Universidad Autónoma de Madrid

Departamento de Informática

Sistemas Operativos
Memorias Práctica - 2

Hecho por: Diego Rodríguez Ortiz

y Alejandro García Hernando

Introducción

En esta práctica hemos desarrollado un programa que simula votaciones. La idea del programa es lanzar múltiples procesos que competirán por proclamarse candidato. Tras nombrar a un candidato, se convoca una votación en la que cada uno votará de forma aleatoria y el candidato contará los votos y decidirá si ha sido elegido.

El programa tiene tres formas de acabar: por señal, por alarma o por error. En el primer caso, cuando el programa principal recibe una señal de SIGINT envía una señal SIGTERM para finalizar los procesos hijos de forma controlada. En el segundo, tras haber transcurrido el tiempo especificado se recibe un SIGALRM que de nuevo envía un SIGTERM a los procesos. Por último en caso de error en los hijos, los procesos hijos envían SIGUSR1 al principal para que finalice el programa de forma controlada.

Estructura de código

El proyecto contiene 8 ficheros: 5 de código y 3 auxiliares

-Makefile: contiene una serie de comandos para la compilación y ejecución del código. Además de un comando que permite comprobar que se han cerrado correctamente los semáforos y un segundo que los elimina en caso de que no se hayan cerrado correctamente.

– volt_mult.c: la función principal se encuentra aquí. Esta función recoge los parámetros introducidos por el usuario y crea los procesos votantes con la rutina fork(). Principal se encarga de enviar señales a los votantes (sus procesos hijos) y controlar el código de salida de los votantes. También prepara una alarma para finalizar los procesos una vez ha transcurrido el tiempo estipulado.

– vot_func.c y vot_func.h: son archivos creados para albergar las principales funciones del programa como son voters() y candidato(). Además funciones como create_sons y la función handle de los votantes.

– vot_utils.c y vot_utils.h: estos archivos cuentan con las funciones auxiliares que son llamadas desde los archivos ya mencionados. Las principales funciones de estos archivos son las dedicadas a la comunicación de errores en los hijos al proceso principal y a la liberación de recursos. También incluye algunas funciones utilizadas para facilitar que los procesos se desincronicen, y comprobar si el programa falla o por el contrario funciona correctamente.

Además contamos con 2 ficheros más, uno para almacenar los PIDs de los procesos votantes y otro para el registro de los votos.

Pruebas

Finalización correcta de todos los procesos

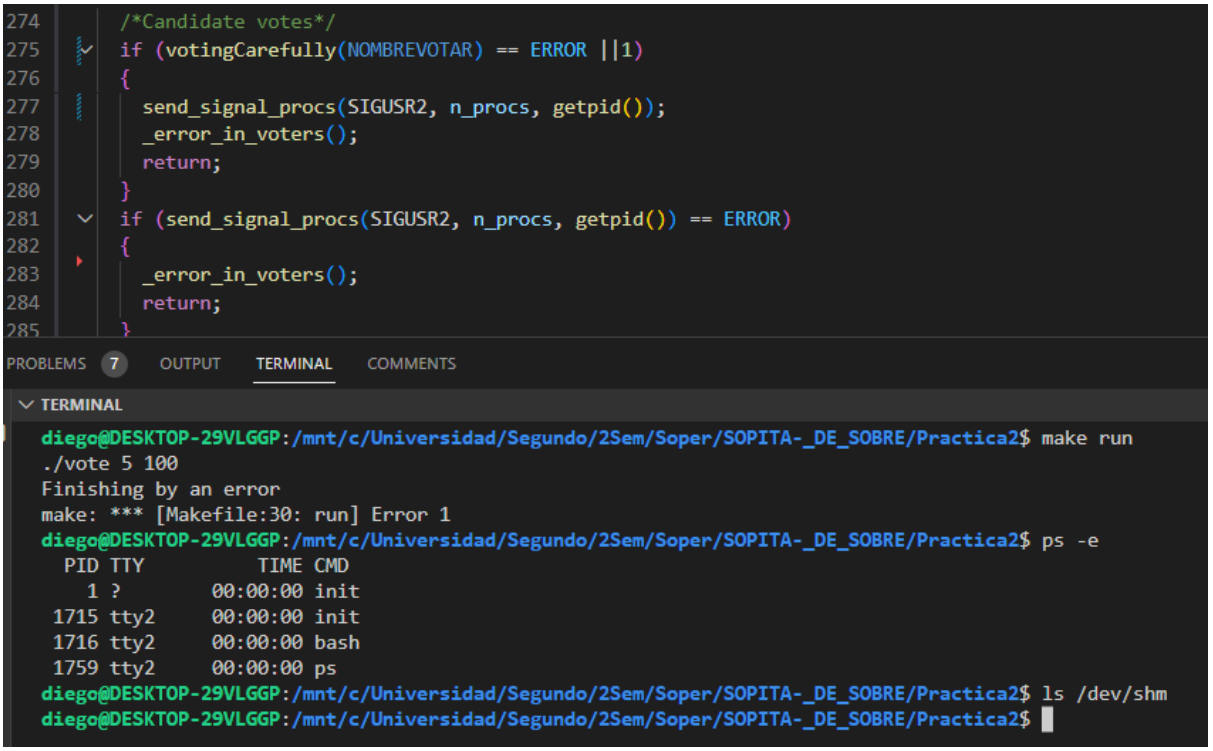
Los procesos finalizan correctamente si: terminan su ejecución cuando el padre se lo indica, sus códigos de retorno se recogen y se liberan todos los recursos utilizados (incluyendo los semáforos). Todo ello lo hemos comprobado cuando el programa se ejecuta el número máximo de segundos indicado y también cuando el usuario introduce por terminal el comando CTRL+C.

