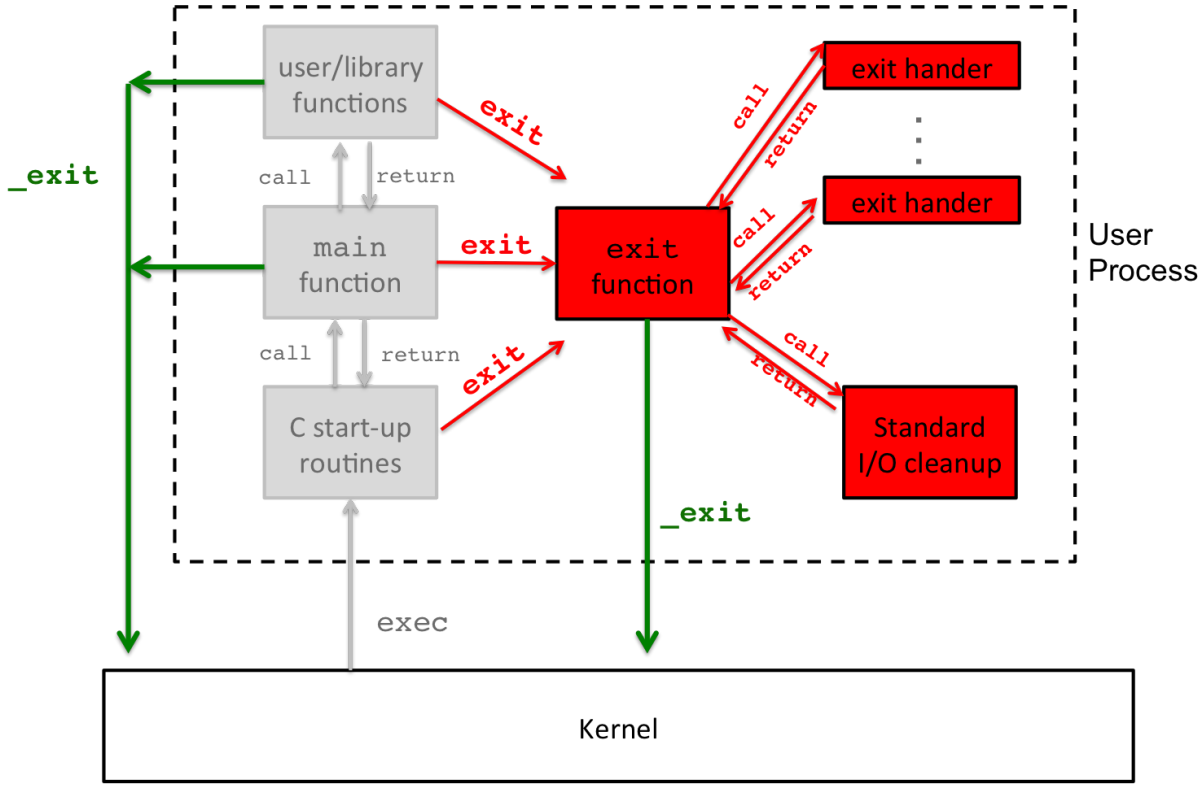
```
            printf("PADRE: Este es el proceso padre!\n");
            printf("PADRE: Mi PID es %d\n", getpid());
            printf("PADRE: El PID de mi hijo es %d\n", pid);
            printf("PADRE: Ahora espero a mi hijo para salir...\n");
            wait(&rv);
            printf("PADRE:El estado de salida de mi hijo es %d\n",
                   WEXITSTATUS(rv));
            printf("PADRE: Aquí estoy afuera!\n");
    }
}
```

1.1 Fin de procesos

Formas de forzar la salida del programa desde el código:

1. **Segfault**[3]
2. **_exit()** : Hace un System calls que llama a una rutina para terminar el proceso, lo debería terminar inmediatamente sin ejecutar código adicional.
3. **exit()** : Procedimiento de la C Standard Library limpiara y terminará el procesos invocando código adicional, como el i/o cleanup y los exits handlers.
4. **_Exit()** : C Standard Library basicamente es un wrapper de _exit()

