Procesos, memoria y concurrencia

Informe práctico

Sistemas operativos

Integrante:

Herbert Mayorga - herbert.mayorga@alu.ucm.cl

Docente:

Felipe Tirado - ftirado@alu.ucm.cl

20 Mayo, 2024

1 Entorno y entrega

Tomaremos como referencia la versión Linux Debian 64 bits del laboratorio de prácticas, aunque la mayoría de los ejercicios funcionan igual en otras distribuciones como Ubuntu 18.04 o 20.04.

Deberían contestar las respuestas a las preguntas realizadas en este enunciado de forma ordenada justificando cada respuesta en un documento con el nombre de los componentes del grupo y entregarlo antes de que se termine el plazo de entrega de la práctica en el campus virtual.

2 Objetivos y habilidades

- Conocer las llamadas al sistema para gestionar procesos y tuberías.
- Conocer los mecanismos básicos de sincronización entre procesos (IPC).
- Desarrollar programas concurrentes con procesos.

3 Ejercicio 1

En este primer ejemplo (`'pexample1.c'`) aprenderemos como crear procesos y esperar a que finalicen. En el siguiente código podemos ver como el padre crea un proceso hijo utilizando la llamada al sistema a `fork()`.

Se imprime por pantalla su **timestamp** y espera a que el hijo termine su ejecución llamando a `wait()`. Por su lado, el hijo también imprime su **timestamp** y termina su ejecución llamando a `exit()`.

Se pide **analizar, compilar y ejecutar el código** `'pexample1.c'` para poder **contestar a las siguientes preguntas**.

```c
int main(int argc,char** argv) {
  int status;
  int pid;
  // Create a process (fork)
  if ((pid=fork())==0) {
    // Child process
    printf("[%s] Child process  (PID=%d)\n",timestamp(),getpid());
    exit(0); // Terminate OK
  } else {
    // Father process
    printf("[%s] Father process (PID=%d)\n",timestamp(),getpid());
    wait(&status); // Wait for child to finish
  }
  return 0; // Return OK
}
```

3.0.1 | Respuestas - Ejercicio 1

Preguntas para contestar en el informe de prácticas. Responde a cada pregunta en el informe intentando explicar el por qué de tus respuestas.

- **¿Comparten el padre y el hijo el mismo código del programa? ¿Ejecutan el padre y el hijo las mismas líneas del código? ¿Cómo discriminamos qué partes del código ejecuta cada proceso?**

  El proceso padre y el hijo comparten en el mismo código del programa y se ejecutan simultáneamente. Sin embargo, debido al uso de condicionales, el padre y el hijo ejecutan diferentes bloques de código del mismo programa. Podemos decidir que parte del código ejecuta cada proceso condicionando según el estado de PID que entrega el `fork()`, el hijo siempre devolverá un **código 0**, mientras que en el proceso padre devolverá el PID del hijo, siendo diferente de 0.

- **¿Qué función realiza la función getpid()? ¿Puede getpid() devolver el mismo resultado a dos procesos distintos? ¿Puede devolver cero? Describe qué pruebas has realizado para comprobarlo.**

  La función de `getpid()` devuelve un identificador del proceso que se esta ejecutando en ese momento, por lo tanto, es imposible que de el mismo resultado a dos procesos distintos y mucho menos 0, dado que el **PID 0** es reservado para ayudar en el arranque del sistema operativo en **sistemas UNIX**.

  Es bastante fácil comprobarlo creando una bifurcación de procesos con `fork()` e imprimir los valores de `getpid()` de cada proceso, comprobando que nunca son iguales y siempre se generan de forma secuencial según jerarquía padre e hijo.

- **¿Qué sucedería si el hijo llamara a wait()?**

  No sucederá nada, dado que el `wait()` está pensado en bloquear al padre para esperar a los procesos hijos. Sin embargo, si el proceso hijo no tiene procesos hijos propios, la función `wait()` ignorará su llamado y los procesos se ejecutarán normalmente de forma simultanea.

