

# 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 2 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_currrent_user_time();        // retorna 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
```

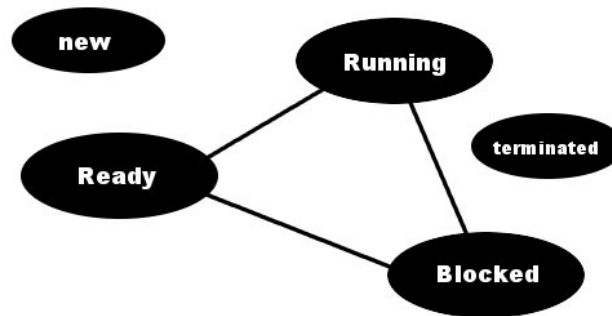
- b) Identificar en el programa escrito en el punto anterior cuáles son los pasos del ejercicio 1.

### Ejercicio 3

Describe 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:



- Dibujar las puntas de flechas que correspondan. Agregar las transiciones que crea necesarias entre los estados desconexos y el resto.
- Explicar cuál es la causa de cada transición y qué componentes (scheduler, proceso, etc.) estarían involucrados.

### Ejercicio 5 ★

Un sistema operativo ofrece las siguientes llamadas al sistema:

<code>pid fork();</code>	Crea un proceso exactamente igual al actual y retorna el nuevo process ID en el proceso padre y 0 en el proceso hijo.
<code>void wait_for_child(pid child)</code>	Espera hasta que el child indicado finaliza su ejecución.
<code>void exit(int exit_code);</code>	Indica al sistema operativo que el proceso actual ha finalizado su ejecución.
<code>void printf(const char *str);</code>	Escribe un string en pantalla.

- 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 Homer, Homer es padre de Bart, Homer es padre de Lisa, Homer es padre de Maggie. Cada proceso debe imprimir por pantalla el nombre de la persona que representa.
- Modificar el programa anterior para que cumpla con las siguientes condiciones: 1) Homer termine sólo después que terminen Bart, Lisa y Maggie, y 2) Abraham termine sólo después que termine Homer.

### 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. Implementar la llamada al sistema *void system(const char \*arg)* usando las llamadas al sistema ofrecidas por este sistema operativo.

## Parte 2 – Comunicación entre procesos

### Ejercicio 7 ★

Un maniático del conocido juguete de malabares *tiki-taka*<sup>1</sup> ha decidido homenajear a su objeto de pasión mediante la ejecución del siguiente programa. El homenaje consiste en:

- Un proceso lee la variable tiki y escribe su contenido incrementado en 1 en la variable taka

<sup>1</sup>[http://es.wikipedia.org/wiki/Tiki\\_taka](http://es.wikipedia.org/wiki/Tiki_taka)

- Otro proceso lee la variable taka, escribiendo su contenido incrementado en 1 en la variable tiki.

```
#define SIZE 5

int tiki;
int taka;
int temp;

void taka_runner() {
    while (true) {
        temp = tiki;
        temp++;
        taka = temp;
    }
}