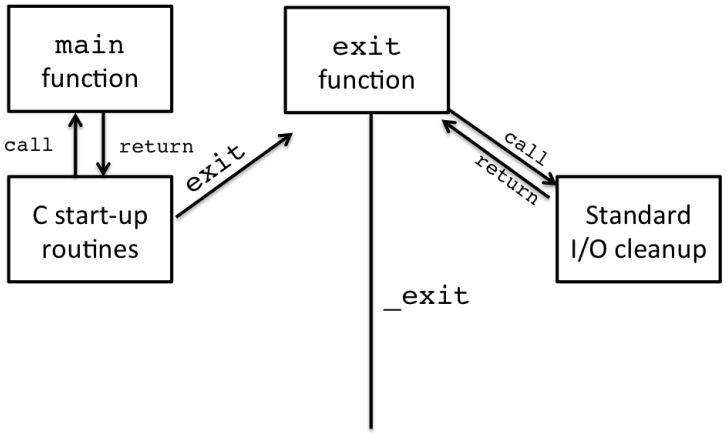
[3] Ver Segunda respuesta donde explica lo de /proc/<pid>/maps Aca hay una.breve intro, muy por arriba como funciona la memoria (stack, heap) y la prevención que toma el compilador para intentar evitar ataques por buffer overflow:
https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/08/lec.html



Con un _exit sale inmediatamente, con un exit llama la función que termina ejecutando el _exit con cosas antes.

1.1.1 I/O Buffering and Exit Procedures

Se Debe manejar el buffer i/o de la librería standard de C (Standard I/O Cleanup), funciones como write/read implican un doble cambio de contexto (aunque a veces parcial -> Cambio de segmento de pila+código) que optimizan escrituras y lecturas usando buffers.



Cuando hay llamadas i/o para escribir, por ejemplo con printf, va a un buffer que se escribe en general cuando llega a un salto de línea, un fin de string o el buffer está lleno.

4 ejemplos de comportamientos:

1.

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
   printf("Hello World!");
   exit (0);
}
```

2.

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("Hello World!");
    _exit (0);
}
```

3.

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("Hello World!\n");
    _exit (0);
}
```

4.

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
```

```c
int main() {
   printf("Hello World!");
   fflush(stdout);
   _exit (0);
}
```

- ➔ Si cleanup no puede cancelar los request Async, se bloquea hasta que lo hace
- ➔ Se cierran todos los fd
- ➔ Manda un SIGCHLD al padre que si no la ignora (comportamiento por defecto), puede quedar como un proceso zombi
- ➔ Si muere el padre no se muere el hijo, lo adopta init (1), que puede rechazarlo o meterlo dentro del grupo de huérfanos, (Daemons && Signal(SIGHUP, SIG_IGN), setsid)
- ➔ Se eliminan los locks file. (fcnlt)

1.1.2 Política de buffering

El stderr no tiene buffer de salida, por lo que es más caro de utilizar, pero tiene la ventaja de que siempre podemos informar el error antes de un _exit de una forma clara.

Los buffers de STDIN y STDOUT tienen buffers por línea, pero podemos deshabilitar esto y dejar solo la protección de cuando el buffer está lleno.

