Grupo : 2202
Asignatura : Sistemas operativos
Alumnos : Lucía Asencio y Rodrigo De Pool

Introducción:

En el siguiente documento responderemos a las preguntas propuestas en la práctica. Además, aclararemos posibles dudas que puedan surgir del código entregado.

Nota:

Para generar todos los ejecutables, solo se necesita ejecutar el comando make en el directorio codigo. Los ejecutables se generarán en este mismo directorio.

Documentación:

La documentación se encuentra dentro de la carpeta doc. El archivo refman.pdf la tiene en el formato indicado y dentro de la carpeta html se puede abrir el archivo index.html que tiene la documentación en formato html.

Ejercicio 4:

Tanto en el caso a) como en el caso b) se generan 7 procesos hijos (8 contando el proceso inicial del programa). En ambos apartados se puede dar el caso de que queden procesos huerfanos, sin embargo la existencia, o ausencia, de estos no será consistente, dependerá fuertemente del planificador y del orden en el que se cierren los procesos.

En el apartado a) se dará muy a menudo ya que no espera a que ningun proceso hijo cierre para finalizar. Podemos comprobarlo con la ejecución del ejercicio: como el ejercicio nos pide imprimir para cada hijo su ppid, y nosotros hemos añadido la impresión de los pid de los padres, sabemos que serán hijos huérfanos aquellos cuyo ppid no se encuentre entre los pid que los padres imprimen. Con esto llegábamos a que extistía cierto proceso (no siempre el mismo, en todas las ejecuciones de un día fue 1518, en todas las de otro día fue 1485) lanzado por init que recogía todos los procesos que quedaban huérfanos.

Por otro lado, en el apartado b) cada uno de los procesos lanzados ejecuta un sólo wait. Esto está bien para los últimos hijos que se crean (cuyo wait es inútil, ya que no tienen nadie a quien esperar), y para sus padres, que sólo tienen un hijo (por lo cual ejecutar un sólo wait es suficiente). A partir de ahí, sus antecesores tienen 2, 3, 4 ,… o más hijos (NUM_PROC como el máximo, éste es el número de hijos que tiene el proceso que ejecuta el primer fork), y de todos estos hijos sólo uno de ellos es waiteado. Por tanto, a no ser que dé la casualidad de que todos los hijos se ejecutan antes que el padre, encontraremos procesos huérfanos. Con números bajos de NUM_PROC es más difícil detectarlos (si como máximo tienes dos hijos, basta con que uno de ellos se ejecute antes que el padre para que no encontremos huérfanos; si tienes 20 hijos, es muy poco probable que 19 de ellos se ejecuten antes que el padre). Así, aumentando NUM _PROC y con el mismo método que en el apartado a), podremos detectar los procesos huérfanos.

Ejercicio 5:

Apartado a:

Para que cada proceso tuviera un único hijo tuvimos que agregar la condicion al bucle de que un proceso no tuviera hijos para mantenerse en el. Luego agregamos un wait(NULL) con el que nos asegurasemos de que cada proceso esperase a su hijo para cerrarse.

Grupo : 2202
Asignatura : Sistemas operativos
Alumnos : Lucía Asencio y Rodrigo De Pool

Podemos ver al ejecutar el programa como cada proceso que imprime su pid, tiene como padre el pid del proceso anterior debido a la ejecución secuencial.

Apartado b:

Ahora para mantenerse en el bucle agregamos la condición de que el proceso sea un proceso padre. Vamos creando los procesos hijos y los cerramos según se vaya imprimiendo su información. Además, en el programa agregamos 3 waits(NULL) al final, de modo que el proceso padre tenga que esperar a que los tres hijos creados finalicen antes de cerrarse.

Vemos que, en efecto, los tres procesos creados tienen el mismo padre, que sería el proceso inicial del programa.

Ejercicio 6:

El proceso padre NO puede acceder a la cadena del hijo. Esto ocurre porque la llamada a fork, al crear un nuevo proceso, otorga al hijo una *copia* del bloque de datos del proceso padre, es decir: los dos procesos tiene acceso a una cadena, pero ésta no es la misma: el fork ha hecho que el proceso hijo tenga su propio espacio de memoria con otra cadena dentro. Por esta razón debemos liberar la cadena tanto en el proceso padre como en el hijo, si queremos evitar fugas de memoria.

Intentamos comprobar si realmente las dos cadenas estaban en dos lugares diferentes en memoria imprimiendo su dirección con %p; sin embargo, este dato era igual para ambas cadenas: se debía a que %p estaba imprimiendo su posición en memoria virtual, que era igual en ambas, no su posición real.

Ejercicio 8:

Se pudo comprobar su correcto funcionamiento ejecutando diferentes programas pasados como argumentos. Es importante tener en cuenta que para ejecutar comandos del PATH se tiene que usar los flags -lp ó -vp, ya que son estos los que ejecutan el código pertinente para los comandos. En caso de que se ejecute con flags -l ó -v tiene que darse el path completo hasta el ejecutable (ó el relativo desde donde se ejecuta el programa).

Ejercicio 9:

NOTA: en el enunciado nos resultaba ambiguo, pero entendimos que cada operación se realizaba con los mismos dos operandos que el padre pedía al principio del main, y que no debíamos pedir nuevos operandos para cada operación. La implementación refleja esta decisión.

Para este ejercicio hemos optado por crear dos “grupos” (arrays) de tuberías: fd1 (escribe padre – lee hijo) y fd2 (escribe hijo – lee padre). A su vez, cada array contiene 4 tuberías (una por operación).

De manera secuencial, hacemos un fork para cada operación, haciendo que el padre espere al hijo correspondiente antes de ejecutar el siguiente fork.

El padre escribe en todos los casos en la tubería los dos operandos que ha pedido por pantalla al principio del main. El hijo los lee, y escribe el resultado correspondiente en la otra tubería, que es leída e impresa por el padre.