# Sistemas Operativos

Práctica 1: Procesos y API del SO

**Notas preliminares**

- Los ejercicios marcados con el símbolo ★ constituyen un subconjunto mínimo de ejercitación. Sin embargo, aconsejamos fuertemente hacer todos los ejercicios.

---

## Parte 1 – Estado y operaciones sobre procesos

### Ejercicio 1

¿Cuáles son los pasos que deben llevarse a cabo para realizar un cambio de contexto?

### Ejercicio 2 ★

El PCB (Process Control Block) de un sistema operativo para una arquitectura de 16 bits es

```
struct PCB {
  int STAT;      // valores posibles KE_RUNNING, KE_READY, KE_BLOCKED, KE_NEW
  int P_ID;      // process ID
  int PC;        // valor del PC del proceso al ser desalojado
  int R0;        // valor del registro R0 al ser desalojado
  ...
  int R15;       // valor del registro R15 al ser desalojado
  int CPU_TIME   // tiempo de ejecución del proceso
}
```

a) Implementar la rutina `Ke_context_switch(PCB* pcb_0, PCB* pcb_1)`, encargada de realizar el cambio de contexto entre dos procesos (cuyos programas ya han sido cargados en memoria) debido a que el primero ha consumido su *quantum*. `pcb_0` es el puntero al PCB del proceso a ser desalojado y `pcb_1` al PCB del proceso a ser ejecutado a continuación. Para implementarla se cuenta con un lenguaje que posee acceso a los registros del procesador R0, R1, ..., R15, y las siguientes operaciones:

```
·=·;                                  // asignación entre registros y memoria
int  ke_current_user_time();          // devuelve el valor del cronómetro
void ke_reset_current_user_time();    // resetea el cronómetro
void ret();                           // desapila el tope de la pila y reemplaza el PC
void set_current_process(int pid)     // asigna al proceso con el pid como el siguiente
     a ejecutarse
```

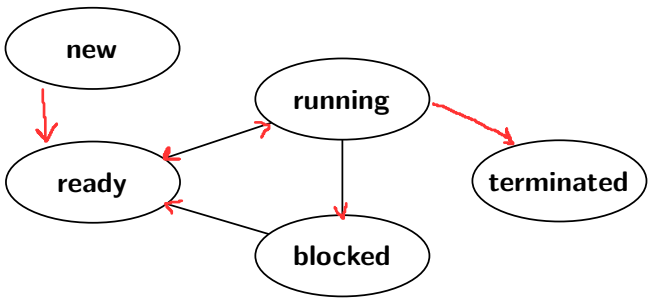
b) Identificar en el programa escrito en el punto anterior cuáles son los pasos del ejercicio 1.

### Ejercicio 3

Describir la diferencia entre un *system call* y una llamada a función de biblioteca.

### Ejercicio 4 ★

En el esquema de transición de estados que se incluye a continuación:




a) Dibujar las puntas de flechas que correspondan. También puede agregar las transiciones que crea necesarias entre los estados disconexos y el resto.
b) Explicar qué causa cada transición y qué componentes (*scheduler*, proceso, etc.) estarían involucrados.

### Ejercicio 5 ★

Un sistema operativo ofrece las siguientes llamadas al sistema:

| | |
|---|---|
| `pid fork()` | Crea un proceso exactamente igual al actual y devuelve el nuevo *process* ID en el proceso padre y 0 en el proceso hijo. |
| `void wait_for_child(pid child)` | Espera hasta que el *child* indicado finalice su ejecución. |
| `void exit(int exit_code)` | Indica al sistema operativo que el proceso actual ha finalizado su ejecución. |
| `void printf(const char *str)` | Escribe un *string* en pantalla. |

a) Utilizando únicamente la llamada al sistema `fork()`, escribir un programa tal que construya un árbol de procesos que represente la siguiente genealogía: Abraham es padre de Homero, Homero es padre de Bart, Homero es padre de Lisa, Homero es padre de Maggie. Cada proceso debe imprimir por pantalla el nombre de la persona que representa.

b) Modificar el programa anterior para que cumpla con las siguientes condiciones: 1) Homero termine sólo después que terminen Bart, Lisa y Maggie, y 2) Abraham termine sólo después que termine Homero.

