SOPER:
Práctica 1

Adrián Lorenzo Mateo, Ari Handler Gamboa
Grupo 221, Equipo 07

Ejercicios de procesos

1. Implementa un programa en C que use fork para crear el siguiente árbol de procesos. Ten en cuenta que no es necesario generar ningún tipo de representación gráfica, más allá de una frase indicando cuándo se ha creado un proceso, su pid y el pid de su padre.

Observa también que cada nodo es un proceso, y que el PID mostrado es únicamente orientativo.

¿Por qué crees que algunos nodos aparecen repetidos en el diagrama?

Porque los procesos siguen en ejecución desde su propia perspectiva.

2. ¿El orden de la salida al ejecutar el programa 1 es siempre el mismo? Justifica tu respuesta.

No, puesto que los procesos padre no esperan a que los hijos finalicen, "solapándose" unos con otros.

```
usuario@16-22-66-169:~/Desktop$ ./programa1
Raiz: PID=3754 PPID=3534
Proceso(GEN 1): PID=3755 PPID=3754
Proceso(GEN 2): PID=3756 PPID=3755
Proceso(GEN 2): PID=3755 PPID=3754
Proceso(GEN 1): PID=3754 PPID=3534
Proceso(GEN 2): PID=3757 PPID=3754
Proceso(GEN 2): PID=3754 PPID=3534
```

```
usuario@16-22-66-169:~/Desktop$ ./programa1
Raiz: PID=3758 PPID=3534
Proceso(GEN 1): PID=3759 PPID=3758
Proceso(GEN 1): PID=3758 PPID=3534
Proceso(GEN 2): PID=3761 PPID=3758
Proceso(GEN 2): PID=3758 PPID=3534
Proceso(GEN 2): PID=3760 PPID=3759
Proceso(GEN 2): PID=3759 PPID=1
```

Nota: En este último ejemplo el proceso con PID 3759 tiene de padre al proceso init (PID 1) ya que su padre ha muerto antes de terminar su propia ejecución, encargándose el SO de reasignarle como hijo del proceso inicial.

4. ¿Por qué puede fallar fork? ¿hay alguna forma de controlar el error de fork? Modifica el código de programa3.c para controlarlo.

Puede fallar si el kernel no ha podido asignar suficiente memoria para crear un nuevo proceso (ENOMEM) o si se ha llegado al número máximo de procesos del usuario actual o del sistema (EAGAIN).

Para controlar estos errores simplemente habría que poner en el switch un caso en el que fork devuelva el valor -1.

5. ¿Qué otro problema tiene el código del programa3.c? Soluciónalo escribiendo el código correcto de programa3.c como programa4.c

El último printf siempre se va a ejecutar sin tener en cuenta el tipo de proceso que lo llame.

Para arreglarlo, se debe añadir una nueva cláusula 'default' al switch(fork()).

6. ¿Cuánto vale la variable a para el código padre del programa4.c? ¿y el para el hijo? ¿por qué no valen lo mismo? ¿en algún momento valen lo mismo?

Para el código padre la variable 'a' siempre vale 1.

Para el hijo , 'a' vale 1 hasta que se accede al segmento de código del case en el que la función fork() devuelve 0, en ese momento vale 3.

7. Modifica el código de programa4.c como un nuevo programa5.c en el que el hijo devuelve el valor de la variable a para que el printf que hace el padre muestre el mismo valor que el printf que hace el proceso hijo.

Se ha cambiado el **break** del segmento de código del caso del hijo por un **exit(a)**, siendo el padre el que espere al retorno del hijo (mediante la sentencia wait()), asignando a la variable **a** el valor que espefica su hijo.

8. Explica las macros de wait en un ejemplo de programa padre (programa6.c) que cree un proceso, y lo espere con wait, ¿cómo afecta a wait el uso de las macros?

WIFEXITED(estado del hijo) devuelve 0 si el hijo ha terminado de una manera anormal (caida, matado con un kill, etc). Distinto de 0 significa que ha terminado porque ha hecho una llamada a la función exit().

WEXITSTATUS(estado del hijo) devuelve el valor que ha pasado el hijo a la función exit(), siempre y cuando la macro **WIFEXITED** indique que la salida ha sido por una llamada a exit().

9. Modifica programa6.c como programa7.c para que en lugar de usar wait, espere al proceso hijo usando waitpid. ¿Qué diferencias encuentras entre el uso de wait/waitpid?