Los cambios de políticas son con setbuf, setbuffer, setlinebuf, **setvbuf**[4] (ver man setbuf)

```c
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

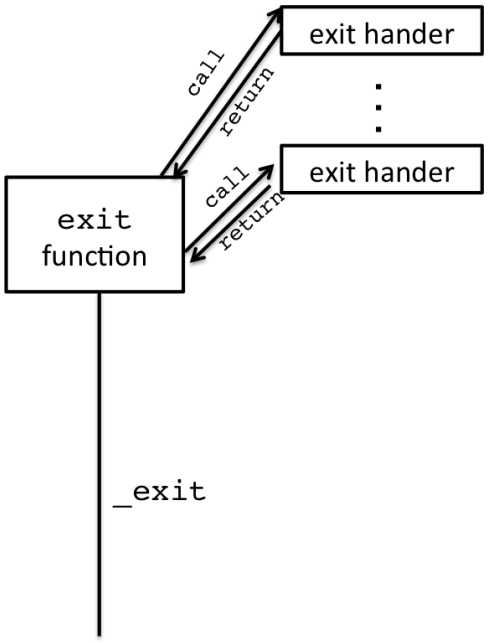
- ➢ **_IONBF** unbuffered
- ➢ **_IOLBF** line buffered
- ➢ **_IOFBF** fully buffered

Si no se desea cambiar el buffer se lo setea en tamaño 0 con puntero a null

[4] man setvbuf

1.2 Exit Handlers

Muchas veces queremos estandarizar la salida o cortar la ejecución en muchos lugares, o en algún lugar determinado pero por scope hay operaciones que no podemos realizar, pero son realmente necesarias antes de salir (ej: cerrar sets de archivo, liberar puertos u otros recursos)



La solución es registrar handlers al exit para que se ejecuten en caso de terminar el programa con exit (o con return en el main que hace llamada implícita al exit)

El manejador de salida tiene una pila enlazada (se ejecuta al revés) con los punteros a funciones que son llamadas antes de salir. Para agregar elementos a la pila es con atexit y on_exit, donde en esta segunda se le pueden pasar argumentos. (Ver manuales on_exit y atexit)

ej:

```c
#include <stdio.h>
#include <stdlib.h>

void hand_ex1(){
  printf("Yo dsp\n");
}

void hand_ex2(){
  printf("Yo me ejecuto primero\n");
}
```

```c
int main(){

  atexit(hand_ex1);
  atexit(hand_ex2);

  return 0;
}
```

1.3 Exit Status

Todos los programas salen con algún estado, por convención

0 : success

1 : failure

2 : error

```c
_exit(int status);
exit(int status);
_Exit(int status);
```

En bash, la variable $? guarda información del estado de la salida de la última ejecución. La salida en verdad es más que solo el estado (Por qué no usar void main?)

1.2 Zombies

Un ejemplo aca[5], este es el mejor de los casos, en el que el hijo no muere, queda esperando que el padre recoja la información y lo mate.

- ➔ Cuando un proceso finaliza, toda la memoria y los recursos asociados con él se liberan.
- ➔ El estado de salida se mantiene en la PCB hasta que el padre toma el estado de salida usando wait y elimina la PCB (Process Control block -> /proc/)
- ➔ Un proceso hijo siempre se convierte primero en zombie
- ➔ En la mayoría de los casos, bajo la operación normal del sistema, los zombis son atendidos inmediatamente por sus padres.

[5] https://github.com/maly6600/operating/blob/master/Kod/module-2/examples/src/fork_zombie.c

- ➔ Los procesos que permanecen zombis durante mucho tiempo son generalmente un error y causan un leak de recursos. (Mejoró mucho el comportamiento en Kernel 3)
- ➔ Cuando los padres mueren pueden ser adoptados por init que se encarga de limpiarlos de ser necesario.

2. Signals

Es un método muy simple para que un proceso avise algo a otro. Un proceso puede activar una señal y tener algún efecto sobre otro proceso. Son eventos asíncronos que causan que se interrumpa el flujo de ejecución de un proceso y se transfiera el control a la función de manejo de la señal recibida.

Estas señales pueden ser generadas por el kernel o por otro proceso. La llamada al sistema KILL o su wrapper kill de consola, en unix, es una manera de enviar señales a procesos. El comportamiento del proceso depende del tipo de señal, algunas pueden ser ignoradas y otras deben ser atendidas si o si, todos los procesos traen rutinas de manejo de señales por defecto, pero para algunas, el usuario puede reemplazar este comportamiento por defecto.

Es importante tener en cuenta que por ser ejecutadas de manera asíncrona no se puede utilizar cualquier función en el manejo, solo pueden usarse “async safe functions”[6]

La tabla de señales se puede ver en la página 7 del manual de signal (man 7 signal), las señales se pueden enmascarar y también ignorar.

2.1 Cómo escribir un mal código

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <signal.h>
int main(void)
{
  void sigint_handler(int sig);
  char s[200];
/* prepara el handler */
  if (signal(SIGUSR1, sigint_handler) == SIG_ERR) {
    perror("signal");
    exit(1);
  }
  printf("Ingrese un string:\n");
  if (gets(s) == NULL) // <??
    perror("gets");
  else
```