void tiki_taka() {
    while (true) {
        temp = taka;
        temp++;
        tiki = temp;
    }
}
```

El sistema operativo ofrece las siguientes llamadas al sistema para efectuar una comunicación entre distintos procesos:

void share_mem(int *ptr)	Permite compartir el puntero a todos
void share_mem(int *ptr, int size)	los hijos que el proceso cree.

- ¿Qué variables deben residir en el área de memoria compartida?
- ¿Existe alguna variable que no deba residir en el espacio de memoria compartida?
- Escribir un procedimiento *main()* para el problema del tiki-taka usando el código presentado y las llamadas al sistema para comunicación entre procesos provistas por este sistema operativo.

### Ejercicio 8 ★

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 retorna el process id del proceso que la invoca.

- Escribir un programa que cree un segundo proceso, para luego efectuar la siguiente secuencia de mensajes entre ambos:  
Padre envía a Hijo el valor 0,

Hijo envía a Padre el valor 1,  
Padre envía a Hijo el valor 2,  
Hijo envía a Padre el valor 3,  
etc...

- b) Modificar el programa anterior para que cumpla con las siguientes condiciones: 1) se cree un tercer proceso, y 2) se respete esta nueva secuencia de mensajes entre los 3 procesos.

Padre envía a Hijo\_1 el valor 0,  
Hijo\_1 envía a Hijo\_2 el valor 1,  
Hijo\_2 envía a Padre el valor 2,  
Padre envía a Hijo\_1 el valor 3,  
etc...

- c) ¿En el punto anterior usó más mensajes que los indicados (es decir, un mensaje para valor 1, otro para valor 2, etc)? Si es así, reescribir el programa usando las llamadas al sistema de memoria compartida del ejercicio 8 para compartir esa información.

## Ejercicio 9

El siguiente programa se ejecuta sobre 2 procesos: uno destinado a ejecutar el procedimiento *very\_hard\_computation\_1* y el otro destinado a ejecutar el procedimiento *very\_hard\_computation\_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 *right\_pid* y *left\_pid*.

```
int result;

void left_process() {
    result = 0;
    while (true) {
        bsend(right_pid, result);
        result = very_hard_computation_1();
    }
}

void right_process() {
    while(true) {
        result = very_hard_computation_2();
        int left_result = breceive(left_pid);
        printf("%s %s", left_result, result)
    }
}
```

El hardware donde se ejecuta este programa cuenta con varios procesadores. Al menos 2 de ellos están dedicados a los 2 procesos que ejecutan este programa. El sistema operativo tiene una cola de mensajes de capacidad cero. Las funciones *bsend* y *breceive* son las mismas descritas 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	very_hard_computation.1	very_hard_computation.2
2	very_hard_computation.1	very_hard_computation.2
3	very_hard_computation.1	very_hard_computation.2
...	...	...

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

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

### Ejercicio 10

Mencione y justifique qué tipo de sistema de comunicación usaría (basado en memoria compartida o en pasaje de mensajes) en cada uno de los siguientes escenarios:

- Los procesos `cortarBordes` y `eliminarOjosRojos` necesitan modificar un cierto archivo `foto.jpg` al mismo tiempo.
- El proceso `cortarBordes` se ejecuta primero y luego de alguna forma le avisa al proceso `eliminarOjosRojos` para que realice su parte.
- 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 11 ★

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

<code>bool send(pid dst, int* msg)</code>	Envía al proceso <code>dst</code> el valor del puntero Retorna <i>false</i> si la cola de mensajes estaba llena.
<code>bool receive(pid src, int* msg)</code>	Recibe del proceso <code>src</code> el valor del puntero Retorna <i>false</i> si la cola de mensajes estaba vacía.

- Modificar el programa del ejercicio 9 para que utilice estas llamadas al sistema.
- ¿Qué capacidad debe tener la cola de mensajes para garantizar el mismo comportamiento?

### Ejercicio 12

¿Qué sucedería si un sistema operativo implementara *pipes* como único sistema de comunicación interprocesos?

### Ejercicio 13 ★

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 14

El comportamiento esperado del siguiente programa es que el proceso hijo envíe el mensaje *hola* al proceso padre, para que luego el proceso padre responda *chau*. Encontrar un defecto en esta implementación y solucionarlo.