La principal diferencia entre 'wait' y 'waitpid' consiste en la especificación del PID del hijo que debe esperar el padre.

10. Escribe el código de un programa8.c que reserve en el proceso padre memoria dinámica para una cadena de 20 caracteres de longitud y un proceso hijo. Si en el proceso hijo se pide al usuario que introduzca un nombre para guardar en la cadena. ¿El proceso padre tiene acceso a ese valor? ¿qué ocurre si el usuario no teclea nada? ¿dónde hay que liberar la memoria reservada y por qué?

La cadena de caracteres creada en el código del padre se copia en el del hijo, asignándola una nueva dirección de memoria. Por lo tanto, el hijo rellena su propia copia de la cadena, manteniéndose la del padre vacía.

La memoria reservada debe ser liberada justo antes de la salida del programa principal para que se liberen las dos copias de la cadena, la del padre y la del hijo.

Ejercicios de señales

11. Ejecuta el programa9.c, ¿la llamada a signal supone que se ejecute la función captura? ¿cuándo aparece el printf en pantalla? ¿por qué no aparece antes?

Sí, la función signal asigna a la señal SIGINT el manejador de señal captura. El printf aparece cuando se captura la señal.

No aparace antes porque captura sólo se pone en funcionamiento cuando se produce la señal SIGINT.

12. ¿Qué ocurre por defecto cuando un programa recibe una señal y no la tiene capturada? Demuestra tu respuesta modificando el código de programa9.c y guardando el código resultante como programa10.c

Se pone en funcionamiento el “signal handler” por defecto. En el caso de la señal SIGINT, el proceso se interrumpe automáticamente.

13. Escribe el código de un programa11.c que ignore la pulsación de Ctrl+C en el teclado. ¿Cómo podrías demostrar que realmente la señal ha sido ignorada?

Porque el proceso continua, sólo finaliza si hacemos Ctrl+Z y miramos en jobs el número de trabajo, matándolo con kill %n.

14. ¿Todas las señales tienen asociada una semántica previa? Si no es así, pon un ejemplo en un programa12.c de algún tipo de señal definida por el usuario

No, las señales de usuario SIGUSR1 y SIGUSR2 no la tienen. En el ejercicio propuesto se ha creado un proceso hijo que capture SIGUSR1 y es el padre el que se la envía.

15. Escribe el código de un programa13.c que intente capturar SIGKILL, y que escriba en la función manejadora “He conseguido capturar SIGKILL”. ¿Por qué nunca sale por pantalla “He conseguido capturar SIGKILL”?

Nunca sale por pantalla el mensaje “He conseguido capturar SIGKILL” puesto que ésta señal junto con la 19 son las únicas señales que no se pueden capturar.

16. ¿Por qué se permite el uso de variables globales para compartir datos entre la función manejadora y el resto del programa que tiene capturada una señal?

Es la única manera de compartir el uso de datos ya que el prototipo de las funciones manejadoras no permiten el uso de más argumentos que el número de la señal y su retorno siempre debe ser **void**.

17. ¿Qué diferencia hay entre usar pause o sigsuspend? Demuestra tu respuesta con dos ejemplos de código (programa14.c y programa15.c)

La sentencia **pause()** suspende el proceso hasta que éste reciba cualquier tipo de señal. En cambio, a **sigsuspend()** se le debe pasar cómo argumento una máscara de señales que indica las señales que debe esperar para reanudar la ejecución.

En ambos ejercicios se ha generado una alarma de 3 segundos. El proceso se suspenderá en ambos casos hasta que se reciba la señal SIGALRM.

18. Escribe un programa16.c que capture todas las señales capturables y sólo esas.

Se ha implementado un bucle **for** que recorra las 31 señales asignándo la misma función manejadora a todas. Se han desechado las señales 9 y 19 por ser las únicas que no se pueden capturar.

19. Mientras se ejecuta el código de la función manejadora del programa16.c asociada a SIGUSR1, envíale SIGUSR2 desde otra terminal, ¿qué sucede? ¿por qué?

No aparece el mensaje “"He conseguido capturar la señal (SIGUSR1)” porque el signal handler de esa señal se queda en un bucle infinito. Sin embargo la función manejadora consigue capturar SIGUSR2 demostrando que los signal handlers se quedan escuchando durante toda la ejecución del programa a las señales.

20. Escribe el código de un programa17.c que establezca una alarma inicial de 20 segundos en los que el programa debe mostrar por pantalla una secuencia numérica creciente separada por comas. Cuando salte la alarma, la última línea mostrada en la pantalla debe ser: "Estos son los números que me ha dado tiempo a contar en 20 segundos".