### Ejercicio 6

El sistema operativo del punto anterior es extendido con la llamada al sistema `void exec(const char *arg)`. Esta llamada al sistema reemplaza el programa actual por el código localizado en el string (char *arg). Implementar una llamada al sistema que tenga el mismo comportamiento que la llamada `void system(const char *arg)`, usando las llamadas al sistema ofrecidas por el sistema operativo. Nota: Revisar man system, como ayuda.

### Ejercicio 7 (*Interfaz del SO POSIX*) ★

Programar en `C` el ejercicio 5b y 6.

**Ejercicio 8** (*Interfaz del SO POSIX*) ★

Veamos el siguiente fragmento de código de un fork

```
int main(int argc, char const *argv[]){
  int dato = 0;
  pid_t pid = fork();
  //si no hay error, pid vale 0 para el hijo
  //y el valor del process id del hijo para el padre
  if (pid == -1) exit(EXIT_FAILURE);
  //si es -1, hubo un error
  else if (pid == 0) {
    for (int i=0; i< 3; i++) {
      dato++;
      printf("Dato hijo: %d\n", dato);
    }
  }
  else {
    for (int i=0; i< 3; i++) {
      printf("Dato padre: %d\n", dato);
    }
  }
  exit(EXIT_SUCCESS); //cada uno finaliza su proceso
}
```

¿Son iguales los resultados mostrados de la variable `dato` para el padre y para el hijo? ¿Qué está sucediendo?

**Ejercicio 9** (*Interfaz del SO POSIX - Señales*) ★

Dado un programa de dos procesos, padre e hijo, se quiere tener el siguiente comportamiento: Uno de los dos procesos debe escribir en pantalla **ping** y su número de `PID`. Automáticamente el otro proceso debe escribir **pong** con su número de `PID`. Se quiere repetir este comportamiento 3 veces. Luego de esto, se desea preguntar al usuario si quiere finalizar la ejecución o no. En caso que conteste que si, el padre debe terminar con la ejecución de su hijo y finalizar. En caso que se conteste que no, se vuelve a repetir el proceso antes dicho.

**Ejercicio 10** (*Uso de strace*) ★

- Dado el siguiente fragmento de strace. Escribir el código correspondiente.

```
execve("./programa", ["./programa"], 0x7fff6ac6fab8 /* 50 vars */) = 0
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffe6cd07d90) = -1 EINVAL (Argumento inválido)
brk(NULL)                               = 0x2460000
brk(0x24611c0)                          = 0x24611c0
arch_prctl(ARCH_SET_FS, 0x2460880)      = 0
uname({sysname="Linux", nodename="compu", ...}) = 0
readlink("/proc/self/exe", "/so/2"..., 4096) = 88
brk(0x24821c0)                          = 0x24821c0
brk(0x2483000)                          = 0x2483000
mprotect(0x4be000, 12288, PROT_READ)    = 0
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLDstrace: Proc
```

```
, child_tidptr=0x2460b50) = 10552
[pid 10551] write(1, "Soy Juan\n\0", 10) = 10
[pid 10552] write(1, "Soy Julieta\n", 12 <unfinished ...>
[pid 10551] clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0},  <unfinished ...>
[pid 10552] <... write resumed>)        = 12
[pid 10552] clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0},  <unfinished ...>
[pid 10551] <... clock_nanosleep resumed>0x7ffe6cd07ca0) = 0
[pid 10551] wait4(-1,  <unfinished ...>
[pid 10552] <... clock_nanosleep resumed>0x7ffe6cd07ca0) = 0
[pid 10552] clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD
, child_tidptr=0x2460b50) = 10557
[pid 10557] write(1, "Soy Jennifer\n\0", 14 <unfinished ...>
[pid 10552] exit_group(0)               = ?
[pid 10557] <... write resumed>)        = 14
[pid 10557] clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0},  <unfinished ...>
[pid 10552] +++ exited with 0 +++
[pid 10551] <... wait4 resumed>[{WIFEXITED(s) && WEXITSTATUS(s) == 0}], 0, NULL) = 10552
[pid 10551] --- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED, si_pid=10552, si_uid=1000,
[pid 10551] clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD
strace: Process 10558 attached
[pid 10551] exit_group(0)               = ?
[pid 10558] write(1, "Soy Jorge\n", 10) = 10
[pid 10558] clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0},  <unfinished ...>
[pid 10551] +++ exited with 0 +++
[pid 10557] <... clock_nanosleep resumed>0x7ffe6cd07ca0) = 0
[pid 10557] exit_group(0)               = ?
[pid 10558] <... clock_nanosleep resumed>0x7ffe6cd07ca0) = 0
[pid 10558] exit_group(0)               = ?
[pid 10557] +++ exited with 0 +++
+++ exited with 0 +++
```