[6]async safe functions: http://man7.org/linux/man-pages/man7/signal-safety.7.html

```
    printf("usted ingresó: \"%s\"\n", s);
  return 0;
}
/* este es el handler */
void sigint_handler(int sig)
{
  printf("Ahora no!\n");
}
```

Vemos un ejemplo de cómo funciona con pstree/ps aux / kill -10

2.2 Uso de sigaction

Puede verlo en la página 2 y 3p del manual de sigaction para información más detallada.

Se utiliza para setear manejadores de señal y/o obtener información sobre el manejador actual.

```
int sigaction(int sig, const struct sigaction *restrict act,  struct
sigaction *restrict oact);
```

- **int sig :** numero de señal
- **sigaction act :** estructura que contiene:
    - Puntero a la nueva función para manejar la señal, si es NULL no se modifica el manejador actual
    - Máscara de señales que se desea deshabilitar durante la ejecución del manejo de la señal
    - flags: setea comportamientos dicionales, por ejemplo si una función fue interrumpida por la señal, al finalizar el manejador se realiza de nuevo la llamada a la función
- **sigaction oact**: mismo que el anterior, solo que aquí se almacena el manejador actual, si act no es NULL el manejador viejo queda almacenado aquí, si act es NULL esto puede usarse para obtener info del manejador actual.

2.3 Uso de pause

Causa que el proceso se suspenda hasta recibir una señal. Si el comportamiento que involucra el manejo de la señal es la terminación del proceso esta función no retorna.

3. Pipe

Es la forma de IPC más simple, una llamada a la función pipe retorna un par de file descriptors. Uno de estos descriptores se conecta al extremo de escritura y el otro al de lectura. Algo puede escribirse en un extremo del pipe, y ser leído en el otro extremo en el orden en el que vino. El tamaño del buffer suele ser de 10K.

El pipe debe compartirse entre procesos, por lo que en general es necesario un fork.

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
int main ()
{
  int pfds[2];
  char buf[30];
  if (pipe (pfds) == -1)
    {
      perror ("pipe");
      exit (1);
    }
  printf ("file descriptor para escritura #%d\n", pfds[1]);
  write (pfds[1], "test", 5);
  printf ("file descriptor para lectura #%d\n", pfds[0]);
  read (pfds[0], buf, 5);
  printf ("leo \"%s\"\n", buf);
  return 0;
}
```

No vamos a profundizar en su uso por que es extremadamente sencillo y fue visto en SOI pero su utilidad más fuerte es combinarlo con un fork, y es tan comun que hay una llamada para hacerlo todo junto llamada popen()[7]

[7] https://pubs.opengroup.org/onlinepubs/009695399/functions/popen.html

4. FIFO

Un FIFO es como un pipe con nombre. El nombre es el de un archivo que los procesos múltiples pueden abrir con open ( ), leer y escribir.

Las FIFOs se diseña para saltar las limitaciones de los pipes normales a los que 2 procesos no relacionados se quieran comunicar ya que si uno crea el pipe no puede pasarlo al otro proceso al menos que se haya creado con un fork luego de llamar a pipe.

La solución es crear un archivo virtual al igual que lo hacíamos cuando queríamos crear entradas en /dev, solo que ahora no va ser un dispositivo de caracteres ni de bloques, va a ser de tipo FIFO

```
mknod "fifofile" p
```

Prototipo, donde el número de dispositivo se ignora, el modo son los permisos junto con el tipo de archivo.

