Ejercicios de repaso para el primer parcial

¡Incluye soluciones!

Sistemas Operativos

18 de abril de 2017

Ejercicio 1

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.

Resolución

Nota: El código que incluimos en esta resolución contiene solo los aspectos relevantes del mismo. Pueden encontrar una versión completa, que compila y hace chequeo de errores al ejecutar las *syscalls*, en `https://gist.github.com/sromano/93b968ad2e72e9f63f9c6ae2b180ef33`.

```
int main(int argc, char *argv[]) {
    pid_t child_pid;
    int pipe_fd[2];

    // Obtengo el número del file descriptor de stdin y de stdout
    //   (manera elegante para no poner 0 y 1 hardcodeado)
    int STDOUT = fileno(stdout);
    int STDIN = fileno(stdin);

    // Inicializo el pipe
    pipe(pipe_fd);

    // Creo el proceso hijo
    child_pid = fork();

    // Si soy hijo, ejecuto ls
    if (child_pid == 0) {
        // Cierro el extremo de lectura del pipe en el hijo
        close(pipe_fd[0]);

        // Conecto stdout al extremo de escritura del pipe
        dup2(pipe_fd[1],STDOUT);

        // Cambio el código a ls
        execlp("ls", "ls", 0, 0);
    }

    // Si soy padre, ejecuto wc
```

```
    else {
        // Cierro el extremo de escritura del pipe en el padre
        close(pipe_fd[1]);

        // Conecto stdin al extremo de lectura del pipe
        dup2(pipe_fd[0],STDIN);

        // Cambio el código a wc
        execlp("wc", "wc", "-l", 0, 0);
    }

    return 0;
}
```

Ejercicio 2

El algoritmo de elección de *token ring* es un mecanismo descentralizado que se utiliza para decidir un coordinador entre un conjunto de procesos. A continuación presentamos una variación de este algoritmo.

Inicialmente, se organizan N procesos identificados del 0 al $N-1$ en una estructura de anillo, de forma tal que cada uno tiene un único antecesor y un único sucesor. A la vez, cada proceso tiene un número mágico asociado, que lo calcula llamando a la función `int númeroMágico()`.

Una vez armado el anillo, el proceso 0, llamado *iniciador*, envía un mensaje a su sucesor que consiste en un arreglo de dos enteros: su número de proceso (en este caso 0) y su número mágico. Este mensaje es recibido por el siguiente proceso del anillo, que verifica si su propio número mágico es mayor que el que recibió de su antecesor. Si lo es, entonces envía al proceso siguiente su propio arreglo (con su número de proceso y su número mágico); si no lo es, envía el mismo arreglo que recibió de su antecesor. Esta secuencia es repetida hasta que el proceso iniciador recibe el último mensaje del anillo. Una vez que se recorrió todo el anillo, el número de proceso que recibió el iniciador corresponde al proceso elegido como coordinador. Es decir, el proceso coordinador acaba siendo el que tiene mayor número mágico. Finalmente, el iniciador envía un mensaje (que también debe recorrer todo el anillo) indicando cuál es el proceso coordinador.

Se pide implementar un programa que cree N procesos y utilice el algoritmo anterior para elegir un proceso coordinador entre el conjunto de procesos. Una vez finalizada la elección, el resto de los procesos debe ejecutar la función `double operaciónComplicada()` y enviar su resultado al coordinador, que debe imprimirlos en pantalla. Finalmente, todos los procesos deben terminar.

Tener en cuenta que:

- Los procesos deben comunicarse entre sí utilizando pipes.
- Puede asumir que posee las funciones `int anterior(int i)` e `int siguiente(int i)`, que dado un `i` entre 0 y $N-1$ devuelven cuál es el `i` anterior o posterior del anillo respectivamente.
- Puede asumir que, al escribir varios procesos en un mismo pipe, sus datos no se mezclarán.
- Se deben cerrar todos los pipes que no se utilicen.

