Grupo 8
SOPER P2

Eduardo Terrés Caballero
Gregorio Blázquez Martínez

PRÁCTICA 2

Ejercicio 1: Comando kill de Linux.

a) Con el comando `kill -l` se muestran las distintas señales que puede enviar kill:

```
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

b) Con el comando `kill -l SIGKILL` se obtiene que la señal *SIGKILL* tiene número 9. Mediante `kill -l SIGSTOP`, se obtiene que *SIGSTOP* tiene número 19. La tabla también da la correspondencia de números y señales.

Ejercicio 2: Envío de señales.

a) Se adjunta el archivo *sig_kill.c* con el código añadido. El código añadido es el siguiente:

```
if(kill(pid, sig) == -1){
        perror("kill");
        exit(EXIT_FAILURE);
}
```

b) Al enviar la señal *SIGSTOP* a la otra terminal, se puede escribir en ella, pero no se mostrará el resultado de la escritura. En su defecto, se almacenan los comandos escritos durante el estado de suspensión en el buffer. De esta forma, al enviar la señal *SIGCONT*, se ejecutarán todos los comandos escritos – en caso de haber incluido *enter* -.

Ejercicio 3: Captura de señales.

a) No, la llamada a *sigaction* no supone la ejecución de *manejador*. De esta forma, al intentar añadir código que da lugar a error de ejecución, no fallará hasta haber recibido una señal de tipo *SIGINT* - aunque le enviemos otra señal no se ejecutará -.

b) La única señal que se bloquea durante la ejecución del manejador es *SINGINT*. Añadiendo un *sleep* a *manejador*, al enviar más señales de tipo *SIGINT* durante el tiempo que duerme el proceso, se esperará a la finalización de la ejecución del manejador. De esta forma, si se introducen 5 *SIGINT* solo se guarda uno de ellos. Otras señales no quedan bloqueadas, ya que la máscara de señales de la estructura *sigaction* se vació. Así pues, si se añaden nuevas llamadas a *sigaction* con otras señales, al enviar el segundo tipo de señal tras enviar *SIGINT*, pasa a ejecutarse su manejador, ignorando el posible tiempo de espera restante por parte de *manejador*.

c) Inmediatamente después de mandar la señal - mediante ctrl + C o el programa *sigkill* -.
d) Mediante el siguiente cambio en el código, se consigue que el programa ignore la señal:

```
act.sa_handler = SIG_IGN;
```

Al enviar la señal *SIGINT* el programa no reacciona, la ignora. Cuando un programa recibe una señal sin tener el manejador instalado se ejecutará el manejador por defecto. El efecto es análogo al siguiente cambio en el código:

```
act.sa_handler = SIG_DFL;
```

e) No se pueden capturar todas las señales. En concreto, las señales *SIGKILL* (9) y *SIGSTOP* (19) no se pueden capturar.

Ejercicio 4: Captura de SIGINT mejorada.

a) Aunque cuando le llegue en cualquier momento se ejecutará el manejador, realmente la gestión de la señal se realiza en `if(got_signal)` , el cual se ejecutará si se ha activado con anterioridad el manejador poniendo *got_signal* a 1.

b) Porque en este caso la variable global es un indicador de si ha llegado la señal que estamos esperando, sirviendo de "mensajero" entre el manejador y el programa principal.

Ejercicio 5: Bloqueo de señales.

a) Nada, el programa bloquea las señales *SIGUSR1* y *SIGUSR2*. Al recibir *SIGINT* termina su ejecución, sin imprimir el último mensaje.

b) Al finalizar la espera se imprime el mensaje *"User defined signal 1"*. No se imprime el mensaje de despedida. Esto se debe a que, una vez restaurada una máscara de señales que no bloquea *SIGUSR1* – que se encontraba pendiente en estado de bloqueo -, se inicia su tratamiento – mediante el manejador por defecto, dado que no se especificó otro -. Se trata de una señal que termina el proceso, por lo que no se ejecuta la línea que imprime el mensaje de despedida.

Ejercicio 6: Gestión de la Alarma

a) El programa finaliza como si se hubiera acabado el tiempo de la llamada a *alarm*, y le hubiera llegado *SIGALRM* desde la función.

b) La alarma se gestiona mediante el manejador por defecto. Así pues, termina su ejecución y se imprime la línea "*Alarm clock*", o su traducción en español "*Temporizador"*.

Ejercicio 7: Creación y Eliminación de Semáforos

Se podría modificar la llamada a *sem_unlink*, poniéndolo tras la creación del semáforo – *sem_open* - , de forma que se incluye lo más temprano en el código. Esto es posible gracias a que el sistema operativo no liberará los recursos del semáforo hasta que haya un total de 0 procesos empleándolo. De esta forma, hasta la llamada a *sem_close* del último proceso - dependiente del orden de ejecución – no se eliminará el semáforo.

Ejercicio 8: Semáforos y Señales

a) Se ejecuta el manejador, y tras esto, continúa la ejecución en la siguiente instrucción – *printf* -. Entonces, la llamada a *sem_wait* no se ejecuta con éxito, ya que la espera inactiva ha sido interrumpida por la llegada de la señal. De esta forma, ha salido de la espera y el valor del semáforo sigue siendo 0.

b) Al ignorar la señal, el proceso no interrumpe la espera del *sem_wait*, y el proceso se mantiene en estado de bloqueo.

c) Habría que bloquear todas las señales, haciendo uso de *sigprocmask*, para desbloquearlas y tratarlas una vez sale de la espera inactiva inducida por *sem_wait*.

Ejercicio 9: Procesos Alternos

El proceso hijo no espera ni altera el valor de ningún semáforo para la impresión del primer número. La dinámica seguida consiste en que cada proceso siempre espera el mismo semáforo, y cada semáforo siempre es esperado por el mismo proceso. Asimismo, un proceso no puede llamar a *sem_post* para su semáforo asignado – es tarea del otro proceso -. En este caso particular, el semáforo de espera asignado para el proceso hijo es *sem1*; para el padre es *sem2*.