```
int mknod(const char *pathname, mode_t mode, dev_t dev);
```

El mknod posee un wrapper en línea de comandos con el mismo nombre, ambas documentaciones pueden verse en las páginas 1 y 2 del manual de mknod.

Un ejemplo simple de las fifo se pueden encontrar en:
https://github.com/LambdaSchool/Applied-C-Programming/blob/master/speak.c
https://github.com/LambdaSchool/Applied-C-Programming/blob/master/tick.c

Estas fifos tienen la característica por defecto de bloquearse en el open, la de escritura cuando no hay nadie para leer del otro lado, y la de lectura cuando no hay nada para leer, aunque los comportamientos de bloqueante y no bloqueante se pueden cambiar como si fuera cualquier otro fd.

Otra de las ventajas que contiene es que puede haber múltiples procesos escribiendo.

5. Colas de mensajes (SysV)

Una cola de mensaje trabaja en forma similar a una FIFO, pero soporta alguna funcionalidad adicional.No son inodos del file system sino memoria.

También tiene lectura destructiva pero se le puede enviar cualquier cosa (no solo strings). Generalmente se sacan mensajes de la cola en el orden en el que fueron colocados. Hay maneras específicas de arrancar ciertos mensajes de la cola antes de que alcancen el frente de la misma.

Un proceso puede crear una nueva cola del mensaje, o puede conectarse a una que exista. Es importante destacar que la cola no se elimina hasta que la destruyamos, si nos olvidamos quedará abierta una vez finalizado los procesos conectados a la misma.

Para crear o conectarse a una cola que ya existe se usa msgget

```
int msgget(key_t key, int msgflg);
```

Retorna un ID de la cola creada o a la conectada.

key_t key: Es una key que funciona como un UID a nivel SO, cualquiera que lo conozca se puede conectar a la cola (Si tiene los permisos)

msgflg: Parámetros para crear la cola, básicamente un **IPC_CREAT** | **Permisos** (UGO)

Las colas determina su grupo y usuario con el grupo y usuario del proceso que la creó.

5.1 De donde sale el Master Key

key_t es un typedef de un long, pero es importante no hardcodear el numero por si alguien hardcodear el mismo sería un desastre. Para asegurarse de que ese número será único está la función ftok. Esta genera un número único a partir de una ruta de un archivo, obtiene información sobre su inodo y genera el UID.

Si otro proceso quiere obtener el mismo token solo debe llamar a la función con los mismos parámetros.

5.2 Operando sobre la cola

Para operar sobre la cola se debe obtener su ID a través de la función msgget(). Los datos enviados a través de la cola pueden tener cualquier formato, lo importante es que comienzan con un **long** lo cual es requerido por la implementación de la cola de mensajes para identificar los distintos tipos de mensajes.

El envío y recepción de mensajes es bastante simple, solamente es necesario invocar a las funciones respectivamente

```
msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

Debe notarse el argumento extra de msgrcv, en el cual se especifica el tipo de mensaje que se desea leer. El resto de los argumentos en orden son el ID de la cola, un puntero al mensaje, el tamaño del mensaje y los flags (bloqueante, no-bloqueante, etc)

**Para ver las colas y otros mecanismos IPC del sistema usar el comando *"ipcs"*

6. Colas de mensajes (POSIX)

6.1 Acceso / Creación de colas

Las colas son creadas usando la función

```
mqd_t    mq_open(const char *, int, ...);
```

Téngase en cuenta que esta función recibe cantidad de argumentos variable, tal como es indicado por el parámetro '**...**' en su prototipo. Esta función debe ser usada tanto para crear la cola como para obtener el identificador de una cola previamente creada. Al igual que las colas systemV son persistentes por lo que a menos que sean destruidas usando la función mq_unlink() permanecerán en el sistema hasta que ocurra un reboot.

Como se puede observar en las páginas del manual aparecerán dos formas de usar mq_open debido a que puede recibir cantidad variables de argumentos.

1) mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr);

2) mqd_t mq_open(const char *name, int oflag);

Al momento de crear una cola se debe usar 1) ,el parámetro name sirve para identificar la cola y debe ser un string con el formato "/nombredecola", en el parámetro flag debe incluir O_CREAT el cual indica que deseamos crear una nueva cola, el argumento mode_t setea permisos y sus posibles valores se pueden ver en **man 2 open** ya que las macros de permisos son comunes con la de los archivos. Luego como último parámetro debe pasarse una estructura mq_attr donde seteamos el tamaño máximo de los mensajes que pueden escribirse en la cola, y la cantidad máxima de mensajes.