- Ejecutar strace del código que realizó y contrastarlos. ¿Qué comando exacto utilizó para strace?

## Parte 2 – Comunicación entre procesos

### Ejercicio 11 ★

Un nuevo sistema operativo ofrece las siguientes llamadas al sistema para efectuar comunicación entre procesos:

| | |
|---|---|
| `void bsend(pid dst, int msg)` | Envía el valor `msg` al proceso `dst`. |
| `int breceive(pid src)` | Recibe un mensaje del proceso `src`. |

Ambas llamadas al sistema son bloqueantes y la cola temporal de mensajes es de capacidad *cero*. A su vez, este sistema operativo provee la llamada al sistema `pid get_current_pid()` que devuelve el *process id* del proceso que invoca dicha llamada.

a) Escribir un programa que cree un segundo proceso, para luego efectuar la siguiente secuencia de mensajes entre ambos:

1. *Padre* envía a *Hijo* el valor 0,

2. *Hijo* envía a *Padre* el valor 1,
3. *Padre* envía a *Hijo* el valor 2,
4. *Hijo* envía a *Padre* el valor 3,

   etc...

b) Modificar el programa anterior para que cumpla con las siguientes condiciones: 1) *Padre* cree dos procesos hijos en lugar de uno, y 2) se respete esta nueva secuencia de mensajes entre los tres procesos.

1. *Padre* envía a *Hijo_1* el valor 0,
2. *Hijo_1* envía a *Hijo_2* el valor 1,
3. *Hijo_2* envía a *Padre* el valor 2,
4. *Padre* envía a *Hijo_1* el valor 3,

   ...hasta llegar al valor 50.

**Ejercicio 12 ★**

El siguiente programa se ejecuta sobre dos procesos: uno destinado a ejecutar el procedimiento `cómputo_muy_difícil_1()` y el otro destinado a ejecutar el procedimiento `cómputo_muy_difícil_2()`. Como su nombre lo indica, ambos procedimientos son sumamente *costosos* y duran prácticamente lo mismo. Ambos procesos se conocen mutuamente a través de las variables `pid_derecha` y `pid_izquierda`.

```
int result;

void proceso_izquierda() {
  result = 0;
  while (true) {
    bsend(pid_derecha, result);
    result = cómputo_muy_difícil_1();
  }
}

void proceso_derecha() {
  while(true) {
    result = cómputo_muy_difícil_2();
    int left_result = breceive(pid_izquierda);
    printf("%s %s", left_result, result);
  }
}
```

El hardware donde se ejecuta este programa cuenta con varios procesadores. Al menos dos de ellos están dedicados a los dos procesos que ejecutan este programa. El sistema operativo tiene una cola de mensajes de capacidad cero. Las funciones `bsend()` y `breceive()` son las mismas descriptas en el ejercicio anterior (ambas bloqueantes).