Resolución

Nota: El código que incluimos en esta resolución contiene solo los aspectos relevantes del mismo. Pueden encontrar una versión completa, que compila y hace chequeo de errores al ejecutar las *syscalls*, en `https://gist.github.com/sromano/de661f2745d5bff37afc`.

```
int main(int argc, char *argv[]) {
    int ring_pipes[N][2];
    int data_pipe[2];

    // Inicializamos los pipes para el anillo
    // TIP: Pensar por qué se inicializan antes del fork()
    for (int i = 0; i < N; i++) {
        pipe(ring_pipes[i]) == -1;
    }

    // Inicializamos el pipe por donde se van a transmitir los datos al coordinador
    // TIP: Pensar por qué no utilizamos ring_pings para eso
    pipe(data_pipe);

    // Creamos los procesos
    // TIP: Prestar atención a que ningún hijo cree procesos de más cuando cicla el for
    for (int i = 0; i < N; i++) {
        pid_t pid = fork();

        if (pid == 0) {

            // TIP: Prestar atención al uso de i como manera de numerar los procesos creados
            int me = i;
            int coordinadorTupla [2];
            int miNúmeroMagico = númeroMagico();
            int miTupla[] = {me, miNúmeroMagico};

            // TIP: Entender bien por qué se cierran estos pipes
            //   (para eso lo hice en dos for :P )

            // Cierro todos los pipes de lecturas menos el "mío"
            //   (donde voy a leer lo que me escriban a mí)
            for (int j = 0; j < N; j++) {
                if (j != me) {
                    close(ring_pipes[j][0]);
                }
            }

            // Cierro todos los pipes de escrituras menos el del "siguiente"
            for (int j = 0; j < N; j++) {
                if (j != siguiente(me)) {
                    close(ring_pipes[j][1]);
                }
            }

            // Si soy el primer proceso, tengo un comportamiento diferente
            //   porque tengo que escribir el primer arreglo
            if (i == 0) {
                // Paso mi tupla al siguiente
                write(ring_pipes[siguiente(me)][1], miTupla, sizeof(miTupla));
                // Espero que se complete la vuelta
                read(ring_pipes[me][0], coordinadorTupla, sizeof(coordinadorTupla));

                // Inicio otra vuelta para avisar quien es el líder definitivo
                write(ring_pipes[siguiente(me)][1], coordinadorTupla,
                    sizeof(coordinadorTupla));
                // Espero que se complete la vuelta
```

```
    read(ring_pipes[me][0], coordinadorTupla, sizeof(coordinadorTupla));
}
else {
    // Recibo el líder actual
    // TIP: Pensar por qué puedo llamar a read a pesar de que (tal vez) todavía
    //   no se hayan creado los otros procesos o que todavía no hayan escrito.
    // TIP: Pensar qué pasaría si nadie hace un write primero
    //   y todos empiezan con un read.

    read(ring_pipes[me][0], coordinadorTupla, sizeof(coordinadorTupla));

    // Preparo la tupla para mandar
    if (coordinadorTupla[1] < miNúmeroMagico) {
        coordinadorTupla[0] = me;
        coordinadorTupla[1] = miNúmeroMagico;
    }

    // Envío la tupla al siguiente
    write(ring_pipes[siguiente(me)][1], coordinadorTupla,
        sizeof(coordinadorTupla));

    // Espero que me digan quien es el líder definitivo
    // TIP: Ver qué estoy leyendo una tupla, aunque podría mandar solo el
    //   entero de quién es el coordinador
    read(ring_pipes[me][0], coordinadorTupla, sizeof(coordinadorTupla));

    // Envío al siguiente el líder definitivo
    // TIP: Ver qué estoy mandando una tupla, aunque podría mandar solo el
    //   entero de quién es el coordinador
    write(ring_pipes[siguiente(me)][1], coordinadorTupla,
        sizeof(coordinadorTupla));
}

double result;
if (coordinadorTupla[0] == me) {
    // Soy el coordinador

    // Como soy el coordinador, cierro el pipe de escritura para mandarle datos
    //   al coordinador
    close(data_pipe[1]);

    // Recibo los resultados y los imprimo
    for (int j = 0; j < (N - 1); j++) {
        read(data_pipe[0], &result, sizeof(result));
        printf("%f\n", result);
    }
}
else {
    // No soy el coordinador

    // Como no soy el coordinador, cierro el pipe de lectura para recibir datos
    //   del resto
    close(data_pipe[0]);

    // Hago la operación y le paso el resultado al líder
    result = operaciónComplicada();
    write(data_pipe[1], &result, sizeof(result));
```