Para obtener el identificador de una cola que ya fue creada debe usarse 2) aqui el parámetro name debe ser igual al que fue pasado por el proceso que creó la cola, es decir, ambos procesos deben conocer a priori el nombre de cola que se va a utilizar y luego las flag para especificar el modo en el que se desea acceder a la cola(solo lectura, solo escritura, lectura/escritura, etc). Debe tenerse en cuenta que si el proceso no cuenta con los permisos la llamada fallará, por lo tanto si el proceso que crea la cola desea permitir que otro pueda escribir en ella deberá setear los permisos correspondientes en el parámetro mode_t mode al momento de la creación.

6.2 operando sobre la cola

Para enviar mensajes se utiliza la siguiente función

```
int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned
int msg_prio);
```

Aqui se debe especificar el identificador de cola(obtenido a traves de mq_open) un puntero al mensaje, el largo del mensaje en bytes y la prioridad asociada al mensaje.

Para leer un mensaje se utiliza la función

```
int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned
int msg_prio);
```

Nuevamente el lado receptor debe pasar el identificador de la cola a través de la cual desea recibir, un puntero al buffer donde va a almacenar el mensaje, el largo del mensaje que espera recibir y la prioridad del mensaje que espera recibir por lo tanto el transmisor y receptor deben conocer el formato de los mensajes.

Cuando un proceso desea dejar de utilizar una cola debe llamar a la función mq_close(mqd_t mqdes) pasandole como argumento el identificador de cola o descriptor de cola(que fue obtenido con mq_open) similar a como se trabaja con archivos. Esto solo genera que dicho proceso deje de referenciar a la cola.

Si se desea destruir la cola se debe llamar a la función mq_unlink(cons char *name) proporcionando el nombre de la cola. La cola solo será destruida cuando todos los procesos que la estaban usando dejen de referenciarla a través de la llamada a mq_close().

6.2 Linkeo

Para utilizar colas posix se debe agregar el flag -lrt en la compilación si no se generará un error en la etapa de linkeo

Ej: **gcc main.c -Wall -pedantic -Werror -Wunused - lrt -o main**

7. Memoria compartida

Segmentos de memoria compartidos

Es básicamente lo que el nombre indica, un segmento de memoria que puede ser usado por múltiples procesos.

1) obtener el token con ftok igual que para las colas de mensajes
2) llamar a shmget(key_t key, size_t size, int shmflag) para obtener el numero identificador del segmento.

Argumentos :

-key : valor obtenido por ftok

-size tamaño del segmento(se redondea al tamaño de página)

- shmflag :se puede especificar si se desea crear el segmento IPC_CREAT, también se pueden agregar los permisos usando el formato octal 0777

3) llamar a shmat() para obtener un puntero al segmento compartido
4) para realizar modificaciones y obtener información de control del segmento(permisos, etc) se puede utilizar la función shmctl() .Se debe contar con los permisos necesarios !!!!

ver tmb : shm_open() + ftruncate() + mmap() combo para realizar operaciones similares.Ver tmb man shm_overview para la lista completa de operaciones.

MMAP

La función mmap() permite acceder a datos/recursos a través de la manipulación directa de direcciones sin la necesidad de realizar llamadas al sistema como read(), write(), lseek(), etc.

Esta función asocia un objeto, como por ejemplo un archivo a una parte del espacio de direcciones del proceso, agregando entradas a la tabla de páginas que referencian las objeto. Podría pensarse como un método para “levantar un archivo en RAM”, al acceder a los datos el sistema de manejo de memoria carga los contenidos del archivo en páginas de la memoria ram(todo esto en el caso de mapeos respaldados por archivos, osea el file descriptor puede también referenciar otro tipo de objeto).Verdaderamente cuantas paginas

de dato esten efectivamente en ram depende del sistema de manejo de memoria del SO, el cual se encuentra fuera del alcanze del programador.