Primeramente se comienza una alarma de 20 segundos. Se define una función capturadora de SIGALRM que cambia el valor de la flag global que para el bucle que imprime la secuencia de caracteres si ésta flag vale 1.

EJERCICIO FINAL

El diseño realizado para éste ejercicio se basa en lo siguiente:

- Se pide al usuario la clave de cifrado Vignère y la frase que se desea codificar. Si en alguna de las dos capturas tarda mas de 60 segundos en realizarse, se ordena la terminación del proceso de forma inmediata mediante la señal SIGKILL (función manejadora `expira`).

- En este punto se crea el proceso cronómetro, el cuál define el comienzo de todos los procesos mediante la función `gettimeofday` (puesto que se crearan prácticamente en el mismo instante de tiempo).

  Éste proceso define 4 señales manejadoras, una para cada proceso del cuál debe medir su tiempo de ejecución:

  - SIGUSR1: Tiempo del codificador
  - SIGUSR2: Tiempo del contador de palabras
  - SIGFPE: Tiempo del contador de caracteres válidos.
  - SIGXFSZ: Tiempo del contador de caracteres no válidos.

  Además define la manejadora de SIGBUS para la impresión de los tiempos por pantalla mediante la suspensión del proceso con `sigsuspend` . Se utiliza la macro `DIFFTIME` definida en `utilidades.h` para la diferencia entre el tiempo final y el inicial.

- El proceso padre crea los 4 procesos hijos que manipularán la frase original. Cada uno de ellos llama a su correspondiente función definida en la librería `utilidades.c` , envíando sus respectivas señales de finalización al proceso cronómetro mediante la función `kill(PIDCronometro, señal)`.

  Éstos procesos hijos difieren en su forma de mostrar su resultado de la siguiente forma:
  - El proceso codificador llama a la función `vignere` pasándole cómo argumentos la función a codificar y la correspondiente clave de cifrado. El retorno de la función es impreso por pantalla directamente por éste hijo, finalizando así su ejecución.

- Tanto el proceso contador de palabras cómo el encargado de contar los caracteres válidos devuelven el resultado de sus respectivas operaciones mediante el retorno de su exit al proceso padre, el cuál se encargará de mostrarlos por pantalla.

- El proceso contador de caracteres que no se pueden cifrar utiliza el pipe definido al comienzo del código del proceso padre para escribir su resultado en una cadena de caracteres que recibirá éste al final del programa.

- Una vez creados éstos cuatro procesos hijos, el padre se dedicará a esperar la finalización de éstos mediante `waitpid` , recogiendo las salidas de los contadores de su retorno del exit y leyendo del pipe común el string de caracteres no válidos.

  Cuándo termine de mostrar los resultados correspondientes por pantalla, envía la señal correspondiente (`SIGBUS`) al proceso cronómetro para que imprima sus propios resultados, finalizando el programa una vez que éste ha terminado.

Hasta aquí el funcionamiento básico del programa principal, ahora se mostrará una breve explicación de las funciones auxiliares más importantes definidas en la librería `utilidades.h`.

La función `cuentachar(char *cadena)` recorre la cadena pasada cómo argumento contando los caracteres que se encuentren en el rango [A...Z]U[a...z]U[ñ,Ñ]. La función `noencripchar(char *cadena)` tiene una funcionalidad análoga, radicando su diferencia en el conteo de caracteres que se encuentran en la transposición del rango antes mencionado.

`cuentapalab(char *cadena)` devuelve el número de palabras de la cadena mediante llamadas sucesivas a `strtok` , siendo los espacios los delimitadores entre palabras.

La función `vignere(char *claveCifrado,char *FraseOriginal)` codifica mediante éste método la frase original carácter a carácter si éste es válido, si no lo es se deja cómo estaba.

Sobre la aplicación finalizada se han realizado las siguientes pruebas:

- Se ha introducido cómo clave de cifrado 'Kripto.', el programa notifica al usuario que ha introducido caracteres no validos en dicha clave y la vuelve a pedir.
- Si se introducen más de una palabra cómo clave ocurre lo mismo que en la prueba anterior.
- En cualquier momento del programa se puede finalizar su ejecución mediante el atajo CTRL + C.
- Si pasan más de 60 segundos durante la captación de cadenas con las que trabajar, se finaliza el programa.