**Sistemas Operativos**

*Prácticas curso 2021-2022*

Programación con llamadas al sistema: Práctica 2

1. ¿Qué llamada al sistema permite crear procesos? Explica el valor retornado por esta llamada al sistema.

La llamada al sistema que permite crear procesos es ‘fork()’. Fork retorna el valor del proceso hijo si la llamada tuvo éxito desde el proceso padre, y 0 si se ejecuta desde el proceso hijo.

1. Observa cómo se crean procesos hijo en los dos programas de la transparencia 34. Posteriormente, escribe (y luego entrega) el programa indicado en la transparencia 35 e indica cómo se crean los procesos hijos (de forma análoga a como se indica en la transparencia 34) y explica por qué. ¿Qué pasaría si los procesos hijo comenzaran su ejecución desde el inicio del programa (inicio de la función ***main***)?
2. int main() {
3. for (int i=0; i < 3; i++) fork();
4. cout << "Proceso " << getpid() << "; padre = " << getppid() << endl;
5. }

En el ejemplo, el proceso padre genera tres hijos mediante ‘fork()’, por lo que tiene 3 hijos. El primer hijo tendrá dos hijos, el segundo dos, y el último uno, puesto que no se repiten las mismas instrucciones al clonarse los procesos.

1. Escribe (y posteriormente entrega) el programa indicado en la transparencia 36. ¿Qué variables son compartidas entre el proceso padre (proceso inicial) y sus hijos? Explica cómo están relacionadas las variables en el proceso padre y en sus procesos hijo. ´
2. int main() {
3. int var;
4. pid\_t pid;
5. cout << "Comienzo la ejecución" << endl;
6. var = 88;
7. if ((pid = fork()) == EXIT\_FAILURE) exit(EXIT\_FAILURE);
8. if (!pid) { glob++; var++; }
9. else *// Se asegura que padre espera a que termine hijo*
10. sleep(2);
11. cout << "pid = " << getpid() << ", glob = " << glob
12. << ", var = " << var << endl;
13. exit(EXIT\_SUCCESS);
14. }

Ambas variables son compartidas, es decir, tanto `glob` como `var`, pese a que `glob` sea una variable global y `var` una local.

Al final de la ejecución, ambas variables están desincronizadas porque el programa le suma uno al valor de cada variable cuando el hijo ejecuta la condición, mientras que el padre “duerme”, por lo que al final tienen valores diferentes. El hecho de que ambas variables posean valores diferentes significa a su vez que las variables del hijo son una copia de las variables del padre y no son el mismo objeto ni apuntan a la misma posición en memoria.

1. ¿Por qué la llamada **exec**() nunca regresa en caso de éxito? ¿Y por qué si lo hace en caso de fallo? Implementa (y posteriormente entrega) un comando ***listado*** que ejecute un “**ls –l**”

La variable nunca regresa en caso de éxito porque el código del proceso actual es completamente reemplazado por el código del proceso indicado en `exec()`, por lo que nunca se debería ejecutar más allá de un exec a menos que ocurra un error en el reemplazo.

int main() {

pid\_t pid = fork();

switch (pid) {

case -1: *// Error del fork*

perror("fork");

break;

case 0: *// Proceso hijo*

execlp("ls", "ls", "-l", NULL);

perror("exec");

break;

default: *// Proceso padre*

break;

}

}

1. Implementa (y posteriormente entrega) un comando ***orden*** que ejecute el programa (junto con sus argumentos) indicados por parámetro. Ejemplo: *orden ls –l*  Ejecutaría *ls -l*

int main(int argc, char \*argv[]) {

*// Si se invoca sin parámetros, lanzar error.*

if (argc < 2) {

cerr << "Uso: " << argv[0] << " [<argumentos>]" << endl;

exit(EXIT\_FAILURE);

}

*// Clonar el vector de parámetros.*

*// Importante ignorar el primer parámetro, que es el nombre del programa.*

char \*\*params = new char\*[argc + 1];

for (int i = 0; i < argc - 1; i++) params[i] = argv[i + 1];

*// Añadir un null al nuevo vector.*

params[argc + 1] = NULL;

*// Reemplazar el contenido del proceso actual por el comando indicado en*

*// los argumentos.*

execvp(params[0], params);

}

1. En el directorio **/home/asignaturas/so/Shell** encontrarás las dos primeras versiones del código de un intérprete de comandos (Shell) de UNIX, cópialos a tu directorio personal. Compila y ejecuta la Version1 y trata de ejecutar varios comandos UNIX o programas. ¿Qué problema serio tiene esta primera versión? ¿Por qué? (Deberías mirar el código fuente de la función ***main*** para responder a estas preguntas). Vuelve a ejecutar la Version1 e intenta ejecutar varios programas que no existan (por ejemplo *novale*, *noexiste*, etc). ¿Qué diferencia observas? Explica por qué ocurre.

Si el exec tiene éxito, es decir, se ejecuta una orden válida, el Shell se cierra puesto que no se crea ningún proceso hijo. Esto supone además que no se devuelve ningún tipo de mensaje de error al ejecutar instrucciones inválidas como las propuestas en el enunciado.

1. Observa el código fuente de la Version1 (sobre todo la función ***main***) ¿Qué variante del **exec**() utiliza el intérprete de comandos para ejecutar los programas? ¿Por qué se utiliza esa variante y no otra? ¿Por qué es necesaria la función ***ObtieneArgumentos***?

Se utiliza `execvp` para utilizar el vector modificado por `ObtieneArgumentos` y poder buscar ficheros válidos en el PATH actual.

1. Compila y ejecuta ahora la Version2 y trata de ejecutar varios comandos UNIX o programas. ¿Qué se mejora respecto a la Version1? ¿Qué solución se ha adoptado en el código fuente para realizar tal mejora? ¿Cuántos procesos hijo se crean cada vez que se intenta ejecutar un programa en el intérprete? ¿Qué proceso ejecuta dicho programa? ¿En qué momento finalizan los procesos hijo? ¿Y el proceso padre?

**El programa padre lee hasta el fin del fichero. El programa hijo solo ejecuta el código del programa indicado en los argumentos tratados. Se crea un proceso hijo por comando, por lo que se corrigen los fallos de la versión anterior. Sin embargo, el programa padre nunca termina.**