El archivo en disco es modificado cuando la memoria mapeada no es referenciada por nadie mas(todos los procesos que estaban usandola llamaron munmap()) o cuando se necesita generar espacio para la memoria de otros procesos.

Usar este metodo tiene algunas ventajas, el mapeo puede senalarse como compartido, es decir que si otro proceso invoca a mmap para el mismo archivo los dos tienen acceso a la misma porcion de memoria y los cambios que realiza un proceso son inmediatamente visibles por el otro. Ademas si el archivo es usado por muchos procesos esto evita que a) los procesos deban allocar un buffer para leer una parte grande del archivo o b)deban hacer muchas llamadas al sistema para leer partes pequenas. Esto tambien permite hacer algunos trucos para disminuir la probabilidad de que las paginas sean swappeadas para hacer lugar a algun otro proceso, ya que los contenidos pertenecen al espacio de direcciones del proceso se puede acceder a ellos unicamente dereferenciando un puntero, esto hace que la frequencia de eso de las paginas que contienen esos datos aumento y consecuentemente que el SO elija otras paginas menos usadas para swappear. Tambien si los patrones de acceso a los datos son muy aleatorios y frequentes aumenta la performance.

De todas maneras lo anterior es simplemente una guia, los casos de uso pueden o no beneficiarse del uso de esta tecnica ya que distintos tamanos de archivo, patrones de acceso y caracteristicas fisicas y dinamicas(cantidad de procesos, etc) del sistema pueden generar variaciones en la performance.

Es importante tener en cuenta dos situaciones:

1) los accesos a la zona de memoria inmediatamente luego del mapeo generaran fallos de pagina, ya que el mapeo es creado pero la transferencia se realiza bajo demanda. Excepto si se usa la flag MAP_POPULATE la cual genera lecturas anticipadas.

2) El proceso de realizar el mapeo de memoria tiene un overhead alto en relacion a read/write, si bien solo se realiza una vez en casos de lecturas/escrituras esporadicas realizar mapeos podria no ser lo conveniente.

El mapeo de archivos es utilizado en la creacion de procesos y el uso de librerias compartidas.

El caso de analisis hasta ahora fue el mapeo de archivos, pero este tipo no operacion no esta limitada a esto, tambien se puede mapear por ejemplo memoria usada por un “device driver” para compartir una zona de memoria con aplicaciones de espacio de usuario.Xorg por ejemplo usa esto para acceder a la zona de memoria que mapea las tarjetas graficas.

7. Semáforos - Locks

eventfd

Esta funcion es usada para notificar eventos entre aplicaciones de usuario o entre el kernel y una aplicacion de usuario. Este objeto contiene un contador que es mantenido por el kernel y que es modificado es las llamdas a read y write luego de su creacion.

int eventfd(unsigned int initval, int flags)

retorna : file descriptor para operar sobre el objeto

initval : valor inciial del contador

flags : se puede setear comportmiento no bloqueante y comportamiento de semaforo.

read sobre fd de un objeto eventfd :

1) si el contador es 0 y se seteo como no-bloquenate retorna EAGAIN
2) si el contador es 0 y no se seteo como no-bloqueante el proceso se bloquea hasta que el valor del contador sea incrementado
3) si el contador es distinto de 0 y se seteo con semanticas de semaforo la llamada decrementa el contador y retorna 1
4) si el contador es distinto de 0 y no se seteo como semaforo la llamada devuelve el valor del contador y setea el contador a 0

write sobre fd de un objeto eventfd :

suma el valor proporcionado por el buffer [write(fd, const void *buff, size_t count)] al valor del contador del objeto eventfd.

1) si dicha suma ocasionaria overflow y esta como bloqueante el proceso se bloquea hasta que alguien lea el fd
2) si dicha suma ocasionaria overflow y esta como no-bloqueante retorna EAGAIN

**deben tener en cuenta que tanto para el read como para el write si el buffer tiene menos de 8bytes la llamada retorna error

Semaforos POSIX

ver man 7 sem_overview