# *Sistemas Operativos*

Examen Práctico
Convocatoria Extraordinaria – 6 septiembre 2021

Disponemos de un conjunto de ficheros en **C** con sus respectivas cabeceras **.h** que una vez correctamente compilados mediante **make** generan un ejecutable llamado **procesar** que generara una salida por pantalla. Se dispone a su vez de un Shell script en Bash llamado **preparar.sh** que preparara un fichero y lanza al ejecutable **procesar**.

Para aprobar es necesario aprobar ambas partes por separado.
Parte C: 5 Puntos. Parte Bash: 5 Puntos

**Técnicas utilizadas:**

- Pipes
- Semáforos usando la librería sem.h/sem.c

Los ficheros **preparar.sh** y **procesar.c** son plantillas o prototipos que forman una solución. El alumno deberá entenderla y completarla. Editará correctamente los bloques **//Begin TODO … //End TODO** colocados en los ficheros **procesar.c y preparar.sh** para que realicen el problema.

Los prototipos de los ficheros están documentados para que el alumno tenga claro lo que realizan, a su vez los bloques TODO tienen comentarios de lo que realiza el bloque y los distintos pasos que se deben cumplir incluyendo el marcador "…" (tres puntos) que el alumno debe sustituir por el código apropiado.

En el prototipo de los ficheros existen constantes definidas que no se deben cambiar, pero el alumno puede añadir las que necesite. El alumno dispone de un fichero "grafo00_dep" que puede utilizar para pasarlo como parámetro a la parte C para probarla, aunque no tenga realizada la parte bash.

## *Control de parámetros:*

Un único parámetro obligatorio para el script bash y para el programa C. Para el script será el nombre del fichero de definición del grafo y para el programa C el nombre del fichero de dependencias que el script genera.

En el Script Bash hay que validar que se pasa un único parámetro y que éste corresponde con un fichero que existe en el directorio actual.

## *El problema:*

Se pretende construir un sistema para lanzar los procesos de un grafo dirigido y sin ciclos a partir de un fichero con la definición del grafo. En el grafo dirigido los nodos representan procesos con una carga de ejecución y las aristas indican el camino (unidireccional) de un nodo a otro. Puede haber varias rutas para recorrer el grafo, pero en una misma ruta no puede haber bucles, es decir no se puede volver a un nodo anterior para un camino dado. (El fichero grafo a utilizar en el examen ya cumple esa condición, Figura 1)

Se pretende que los nodos se ejecuten **tan pronto como sus dependencias le permitan**. Un nodo depende de otro si es destino de una arista que sale del nodo precedente.

En el grafo de ejemplo a usar en el examen se llama *grafo01* que se muestra en la Figura 1. Los nodos están etiquetados con *numNodo/Carga*. Podemos ver como comienza la ejecución en el nodo 0 y se reparte en los nodos 1, 2 y 3, de los cuales se pueden seguir varios paths para llegar al nodo final. El path marcado en rojo es el crítico o camino más largo. Si se ejecutasen todos los procesos secuencialmente tardarían aproximadamente la suma de las cargas de todos, es decir, unos 17 segundos aproximadamente, pero si se permite la ejecución en paralelo de aquellos nodos que puedan correr en paralelo en función de sus dependencias, el tiempo total será aproximadamente la suma de las cargas del camino crítico, es decir, aproximadamente 12 segundos.

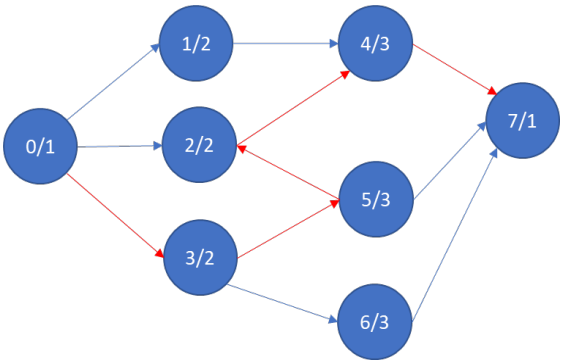


*Figura 1: Grafo modelado en el fichero grafo01*

El proceso bash **preparar.sh** leerá el fichero *grafo01* (el nombre lo recibe como parámetro) y generará el fichero *grafo01_dep* (concatena *_dep* al nombre recibido) que será el que use el programa en C.

**El fichero definición del grafo.**

Contiene la definición del grafo01, Figura 2.