A continuación se muestra una imagen de las comprobaciones indicadas anteriormente para el caso de que el usuario introduzca CTRL+C:

Se puede ver que inmediatamente después de introducir CTRL+C los votantes paran de imprimir y acaban, mientras que el proceso principal imprime un último mensaje de salida y finaliza. Además, no quedan semáforos abiertos. Por último, comprobamos con el comando *ps -e* que no hay procesos zombies ni ejecutándose. Por ello, deducimos que la salida del programa ha sido exitosa.

Control de errores

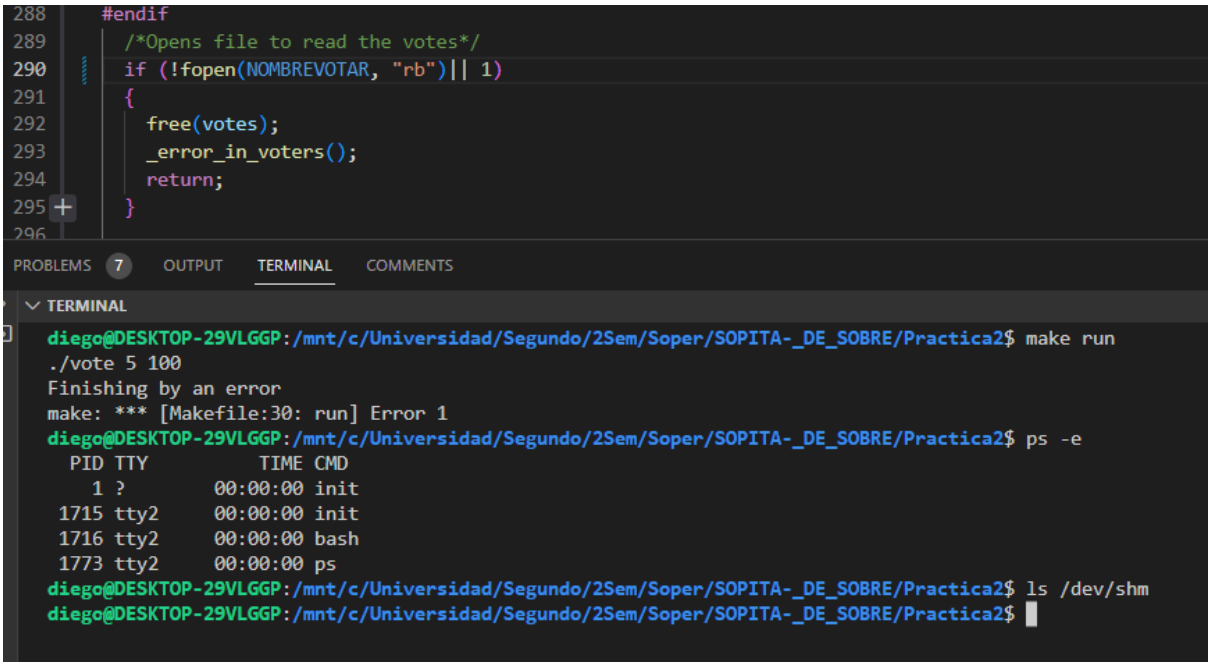
Dado que se trata de un ejercicio de gran complejidad, muchas cosas pueden fallar. Por ello, comprobamos que en caso de que algo falle, el programa es capaz de detectarlo y finalizar adecuadamente liberando todos los recursos. Para asegurarnos, hemos modificado algunas comprobaciones de error de la siguiente forma, haciendo que siempre se ejecute el protocolo de error:



```
274     /*Candidate votes*/
275     if (votingCarefully(NOMBREVOTAR) == ERROR ||1)
276     {
277         send_signal_procs(SIGUSR2, n_procs, getpid());
278         _error_in_voters();
279         return;
280     }
281     if (send_signal_procs(SIGUSR2, n_procs, getpid()) == ERROR)
282     {
283         _error_in_voters();
284         return;
285     }
```

```
PROBLEMS 7   OUTPUT   TERMINAL   COMMENTS

TERMINAL
diego@DESKTOP-29VLGGP:/mnt/c/Universidad/Segundo/2Sem/Soper/SOPITA-_DE_SOBRE/Practica2$ make run
./vote 5 100
Finishing by an error
make: *** [Makefile:30: run] Error 1
diego@DESKTOP-29VLGGP:/mnt/c/Universidad/Segundo/2Sem/Soper/SOPITA-_DE_SOBRE/Practica2$ ps -e
  PID TTY          TIME CMD
    1 ?        00:00:00 init
 1715 tty2     00:00:00 init
 1716 tty2     00:00:00 bash
 1759 tty2     00:00:00 ps
diego@DESKTOP-29VLGGP:/mnt/c/Universidad/Segundo/2Sem/Soper/SOPITA-_DE_SOBRE/Practica2$ ls /dev/shm
diego@DESKTOP-29VLGGP:/mnt/c/Universidad/Segundo/2Sem/Soper/SOPITA-_DE_SOBRE/Practica2$
```



```
288     #endif
289         /*Opens file to read the votes*/
290         if (!fopen(NOMBREVOTAR, "rb")|| 1)
291         {
292             free(votes);
293             _error_in_voters();
294             return;
295         }
296
```

```