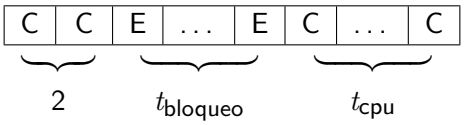
```
            }

            exit(EXIT_SUCCESS);
        }
    }
    return 0;
}
```

Ejercicio 3

Queremos elaborar un sistema para manejo de llamadas a los bomberos. El sistema tomará las llamadas, que pasarán a ser procesos, utilizando siempre 2 unidades de tiempo de CPU para setear los parámetros necesarios. Luego le permitirá al usuario la entrada del nivel de emergencia de la llamada (del 1 a 5) mediante tonos DTMF (también conocidos como *pi pip puuu puu piiii*). Finalmente se procesará la llamada para decidir hacia qué móvil despacharla, lo que nuevamente producirá uso de CPU.

Una tarea estará definida por 3 parámetros: la prioridad que ingresará el usuario, el tiempo que permanecerá bloqueada y el tiempo de procesamiento final para poder ser analizada.

Por ejemplo, si tenemos **TaskLlamada** P t_{bloqueo} t_{cpu}, será interpretado como el proceso como se ve en la figura:



En P vendrá la prioridad a otorgarle a la llamada cuando vuelva del t_{bloqueo}.

Como no tenemos mucho presupuesto, el sistema correrá sobre una máquina con un único procesador. Se espera darle máxima prioridad a los procesos que aún no tienen la entrada DTMF y luego el resto de las tareas en base a su nivel de emergencia.

Asumiendo un tiempo de cambio de contexto de una unidad de tiempo:

(a) Diseñar una política de asignación del procesador que respete las prioridades mencionadas (no necesariamente debe ser igual a uno de los vistos en clase). Si hay varias tareas de misma prioridad, deberá ser justo. Dibujar el diagrama de estados del *scheduler*.

(b) Mostrar un esquema con 5 tareas donde se muestre el funcionamiento de la política propuesta. No hace falta mostrar todos los cambios de prioridad, pero sí algunos.

(c) Graficar el diagrama de Gantt de este esquema.

(d) Calcular, para el esquema anterior, el *waiting time*, el *turnaround* y la latencia.