- **¿Es cierto que, dado que el padre crea al hijo con fork(), el padre siempre se ejecuta primero? ¿Cómo lo puedes comprobar?**

  No es cierto, al llamar a la función `fork()` se crea una bifurcación en el proceso y se crea el proceso hijo junto con el proceso padre, dichos procesos se ejecutarán de forma simultanea y dependerá del sistema operativo y el bloque de código a ejecutar de cada proceso cual terminará primero.

Esto se puede comprobar tomando el tiempo de ejecución con **timestamp** de los dos procesos sin utilizar `wait()` y con un bloque de código prácticamente igual para que no haya diferencias por CPU.

```
if ((pid = fork()) == 0) {
    // Child process
    printf("[%s] Child process  (PID=%d)\n", timestamp(), getpid());
    gettimeofday(&end, NULL);
    TimeP = (end.tv_sec - start.tv_sec) * 1000.0;
    TimeP += (end.tv_usec - start.tv_usec) / 1000.0;

    printf("Tiempo transcurrido: %.2f milisegundos\n", TimeP);
    exit(0);
} else {
    // Father process
    printf("[%s] Father process (PID=%d)\n", timestamp(), getpid());
    gettimeofday(&end, NULL);
    TimeP = (end.tv_sec - start.tv_sec) * 1000.0;
    TimeP += (end.tv_usec - start.tv_usec) / 1000.0;

    printf("Tiempo transcurrido: %.2f milisegundos\n", TimeP);
}
```

3.0.2 | Código a desarrollar - Ejercicio 1

Código para desarrollar y presentar en el informe con los comentarios necesarios

Extender el código anterior para **crear 50 procesos hijos**. Cada proceso hijo tiene que imprimir su **timestamp** y terminar. El proceso padre deberá esperar la finalización de todos los procesos hijos y, solo entonces, terminar.

- **¿Qué se observa en los resultados al ejecutar múltiples veces el programa?**

  Al ejecutar el programa múltiples veces, se observa que se ejecutan 50 procesos, cada uno con un PID diferente, sus últimos números siempre son valores cercanos o consecutivos, por lo que el número del PID no es al azar si no, que el sistema operativo les asigna PIDs secuenciales.

  Debido al **timestamp** se puede apreciar que la ejecución es simultanea y se imprimen prácticamente al mismo tiempo por diferencia de milisegundos.

- **¿Genera siempre los mismos resultados?**

  No genera siempre los mismos resultados, los PID siempre son diferentes. Dado que son identificadores únicos, como la palabra lo índica, sirven para identificar los procesos y es imposible que se clonen.

- **¿Se ejecutan los procesos siempre en el mismo orden? ¿Cómo lo has comprobado? ¿Por qué pasa esto?**

  Nunca se ejecutan los procesos en el mismo orden, esto se ve en el orden de impresión de los PID. Esto se debe al planificador que asigna un intervalo de tiempo de CPU de ejecución a los procesos y el orden siempre dependerá de la carga del sistema y prioridad de procesos. Aparte de esto, teniendo en cuenta que los procesos solo imprimen un mensaje, el tiempo de ejecución es muy corto y hay más posibilidad de que se entrelacen y no sea de forma secuencial.

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>

char* timestamp() {
    char* timestamp_buffer = malloc(100);
    struct timeval tv;
    gettimeofday(&tv, NULL);
    strftime(timestamp_buffer, 100, "%Y:%m:%d %H:%M:%S",
     localtime(&tv.tv_sec));
    sprintf(timestamp_buffer + strlen(timestamp_buffer), ":%ld",
    tv.tv_usec);
    return timestamp_buffer;
}

int main(int argc, char** argv) {
    int status;
    pid_t pid;
    pid_t pids_hijos[50];

    // Imprimir el timestamp del padre una sola vez
    printf("[%s] Proceso padre (PID=%d)\n", timestamp(), getpid());
```

