Grupo 2101 Pablo Marcos Manchón <pablo.marcosm@estudiante.uam.es>
David Nevado Catalán <david.nevadoc@estudiante.uam.es>

Sistemas Operativos - Practica 1

Documentación

Ejercicio 4

En ambos programas, para que sea el hijo el que imprima el pid del padre y el suyo, en la rama de ejecución del hijo, habrá que añadir dos printf, uno haciendo uso de la función getpid(), para imprimir su pid y otro a getppid() para imprimir el de su padre.

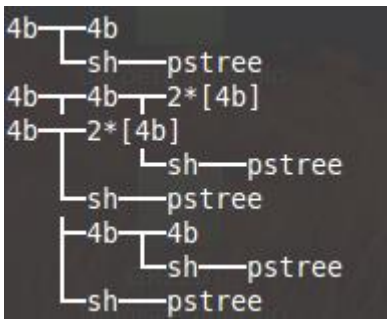
En ambos programas es posible que queden procesos huérfanos. En el programa del apartado a) no se realiza ningún wait, por lo tanto, aparte de que todos los hijos quedarán zombies, no hay forma de controlar que los procesos hijos acaben antes que sus padres, pues no hay ningún mecanismo para bloquear los procesos padre hasta que sus hijos finalicen.

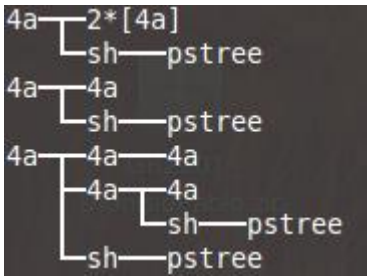
En el segundo programa también se generarán procesos huérfanos, aunque con menos probabilidad que en el primero, pues los procesos no realizan el mismo número de llamadas a wait como forks realizan. Los procesos generados en la última iteración del bucle si serán esperados y no quedarán zombies, pero no se limpiarán otros porcesos generados en forks anteriores de la tabla de procesos, y no hay garantía de que finalicen antes que sus padres, quedando huérfanos. Cabe destacar que los procesos hijos generados en el fork de la última iteración del bucle realizan un wait sin haber hecho un fork.

Al ejecutar ambos programas, en algunos casos se observa que el pid de los padres de algunos procesos es 1, esto es debido a que se han quedado huérfanos y el init los ha apadrinado.

A continuación se muestran algunos árboles generados por pstree, a partir de los cuales se ha sacado el análisis anterior.

```
|-gnome-terminal-+-bash---4a-+-4a-+-4a---4a
|                |           |    `-4a
|                |           |-4a---4a
|                |           |-4a
|                |           `-sh-+-less
|                |                `-pstree
```





En total se realizan 7 forks, generando cada uno su correspondiente proceso hijo.

Ejercicio 5

a) Para que el conjunto de procesos se genere de forma secuencial, basta con introducir la llamada a la función *wait()* dentro de la rama de ejecución del padre, de esta manera cuando se realiza un *fork()* el proceso padre entra en estado de bloqueado, esperando al hijo que ha lanzado, el cual a su vez seguirá lanzando procesos de la misma forma hasta agotar el bucle *for*.
Como cada proceso solo debe generar un único hijo, en la rama del padre, tras la llamada a wait() se realiza un break para salir del bucle, evitando que el proceso que ha lanzado ya un hijo vuelva a lanzar otros hijos después de salir del estado de bloqueado.
De esta forma se generará una secuencia de procesos de longitud NUM_PROC.

b) Para que el proceso padre lance varios hijos y espere a todos ellos basta con añadir un break, al código del ejercicio 4, en la rama de los procesos hijos al realizar un *fork(),* para que salgan del bucle y no sigan generando procesos, y en la rama del padre hacer una llamada a la función *wait()* para que espere a los procesos lanzados.
En el enunciado no se especifica si el proceso padre ha de esperar a los procesos hijos nada más lanzarlos o después de acabar de crear todos, por lo tanto se ha optado por esperarlos nada más lanzarlos.

Ejercicio 6

El proceso padre no tendrá acceso al valor recibido por el proceso hijo, debido a que se guarda en una variable accesible únicamente desde el proceso hijo.

Se deberá liberar memoria en ambos procesos, porque al realizar la llamada *fork*() se realiza una copia exacta del proceso padre, con todas sus variables, por lo tanto volviéndose a reservar memoria para la cadena.

Ejercicio 8

a) Al ejecutar el comando du mediante la llamada a una función de la familia *exec*, se le pasa como nombre del ejecutable a mostrar su tamaño “ejercicio8”, nombre generado por el *makefile*. Se debe respetar este nombre para que funcione correctamente.

b) Para lanzar el comando pasado como argumento en foreground hay que realizar una llamada a una función exec y el nuevo código sustituirá al del ejercicio, únicamente en caso de error seguirá ejecutándose el código del ejercicio, mostrando un mensaje de error y devolviendo EXIT_FAILURE.

   Para lanzar el comando en background se realiza un fork y es el proceso hijo el encargado de hacer un exec y ejecutar el comando.
   En el proceso padre no puede hacerse un wait() para evitar que el proceso hijo quede zombie, porque entonces se bloquearía, siendo justamente esto lo que queremos evitar al lanzar el comando en segundo plano, por lo que en el proceso hijo se hace un setsid() para desvincularlo del proceso padre.

Ejercicio 9

Este ejercicio requería crear a partir de un proceso padre, crear varios procesos hijos, y a partir de estos otros procesos hijos (también llamados nietos) y comunicar algunos mensajes a través de tuberías.

Para controlar las líneas de ejecución de cada proceso se han utilizado if's anidados que, en función del retorno de las funciones fork, ejecutan diferentes bloques para el padre, los hijos y los nietos.

Para realizar las comunicaciones se han utilizado tuberías. En total se necesitan 12 tuberías, sin embargo, cada proceso individual utiliza como máximo 6 tuberías (los hijos utilizan 2 para la comunicación con su padre y 4 para la comunicación con sus 2 hijos). Para los handlers de las tuberías se han utilizado matrices 2x2 de int's (int fd[2][2]) de tal forma que el primer número de la matriz indica si se trata del handler de escritura o de lectura, y el segundo indica si el flujo de datos la tubería va "hacia arriba" (de hijos a padres), o "hacia abajo" (de padres a hijos).

Por último, se han utilizado algunas variables para almacenar los datos recibidos por las tuberías, que después se impriman por pantalla como confirmación de que la comunicación entre procesos se ha realizado con éxito.

El código fuente de este ejercicio ha sido comentado muy detalladamente para facilitar su comprensión, ya que debido a su complejidad y sus múltiples líneas de ejecución, consideramos que era muy necesario detallar el rol de cada variable y de cada bloque de código.

Para probar el ejercicio, ejecutamos el programa con valgrind. También utilizamos el comando pstree. De esta manera comprobamos que el programa lanzaba los procesos adecuadamente y que hacia un buen uso de la memoria.