Resolución

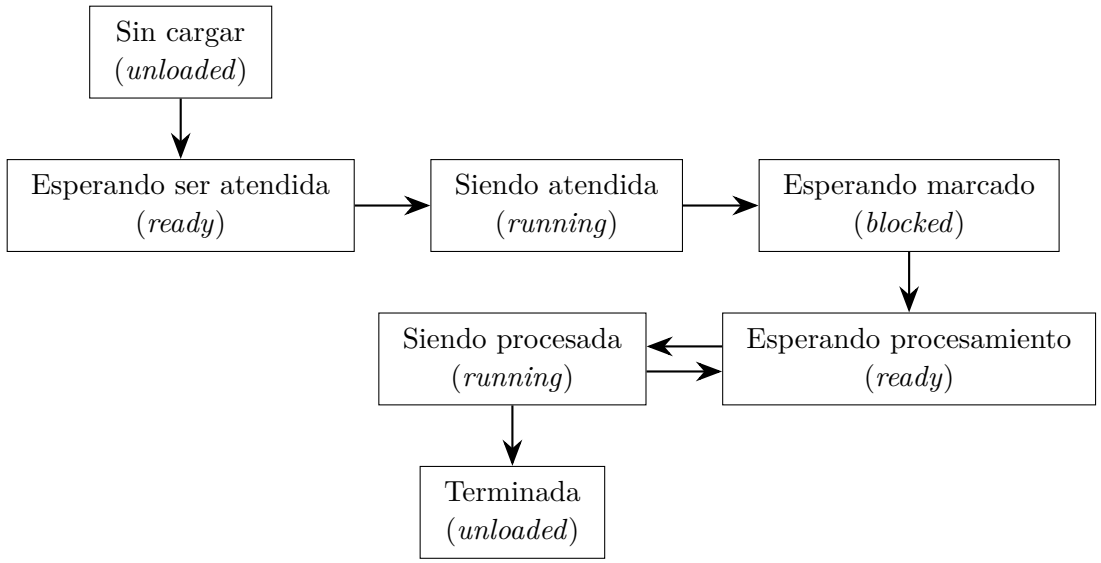
(a) Diseñamos una política que diferencia a las llamadas que están en la etapa inicial de las que ya recibieron el *input* del usuario y están en la etapa de procesamiento. Esto nos permite priorizar siempre a las primeras sobre las segundas. Si llega una nueva llamada al procesador y se está realizando el procesamiento de otra, la última es desalojada para darle lugar a la primera.

Decidimos además que una tarea que está en la etapa inicial no puede ser desalojada, dado que conviene esperar a que se bloquee antes que pagar el costo de un cambio de contexto; además,

esto nos permite simplificar la lógica del *scheduler*. Entre las tareas que se encuentran en la etapa inicial, el orden de ejecución a seguir es FIFO (es decir, tendremos una cola).

Para las tareas en etapa de procesamiento, usaremos una cola para cada una de las prioridades posibles, y solo tomaremos tareas de la cola de mayor prioridad que no esté vacía. Para que el *scheduler* sea justo entre las tareas con igual prioridad, dentro de cada una de las colas utilizaremos una política *round-robin* con un *quantum* de 4 unidades de tiempo.

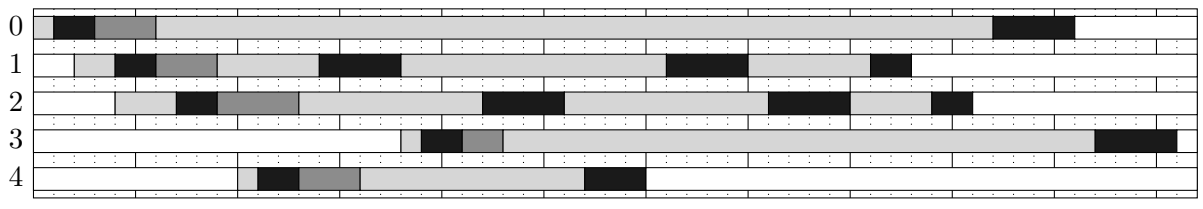
El diagrama de estados del *scheduler* será el siguiente (entre paréntesis, se aclara el estado en que estará el proceso desde el punto de vista del sistema operativo).



(b) Proponemos las siguientes cinco tareas:

#	Tiempo de llegada	P	t_{bloqueo}	t_{cpu}
0	0	3	3	4
1	2	2	3	10
2	4	2	4	10
3	18	4	2	4
4	10	2	3	3

(c) *Ready* *Running* *Blocked* *Unloaded*



(d)
- *Waiting time*: $\dfrac{42+26+26+30+12}{5} = \dfrac{136}{5} = 27.2.$
- *Turnaround*: $\dfrac{51+41+42+38+20}{5} = \dfrac{192}{5} = 38.4.$
- Latencia: $\dfrac{1+2+3+1+1}{5} = \dfrac{8}{5} = 1.6.$