PROBLEMS 7   OUTPUT   TERMINAL   COMMENTS

TERMINAL
diego@DESKTOP-29VLGGP:/mnt/c/Universidad/Segundo/2Sem/Soper/SOPITA-_DE_SOBRE/Practica2$ make run
./vote 5 100
Finishing by an error
make: *** [Makefile:30: run] Error 1
diego@DESKTOP-29VLGGP:/mnt/c/Universidad/Segundo/2Sem/Soper/SOPITA-_DE_SOBRE/Practica2$ ps -e
  PID TTY          TIME CMD
    1 ?        00:00:00 init
 1715 tty2     00:00:00 init
 1716 tty2     00:00:00 bash
 1773 tty2     00:00:00 ps
diego@DESKTOP-29VLGGP:/mnt/c/Universidad/Segundo/2Sem/Soper/SOPITA-_DE_SOBRE/Practica2$ ls /dev/shm
diego@DESKTOP-29VLGGP:/mnt/c/Universidad/Segundo/2Sem/Soper/SOPITA-_DE_SOBRE/Practica2$
```

En este caso se puede apreciar que el error ha sido detectado y se ha ejecutado el protocolo pertinente para finalizarlo. Al igual que en la ejecución sin errores, vemos que no quedan recursos sin liberar, pudiendo concluir que la salida ha sido exitosa.

Sleeps aleatorios

Esta otra prueba tiene como objetivo asegurarnos que no hemos realizado ninguna suposición de tiempo a la hora de programar. Para ello, hemos declarado una macro TEST que está destinada a que se ejecuten sleeps aleatorios llamando a la función propia nanorandsleep(). Estos sleeps aleatorios provocan que los procesos se desincronicen. Con esto intentamos que se den todos los posibles escenarios en cuanto a diferencia de velocidad de ejecución, lo cual no es posible. Por ello la robustez del programa no se puede asegurar. Sin embargo, creemos que con las pruebas y el estudio teórico realizado es sólido.

```
diego@DESKTOP-29VLGGP:/mnt/c/Universidad/Segundo/2Sem/Soper/SOPITA-_DE_SOBRE/Practica2$ make run
./vote 5 20
Candidate 2186 => [ Y N Y N Y ] => Accepted
Candidate 2190 => [ Y N Y N Y ] => Accepted
Candidate 2187 => [ Y N N N Y ] => Rejected
Candidate 2190 => [ Y Y Y N N ] => Accepted
Finishing by alarm
diego@DESKTOP-29VLGGP:/mnt/c/Universidad/Segundo/2Sem/Soper/SOPITA-_DE_SOBRE/Practica2$ make run
./vote 5 20
Candidate 2202 => [ Y N Y Y N ] => Accepted
Candidate 2202 => [ N Y Y Y N ] => Accepted
Candidate 2203 => [ Y N Y Y Y ] => Accepted
^C
Finishing by signal
```

```
139  void nanorandsleep(){
140      struct timespec time ={0,rand()%BIG_PRIME};
141      nanosleep(&time, NULL);
142  }
```