```

pid shared_parent_pid;
pid shared_child_pid;
mem_share(&shared_parent_pid);
mem_share(&shared_child_pid);
shared_parent_pid = get_current_pid();
pid child = fork();

if (child == 0) {
    shared_child_pid = get_current_pid();
    bsend(shared_parent_pid, "hola");
    breceive(shared_parent_pid);
    exit(OK);
} else {
    breceive(shared_child_pid);
    bsend(shared_child_pid, "chau");
    exit(OK);
}

```

### Ejercicio 15 ★

La secuencia de Fibonacci es la serie de números 0, 1, 1, 2, 3, 5, 8, ... Formalmente se expresa como:

$$\begin{aligned}
 Fib_0 &= 0 \\
 Fib_1 &= 1 \\
 Fib_n &= Fib_{n-1} + Fib_{n-2}
 \end{aligned}$$

Escriba un programa que, usando la llamada al sistema `fork()`, calcule la secuencia de Fibonacci (es decir, no lo resuelva de la manera clásica iterativa ni recursiva). El programa deberá imprimir por pantalla el N-ésimo elemento de la secuencia, donde N se pasa como parámetro al programa por línea de comando. Por ejemplo, si se recibe como parámetro el valor 6, el programa deberá imprimir “8”. La comunicación entre los procesos deberá realizarse mediante memoria compartida. Para ello el sistema cuenta con la siguiente primitiva: `sh_mem(void * ptr)`

### Ejercicio 16

Se desea hacer un programa que corra sobre una arquitectura con 8 núcleos y calcule promedios por fila de un archivo de entrada que contiene una matriz de enteros positivos de  $N \times M$ . Se quiere que el promedio de cada fila sea calculado en un proceso separado, en un máximo de 8 procesos simultáneos, y que los procesos se comuniquen utilizando *pipes*. Cada proceso debe recibir una fila para calcular del proceso padre, quien las distribuirá entre sus hijos siguiendo una política *Round-Robin*. Finalmente, la salida del programa debe mostrarse por la salida standard, ordenada de menor a mayor. Por ejemplo, si el programa recibe como entrada un archivo con la siguiente matriz (con  $N = 3$  y  $M = 4$ ):

$$\begin{pmatrix} 4 & 4 & 2 & 2 \\ 1 & 2 & 8 & 9 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

la salida debe ser  $(1 \ 3 \ 5)^\top$ . Se pide escribir la implementación del programa. Puede asumir que  $N$ ,  $M$ , el nombre del archivo y la cantidad de núcleos se encuentran *hardcodeados*, y que cuenta con las siguientes funciones auxiliares:

- `int cargar_fila(const int fd, int* lista)`: lee una línea del archivo indicado por el *file descriptor* `fd` como una lista de enteros, y la almacena en `lista`. Devuelve 1 en caso de éxito, 0 si no quedan más filas.

- `int calcular_promedio(const int* lista)`: toma la lista de enteros indicada por `lista` y devuelve su promedio utilizando división entera.
- `void sort(char* s)`: toma la cadena de texto indicada por `s` y conformada por un número por línea, y la modifica de forma que quede ordenada de menor a mayor (similar a ejecutar `sort -n` en UNIX).
- `int dup2(int oldfd, int newfd)`: *linkea* los *file descriptors* `newfd` y `oldfd`, de forma tal que realizar una operación sobre `newfd` es equivalente a hacerla sobre `oldfd`.

## Parte 3 – Anexo Taller IPC

### Ejercicio 17

¿Por qué el programa `exec1p.c` del taller de IPC a veces escribe sus datos por encima del *prompt* de la consola? Explique cómo solucionar este comportamiento no deseado.

### Ejercicio 18

Modifique el programa `pipe.c` para que el proceso que recibe los datos le agradezca mediante el envío de un mensaje ¡gracias! al mensaje que los envió. ¿Es posible realizar esto con el *pipe* creado en `sockets[0]` y `sockets[1]`?

Implemente las dos siguientes alternativas:

- Utilizar dos *pipes*, cada uno envía dirección en un sentido.
- Utilizar la llamada al sistema `socketpair`.

### Ejercicio 19

Modifique el programa `display2.c` que implementa una pantalla donde dos clientes escriben a la vez. La nueva versión debe crear un proceso hijo que atienda a uno de los clientes, mientras que el padre atiende al otro.

**Nota:** para simplificar la implementación puede suponer que este proceso funciona por siempre. No es necesario contemplar el problema de sincronizar el final de cada proceso.