```
if (pid == 0) {
                printf("1\n");
                sem_post(sem2);
                sem_wait(sem1);
                printf("3\n");
                sem_post(sem2);

                sem_close(sem1);
                sem_close(sem2);
}
else {
                sem_wait(sem2);
                printf("2\n");
                sem_post(sem1);
                sem_wait(sem2);
                printf("4\n");

                sem_close(sem1);
                sem_close(sem2);
                sem_unlink(SEM_NAME_A);
                sem_unlink(SEM_NAME_B);
                wait(NULL);
                exit(EXIT_SUCCESS);
}
```

Ejercicio 10 f) : Concurrencia y Sincronización de Procesos

Analizar los mensajes que se imprimen por pantalla. ¿Se están ejecutando los procesos en el orden que les corresponde? ¿Hay garantías de que sea así?

Sí están en orden porque, al estar bloqueadas las señales, normalmente no hay un cambio de proceso entre el envío de la señal *SIGUSR1* a su proceso hijo y la impresión del ciclo. No hay garantías de que esto suceda así, pues podría darse el caso en que se cambie de proceso justo después de enviar la señal *SIGUSR1*, lo que permitiría al hijo ejecutar su ciclo – impresión incluida – antes de su padre.

Repetir el análisis anterior, pero tras introducir una espera aleatoria en cada proceso, después de enviar la señal al siguiente proceso pero antes de imprimir por pantalla. ¿Están ahora ordenados los mensajes? ¿Por qué?

Esta vez no se ejecutan en orden pues ahora los procesos envían las señales y, mientras están en el *sleep*, es frecuente que a algún proceso sucesor se le asigne tiempo de CPU y realice su ciclo antes que alguno de sus ancestros, provocando el desorden entre los mensajes impresos.

Explicación ejercicio 10

Nuestra solución hace uso de un semáforo binario, para garantizar la exclusión mutua en el envío de *SIGUSR1* y la posterior impresión del ciclo. De esta manera, solo un proceso puede enviar la señal a su hijo y se evita la condición de carrera. Previo a los *forks*, quedan bloqueadas las señales *SIGUSR1, SIGTERM y SIGINT* para evitar la posible pérdida de estas antes de comenzar los ciclos.

Para poder enviar la señal *SIGINT* desde terminal mediante *Ctrl + C*, esta se bloquea a todos los procesos, ya que no es seguro que solo el padre la reciba - y, por ende la gestione – si se envía por la terminal, mediante *Ctrl + C*.

Los manejadores modifican el valor de una variable global volátil para poder realizar una gestión síncrona de las señales, haciendo así que el ciclo sea robusto. Si en cualquier momento el programa fallara al llamar a alguna función - *sigaction, sigprocmask, kill, malloc* -, se ha implementado la función *killAll* y la gestión de *SIGUSR2* para que finalice sin procesos colgados ni hijos *zombies*.