a) Sea la siguiente secuencia de uso de los procesadores para ejecutar los procedimientos costosos.

| Tiempo | Procesador 1 | Procesador 2 |
|---|---|---|
| 1 | `cómputo_muy_difícil_1` | `cómputo_muy_difícil_2` |
| 2 | `cómputo_muy_difícil_1` | `cómputo_muy_difícil_2` |
| 3 | `cómputo_muy_difícil_1` | `cómputo_muy_difícil_2` |
| ... | ... | ... |

Explicar por qué esta secuencia no es realizable en el sistema operativo descripto. Escribir una secuencia que sí lo sea.

b) ¿Qué cambios podría hacer *al sistema operativo* de modo de lograr la secuencia descripta en el punto anterior?

### Ejercicio 13

Mencionar y justificar qué tipo de sistema de comunicación (basado en memoria compartida o en pasaje de mensajes) sería mejor usar en cada uno de los siguientes escenarios:

a) Los procesos `cortarBordes` y `eliminarOjosRojos` necesitan modificar un cierto archivo `foto.jpg` al mismo tiempo.
b) El proceso `cortarBordes` se ejecuta primero y luego de alguna forma le avisa al proceso `eliminarOjosRojos` para que realice su parte.
c) El proceso `cortarBordes` se ejecuta en una casa de fotos. El proceso `eliminarOjosRojos` es mantenido en tan estricto secreto que la computadora que lo ejecuta se encuentra en la bóveda de un banco.

### Ejercicio 14 ★

Un sistema operativo provee las siguientes llamadas al sistema para efectuar comunicación entre procesos mediante pasaje de mensajes.

| | |
|---|---|
| `bool send(pid dst, int *msg)` | Envía al proceso `dst` el valor del puntero. Retorna `false` si la cola de mensajes estaba llena. |
| `bool receive(pid src, int *msg)` | Recibe del proceso `src` el valor del puntero. Retorna `false` si la cola de mensajes estaba vacía. |

a) Modificar el programa del ejercicio 12 para que utilice estas llamadas al sistema.

b) ¿Qué capacidad debe tener la cola de mensajes para garantizar el mismo comportamiento?

### Ejercicio 15

Pensar un escenario donde tenga sentido que dos procesos (o aplicaciones) tengan entre sí un canal de comunicaciones bloqueante y otro no bloqueante. Describir en pseudocódigo el comportamiento de esos procesos.

### Ejercicio 16 ★

Escribir el código de un programa que se comporte de la misma manera que la ejecución del comando "`ls -al | wc -l`" en una *shell*. No está permitido utilizar la función `system`, y cada uno de los programas involucrados en la ejecución del comando deberá ejecutarse como un subproceso.

### Ejercicio 17 ★

Implementar el inciso b del ejercicio 11 usando pipes en C. Determinar si el comportamiento del intercambio de mensajes obtenido es igual al especificado por las funciones bsend y breceive.

**Ejercicio 18 ★**

Se cuenta con una operación computacional costosa que se desea repartir entre N subprocesos.

Para ello, el proceso padre dispone de una función *int dameNumero(int pid)* que dado el **PID** de un hijo le devolverá un número secreto. Este número secreto deberá ser enviado al hijo correspondiente utilizando pipes. Esta función solo puede ser llamada por el padre.

Cada subproceso deberá encargarse de realizar el cómputo del número correspondiente utilizando para ello la función *int calcular(int numero)*. El número que deben utilizar como parámetro es el resultado de la función *dameNumero* que el padre les envió.

Los subprocesos ejecutarán la función calcular y, a medida que vayan terminando, le informarán el resultado al padre.

El proceso padre deberá llamar a la función *void informarResultado(int numero, int resultado)*, la cual recibirá como parámetros el número sobre el que se ejecutó el cálculo, y el resultado que éste produjo. Esta función solamente podrá ser llamada desde el proceso padre.