Control de que se acaba la votación aunque se reciba SIGTERM

```
diego@DESKTOP-29VLGGP:/mnt/c/Universidad/Segundo/2Sem/Soper/SOPITA-_DE_SOBRE/Practica2$ make run
./vote 10 10
v 990: Y
v 987: N
v 988: N
v 986: Y
v 991: Y
v 989: Y
v 992: Y
v 994: Y
^CSIGTERM 990
SIGTERM 986
SIGTERM 987
SIGTERM 988
SIGTERM 991
SIGTERM 989
SIGTERM 992
SIGTERM 993
SIGTERM 995
SIGTERM 994
v 993: N
v 995: N
Candidate 990 => [ Y N N Y Y Y Y Y N N ] => Accepted
Finishing by signal

diego@DESKTOP-29VLGGP:/mnt/c/Universidad/Segundo/2Sem/Soper/SOPITA-_DE_SOBRE/Practica2$ ls /dev/shm
diego@DESKTOP-29VLGGP:/mnt/c/Universidad/Segundo/2Sem/Soper/SOPITA-_DE_SOBRE/Practica2$ ps -e
  PID TTY          TIME CMD
    1 ?        00:00:00 init
  377 tty2     00:00:00 init
  378 tty2     00:00:00 bash
  997 tty2     00:00:00 ps
diego@DESKTOP-29VLGGP:/mnt/c/Universidad/Segundo/2Sem/Soper/SOPITA-_DE_SOBRE/Practica2$
```

Con el objetivo de asegurar que una votación que ya ha sido convocada se realiza sin errores. Para comprobar esto añadimos sleeps de tamaño considerable , 1 o 2 segundos, para que nos dé tiempo a introducir por pantalla el Ctrl+C en plena votación. Como se aprecia gracias a las impresiones por pantalla, se recibe SIGTERM cuando algunos votantes aún no han votado. También se ve que la votación se procesa con éxito. Además tanto los semáforos como los procesos se liberan correctamente.

Control de que todos los votantes entran a las votaciones

Tratando de evitar que casos extremos provoquen un mal funcionamiento del programa, contemplamos la posibilidad de que un votante se encuentre al comienzo del bucle principal de los votantes (línea 179), mientras que otro se encuentre justo antes de atender a la señal sigterm (línea 228). Si en este momento llega la señal SIGTERM, el segundo votante acabará su ejecución mientras que el primero tratará de hacer una votación más. En esta situación se produce un bucle infinito, pues el candidato de esta última votación estará esperando un voto por cada proceso, pero dado que el segundo votante ya ha finalizado esta espera jamás cesará.

```c
  while (1)
  { /*Main loop*/
#ifdef TEST
    nanorandsleep();
#endif

    /*Proposing a candidate*/
    if (down_try(semC) == ERROR)
    { /*Non candidate*/
#ifdef TEST
     nanorandsleep();
#endif
      while (!got_sigUSR2)
        sigsuspend(&oldmask);
      got_sigUSR2 = 0;
      /*Exclusion Mutua Votar*/
      while (down(semV) == ERROR);
#ifdef TEST
      nanorandsleep();
#endif
      votingCarefully(NOMBREVOTAR);
      if (up(semV) == ERROR){
        send_signal_procs(SIGUSR2, n_procs, getpid());
        _error_in_voters();
      }
    }
    else
    { /*Candidate*/
      candidato(n_procs, semCTRL);
#ifdef TEST
      nanorandsleep();
#endif
      /*Release the sem to choose a new candidate + send USR1 to start a new voting*/
      if (up(semC) == ERROR)
      {
        _error_in_voters();
      }

      if (send_signal_procs(SIGUSR1, n_procs, NO_PID) == ERROR)
      {
        _error_in_voters();
      }
    }
#ifdef DEBUG
    sem_getvalue(semCTRL, &aux);
    printf("PREUSR1 %d, %d\n", getpid(), aux);
#endif
#ifdef TEST
    nanorandsleep();
#endif
    while (!got_sigUSR1 && !got_sigTERM) /*Suspend the process waiting for SIGUSR1*/
      sigsuspend(&oldmask);
    got_sigUSR1 = 0;

#ifdef TEST
    nanorandsleep();
#endif
    if (got_sigTERM)
    {
      up(semCTRL);
#ifdef DEBUG
      printf("Hijo con PID-%ld sale por señal\n", (long)getpid());
#endif
      kill(getppid(), SIGUSR1);
      sem_close(semV);
      sem_close(semC);
      sem_close(semCTRL);
      exit(EXIT_SUCCESS);
    }
  }
```