```
    // Se crean 50 procesos hijos dentro de un bucle
    for (int i = 0; i < 50; i++) {
        pid = fork();
        if (pid == -1) {
            printf("Error al crear el proceso hijo");
            exit(1);
        } else if (pid == 0) {
            // Proceso hijo
            printf("[%s] Proceso hijo (PID=%d)\n", timestamp(), getpid());
            exit(0);
        } else {
            // Se guardan los PID de los hijos en el proceso padre
            // fork() devuelve 0 en el hijo y el PID del hijo en el padre
            pids_hijos[i] = pid;
        }
    }

    // Esperamos que finalicen todos los hijos
    for (int i = 0; i < 50; i++) {
        waitpid(pids_hijos[i], &status, 0);
    }

    printf("[%s] Proceso padre finalizado\n", timestamp());

    return 0;
}
```

4 Ejercicio 2

En el siguiente ejemplo (‘pexample2.c’) se muestra como ambos procesos padre e hijo imprimen por pantalla el valor de la variable ‘number’. **Se pide analizar, compilar y ejecutar el código** ‘pexample2.c’ para poder **contestar las siguientes preguntas**.

```
int main(int argc,char** argv) {
    int pid, status, number = 300;
    // Create a process (fork)
    pid=fork();
    if(pid==0) {
        // Child process
        number = 400;
        printf("Child process: Number is %d\n",number);
        // Terminate OK
        exit(0);
    } else {
        // Father process
        number = 500;
        printf("Father process: Number is %d\n",number);
        // Wait for child to finish
        wait(&status);
}
return 0; // Terminate OK
```

4.0.1 | Respuestas - Ejercicio 2

Cuestiones para resolver y describir el resultado obtenido y su justificación en la memoria final.

- **¿Imprimen los dos procesos el mismo número por pantalla? ¿Por qué?**

  No se imprimen los mismos números, dado que los procesos ejecutan un diferente bloque de código del mismo programa, dado que se condicionan en el caso de ser un proceso padre o hijo.

- **¿Qué sucedería si ambas llamadas a printf() se realizaran antes de la asignación a 'number'? ¿Por qué?**

  Los dos imprimirán el **número 300**, dado que la asignación de valor 300 de number se encuentra en la función principal del programa, función que ejecutan los dos procesos por igual antes de hacer una bifurcación de procesos y ejecuten diferente bloque de código según si es un proceso padre o hijo.

- **¿Depende el resultado de la ejecución del programa del orden en que los dos procesos asignan valor a la variable 'number'? ¿Por qué?**

  El orden en que se realicen las asignaciones a number en el padre y el hijo no afecta el resultado de la ejecución del programa, ya que las variables son independientes en cada proceso. Esto se debe a que, después de la llamada a `fork()`, cada proceso tiene su propia copia de la variable number en su propio espacio de memoria.

- **¿En qué punto del programa se reserva espacio para cada variable 'number'?**

  El espacio para la variable global number se reserva cuando se crea el proceso padre original al iniciar el programa. Esto ocurre antes de llamar a `fork()`. Mientras que en el proceso hijo al llamar a `fork()`, se crea una copia exacta de todas las variables del proceso padre, ahí mismo reservando un espacio de memoria para number en proceso el hijo.

- **¿Es necesario más de una CPU para ejecutar el código anterior? En caso negativo, describe cómo se ejecutarían los dos procesos en un procesador con 1 sola CPU. Por ejemplo, puedes explicar las instrucciones que se van ejecutando de cada proceso.**

  No es necesario más de una CPU para ejecutar este código. Debido a la multiprogramación y uso del planificador de procesos, es posible utilizar un solo procesador en una sola CPU intercalando las tareas que tiene cada proceso asignándoles un pequeño intervalo de tiempo de CPU, creando una ilusión de concurrencia.

1. Se ejecutan las instrucciones iniciales del proceso padre.
2. Se crea el proceso hijo usando la llamada `fork()` hecha por el padre.
3. El planificador asigna un tiempo corto al proceso hijo.
4. El proceso hijo imprime el mensaje `"Child process: Number is 300"` y sale.
5. El planificador reanuda la ejecución del proceso padre.
6. El proceso padre imprime el mensaje `"Father process: Number is 500"`.
7. El proceso padre espera al hijo con `wait()`.
8. El proceso hijo termina de ejecutarse.
9. El planificador reasigna el procesador al proceso padre para que termine su ejecución.

5 Ejercicio 2

Podemos ver como la pregunta natural que se nos plantea después de analizar el ejemplo anterior es cómo podemos comunicar datos entre dos procesos. Para ello analizaremos el ejemplo `'pexample3.c'`.

Este programa **crea un pipe (unnamed pipe)** y posteriormente llama a `fork()` creando un proceso hijo. De esta forma, ambos procesos tienen acceso a los extremos de escritura y lectura del pipe. En el ejemplo, el padre escribe en el pipe un dato de tipo entero, mientras que el hijo espera leyendo del otro extremo del pipe a que el padre le envíe el dato. Puedes revisar la teoría sobre comunicación de procesos usando pipes en los puntos 6.6.4 y 6.6.10 del libro Sistemas Operativos de Pedro de Miguel

Se pide analizar, compilar y ejecutar el ejemplo para poder contestar las siguientes preguntas.