La función *informarResultado* deberá ser llamada en el mismo orden en que los procesos fueron terminando los distintos cómputos.

Se implementó una versión de este programa, en la cual el proceso padre realiza polling sobre los hijos para ver si terminaron, es decir, los va recorriendo en orden y para cada hijo le pregunta si terminó, en caso de responder afirmativamente, llama a la función *informarResultado*.

```
void ejecutarHijo (int i, int pipes[][2]) {
    // ...
}
int main(int argc, char* argv[]){
    if (argc< 2) {
        printf ("Debe ejecutar con la cantidad de hijos como parametro\n");
    return 0; }
    int N = atoi(argv[1]);
    int pipes[N*2][2];
    for ( int i=0; i< N*2; i++){
        pipe(pipes[i]); }
    for (int i=0; i< N; i++) {
        int pid = fork () ;
        if (pid==0) {
            ejecutarHijo(i,pipes);
            return 0;
        } else {
            int numero = dameNumero(pid) ;
            write(pipes[i][1], &numero, sizeof(numero)); } }
    int cantidadTerminados = 0;
    char hijoTermino [N] = {0};
    while (cantidadTerminados < N) {
        for ( int i=0; i< N; i++) {
            if (hijoTermino[i]) {
            continue; }
            char termino = 0;
            write(pipes[i][1], &termino, sizeof(termino));
            read(pipes[N+i][0], &termino, sizeof(termino));
            if (termino) {
                int numero;
                int resultado ;
                read(pipes[N+i][0], &numero, sizeof(numero));
                read(pipes[N+i][0], &resultado, sizeof(resultado));
```

```
                informarResultado(numero, resultado);
                hijoTermino[i] = 1;
                cantidadTerminados++; }  }  }
    wait(NULL) ;
    return 0; }
```

Resolver la función *ejecutarHijo()* utilizando pipes y señales, respetando el siguiente comportamiento. Para poder responder al *polling* del padre, cada hijo deberá crear un segundo subproceso que será el encargado de ejecutar la función *calcular*. Este subproceso (nieto) le avisará a su padre cuando haya terminado mediante una señal, comunicándole además el resultado. El proceso hijo una vez que sepa que su proceso nieto terminó, responderá afirmativamente al polling del padre, enviándole el número y el resultado. A efectos del ejercicio y para evitar las posibles condiciones de carrera ocasionadas por el polling, se asumirá que dos llamados concurrentes a la función calcular no pueden terminar a la vez ni tampoco cercanos en el tiempo, sino con varios minutos de diferencia entre uno y otro.

### Ejercicio 19

Se tiene un programa que cada vez que se lo ejecuta (sin parámetros) imprime lo siguiente a la salida estándar:

```
¿Cuál es el significado de la vida?
Dejame pensarlo...
Ya sé el significado de la vida.
Mirá vos. El significado de la vida es 42.
```

```
¡Bang Bang, estás liquidado!
Me voy a mirar crecer las flores desde abajo.
Te voy a buscar en la oscuridad.
```