Para solucionarlo hemos añadido un semáforo llamado *semCTRL*, que es creado con valor 0. Este semáforo tan sólo valdrá más que 0 cuando un proceso recibe SIGTERM y finaliza el programa. Por ello, cada vez que el candidato compruebe si han votado todos los votantes,

debe comprobar si *semCTRL* tiene un valor positivo, y en caso de que así sea, finalizar la votación pues habrá al menos un voto que no llegará.

Para simular esta situación, hemos decidido mandar a un votante la señal de SIGTERM, de este modo tan sólo él finalizará y el candidato debe darse cuenta de que falta un votante y finalizar la votación.

Resultado de la ejecución mostrado por terminal principal

Comando introducido por terminal secundaria

Como puede verse en la imagen superior, se detecta que el proceso con PID 1243 ya ha recibido la señal SIGTERM y finaliza el programa de forma efectiva. Además, el proceso principal comprueba si la señal SIGTERM ha sido mandada por él (salida exitosa), o por el contrario la ha mandado otro proceso (salida por error), e imprime por pantalla el correspondiente mensaje de finalización.

Análisis de la ejecución

¿Se produce algún problema de concurrencia en el sistema descrito?

Sí, se producen multitud de problemas a continuación describimos algunos ejemplos:

-Escritura simultánea en el fichero de votación: si dos procesos votantes abren el fichero de votación simultáneamente, cuando escriban su voto en él y posteriormente cierran el programa, sólo quedará guardado uno de ellos. Cuando todos los procesos hayan votado, en el fichero faltará al menos un voto, provocando que el candidato jamás salga del bucle que comprueba si están todas las votaciones y siendo imposible que el programa finalice.

-Elección del candidato: supongamos que el candidato es aquel que sea capaz de mandar una señal a todos los votantes antes que el resto. Si dos procesos se ejecutan paralelamente sin coordinación por semáforos, es posible que dos procesos comprueben simultáneamente si les ha llegado la señal y si son los primeros no les habrá llegado ninguna, por ello comienzan a mandar la señal. De este modo, ambos procesos se mandan mutuamente la señal y consideran que ya hay un candidato, por ello llegamos a una situación en la que no tenemos ningún candidato y no lo habrá, pues todos los votantes creen que existe y están a la espera de que indique el comienzo de la votación.

-Salida de proceso antes de la nueva votación: supongamos que nos encontramos en la situación descrita en la última prueba de este informe (control de que todos los votantes entran a las votaciones). Esta situación no se puede resolver de una forma segura con señales, y siempre cabría la posibilidad de que el programa acabe en un bucle infinito.

¿Es sencillo, o siquiera factible, organizar un sistema de este tipo usando únicamente señales?

No es nada sencillo organizar un sistema de este tipo usando únicamente señales. Se podría intentar utilizar al proceso principal como proceso coordinador, pero sería mucho más ineficiente y difícilde implementar. Además, el acceso al fichero de votaciones y la elección del candidato serían rutinas muy complejas, que si se utilizan semáforos se convierten en apenas un par de líneas.

Conclusión

En esta segunda práctica, hemos aprendido que la creación y coordinación de procesos que se ejecutan de forma paralela es imprescindible para construir un programa de votación como el propuesto. Para evitar condiciones de carrera hay que manejar correctamente tanto las señales como los semáforos, con los cuales nos hemos familiarizado a lo largo de este proyecto. Además, cabe resaltar que uno debe dejar de pensar en los programas como ejecuciones secuenciales y tratar de contemplar todas las posibles situaciones en las que se ejecutará el código, pues si no se hace correctamente es posible que existan situaciones límite que puedan provocar resultados indeseados.