```c
int main(int argc,char **argv) {
    int fd[2];
    int pid, status;
    // Create an unnamed pipe (store pipe descriptors into fd)
    pipe(fd);
    // Fork
    if ((pid=fork())==0) {
        // Child process
        printf("Child process: Created\n");
        // Read integer from pipe
        int number = 0;
        read(fd[0],&number,sizeof(int));
        // Print number
        printf("Child process: Number read %d\n",number);
        // Terminate OK
        exit(0);
    } else {
        // Father process
        int number = 900;
        // Write integer into the pipe
        printf("Father process\n");
        write(fd[1],&number,sizeof(int));
        printf("Father process: Number written\n");
        // Wait for child to finish
        wait(&status);
    }
    return 0; // Terminate OK
}
```

5.0.1 | Respuestas - Ejercicio 3

- **¿Qué sucede si el padre envía el dato antes que el hijo este leyendo desde el otro extremo? ¿Se pierde el dato? ¿Salta un error?**

  El valor seguirá intacto en el buffer durante su ejecución y se escribirá en la tubería, manteniéndose ahí hasta que el hijo lo lea. No se generará ningún error, ya que este es el comportamiento esperado por el pipe.

- **¿Qué sucede si el hijo quiere leer el dato antes que el padre haya escrito en el otro extremo? ¿El hijo ignorará la llamada a read()?**

  La llamada a `read()` se bloqueará y el proceso se mantendrá en espera hasta que haya un dato dentro de la tubería que sea posible leerlo.

- **¿Qué le sucederá al padre si la pipe está llena? ¿Qué le sucederá al hijo si la pipe está vacía?**

  Si la tubería esta llena la operación bloquea el proceso escritor. Sin embargo, haciendo pruebas en C, cuando el padre intenta escribir más datos de los que caben en la tubería no se bloquean los procesos, los datos se truncan y solo se escriben en la tubería los bytes que caben en el buffer y lo demás se rellena con caracteres basura.

  ```
  Child process: Data read Hola MundoHS
  ```

  la llamada bloquea el proceso hasta que algún proceso escriba datos en la misma. Sin embargo, en las pruebas, mientras que en el proceso hijo se leen todos los datos disponibles en la tubería y los bytes restantes del buffer se rellenan con caracteres especiales.

  ```
  Child process: Data read Hola Mundo
  ```

  Esto pasaría en un ejemplo con un buffer de tipo char, mientras que en un buffer de tipo entero, si intentas poner un valor mayor a 2.147.483.647 pasa lo mismo, sin embargo, se imprime un número negativo: `-2146483649`.

- **¿Qué sucede si intentamos leer de una pipe cuyo otro extremo se ha cerrado? ¿Qué sucede si intentamos escribir en una pipe cuyo otro extremo se ha cerrado?**

  Si se intenta leer de una tubería cuyo otro extremo se ha cerrado, `read()` devolverá 0 bytes leídos, indicando que no hay más datos disponibles. Esto se considera una condición de fin de archivo.

Si se intenta escribir en una tubería cuyo otro extremo se ha cerrado, se generará una señal `SIGPIPE` y se terminará el proceso.