y al correrlo con `strace` se obtiene la siguiente salida (se omiten las partes irrelevantes):

```
execve("./estrella", ["./estrella"], [/* 33 vars */]) = 0
pipe([3, 4])                            = 0
clone(child_stack=0, flags=CLONE_CHILD_CLEARTID|
     CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x15acb50) = 6590
[6590] close(3)                         = 0
[6590] getppid( <unfinished ...>
[6589] close(4)                         = 0
[6589] rt_sigaction(SIGINT, {0x40105e, [INT], ...},  <
     unfinished ...>
[6590] <... getppid resumed> )          = 6589
[6589] <... rt_sigaction resumed> {SIG_DFL, [], 0}, 8) = 0
[6590] rt_sigaction(SIGINT, {0x4010ea, [INT], ...},  <
     unfinished ...>
[6589] rt_sigprocmask(SIG_BLOCK, [CHLD],  <unfinished ...>
[6590] <... rt_sigaction resumed> {SIG_DFL, [], 0}, 8) = 0
[6589] <... rt_sigprocmask resumed> [], 8) = 0
[6590] rt_sigaction(SIGHUP, {0x40115d, [HUP], ...},  <
     unfinished ...>
[6589] rt_sigaction(SIGCHLD, NULL,  <unfinished ...>
[6590] <... rt_sigaction resumed> {SIG_DFL, [], 0}, 8) = 0
[6589] <... rt_sigaction resumed> {SIG_DFL, [], 0}, 8) = 0
[6589] rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
[6589] nanosleep({1, 0}, 0x7ffdd87913d0) = 0
[6589] fstat(1, ...}) = 0
[6589] mmap(NULL, 4096, PROT_READ|PROT_WRITE, ...) = 0x7f4
[6589] write(1, "¿Cuál es el significado de la "..., 38) = 38
[6589] kill(6590, SIGINT <unfinished ...>
[6590] --- SIGINT {si_signo=SIGINT, si_code=SI_USER, si_pid
     =6589}
[6589] <... kill resumed> )             = 0
[6590] fstat(1, ...}) = 0
[6590] mmap(NULL, 4096, PROT_READ|PROT_WRITE, ...) = 0x7f4
[6590] write(1, "Dejame pensarlo...\n", 19) = 19
[6590] rt_sigprocmask(SIG_BLOCK, [CHLD], [INT], 8) = 0
[6590] rt_sigaction(SIGCHLD, NULL, {SIG_DFL, [], 0}, 8) = 0
[6590] rt_sigprocmask(SIG_SETMASK, [INT], NULL, 8) = 0
```

```
[6590] nanosleep({5, 0}, 0x7ffdd8790e00) = 0
[6590] write(1, "Ya sé el significado de la vida"..., 34) = 34
[6590] write(4, "42", 2)                = 2
[6590] kill(6589, SIGINT)               = 0
[6589] --- SIGINT {si_signo=SIGINT, si_code=SI_USER, si_pid
     =6590} ---
[6590] rt_sigreturn()                   = 0
[6589] read(3, "42", 3)                 = 2
[6589] write(1, "Mirá vos. El significado de la "..., 44) = 44
[6589] write(1, "¡Bang Bang, estás liquidado!\n", 31) = 31
[6589] kill(6590, SIGHUP <unfinished ...>
[6590] --- SIGHUP {si_signo=SIGHUP, si_code=SI_USER, si_pid
     =6589} ---
[6589] <... kill resumed> )             = 0
[6589] rt_sigprocmask(SIG_BLOCK, [CHLD], [INT], 8) = 0
[6590] write(1, "Me voy a mirar crecer las flores"..., 46 <
     unfinished ...>
[6589] rt_sigaction(SIGCHLD, NULL, {SIG_DFL, [], 0}, 8) = 0
[6589] rt_sigprocmask(SIG_SETMASK, [INT],  <unfinished ...>
[6590] <... write resumed> )            = 46
[6589] <... rt_sigprocmask resumed> NULL, 8) = 0
[6590] close(4)                         = 0
[6590] exit_group(0)                    = ?
[6589] nanosleep({10, 0},  <unfinished ...>
[6590] +++ exited with 0 +++
<... nanosleep resumed> {10, 32866})    = ?
     ERESTART_RESTARTBLOCK (Interrupted by signal)
--- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED, si_pid=6590,
      si_status=0, si_utime=100, si_stime=0} ---
restart_syscall(<... resuming interrupted call ...>) = 0
write(1, "Te voy a buscar en la oscuridad.\n", 33) = 33
close(3)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

a) Identificar qué funciones de la `libc` generan cada una de las *syscalls* observadas.

b) Escribir un programa que posea un comportamiento similar al observado. Es decir que, al ejecutarlo, produzca la misma salida, y que la secuencia de *syscalls* observadas al correrlo con `strace` sea la misma que se muestra aquí.