La primera línea contiene la carga de trabajo en segundos para cada nodo, un valor entero por cada nodo del grafo (el primer valor corresponde al nodo 0, y así sucesivamente). Las siguientes líneas indican las aristas mediante dos enteros, el primer valor indica el índice del nodo origen y el segundo valor el índice del nodo destino. Habrá una línea por cada arista en el grafo.

```
1 2 2 2 3 3 3 1
0 1
0 2
0 3
1 4
2 4
3 5
3 6
5 2
4 7
5 7
6 7
```

*Figura 2 : grafo01*

**El fichero de dependencias del grafo.**

Contiene las dependencias de los procesos, grafo01_dep, Figura 3.

Contiene una línea para cada proceso, con valores enteros siendo los tres primeros fijos (todas las líneas tienen al menos 3 valores) y son (en orden):

- Carga en segundos del nodo,
- Número de procesos que dependen de él (o signals que tiene que hacer)
- Número de procesos de los que él depende (o waits que tiene que hacer)

```
1 3 0
2 1 1 0
2 1 2 0 5
2 2 1 0
3 1 2 1 2
3 2 1 3
3 1 1 3
1 0 3 4 5 6
```

*Figura 3: grafo01_dep*

Si un proceso tiene 0 en el tercer valor, es que no depende de otro/otros y por eso ya no tiene más valores en su fila.
Si un proceso tiene un valor > 0 en el tercer valor, le seguirán los números de los procesos de los que depende.
Si un proceso tiene 0 en su segundo valor, es que no hay otros procesos que dependan de él (en el ejemplo el nodo final, el nodo 7)

# Funcionamiento:

El usuario lanzará el script **preparar.sh**, pasándole el nombre de un fichero de grafo:

```
$ preparar.sh grafo01
```

Este script, tras el control de parámetros, lanzará una vez al ejecutable **procesar** pasándole el nombre del fichero generado por preparar.sh. Cuando termina el proceso C se mostrará por pantalla un informe de tiempos que ya está programado en el prototipo de procesar.c.

La Figura 4 muestra la ejecución completa del comando $ *preparar.sh grafo01*

```
----------grafo01---------------
1 2 2 2 3 3 3 1
0 1
0 2
0 3
1 4
2 4
3 5
3 6
5 2
4 7
5 7
6 7
----------grafo01_dep------------
1 3 0
2 1 1 0
2 1 2 0 5
2 2 1 0
3 1 2 1 2
3 2 1 3
3 1 1 3
1 0 3 4 5 6
Procesando grafo grafo01_dep con 8
procesos, espere por favor...
CPU Time measured: 12.004 seconds.
```

*Figura 4: Ejecución completa de preparar.sh grafo01*

## Parte Bash:

En el script ***preparar.sh*** (parte Bash), tras el control de parámetros, realizará las acciones necesarias para generar el fichero de dependencias. Seguir los comentarios del código para completar las secciones TODO del código.

*Ayuda:*

*Se entrega como anexo resumen del uso de arrays en Bash y cómo se descompone un string en tokens (en las cadenas que lo componen).*

## Parte C:

El programa ***procesar.c*** (parte C) leerá el fichero de dependencias *grafo01_dep* y creará un array de estructuras PROCESO donde cada proceso tendrá los pipes para comunicar con el padre y sabrá de que procesos depende, cuantos y cuales dependen de él.

El padre, en un bucle con fork, creará tantos procesos como líneas tenga el fichero (número nodos), que correrán concurrentemente. La estructura *PROCESO* definida en *func.h* es:

```
//Definimos la estructura PROCESO
typedef struct {
        int pid ;         // Pid del proceso
        int p[2];         // Descriptores el pipe del proceso
        int carga;        // Carga del proceso
        int signals;      // Nuero de signals que tiene que hacer el proceso
        int waits;        // Numero de waits que tiene que hacer el proceso
        int waitfor[10]; // Array con el indice de los procesos en los que hace wait
} PROCESO ;
```

Para sincronizar los procesos se utilizarán semáforos, de forma que el padre, antes del fork, creará **un array de semáforos** donde cada proceso o nodo tendrá un semáforo propio.

> *Nota: El array de semáforos es una variable global que una vez inicializada se hereda en cada proceso, los semáforos creados con sCreate de la librería sem.h/sem.c son visibles por los hijos.*

Así, si un proceso depende de otro (según el grafo), realizará un wait sobre el semáforo de éste, si depende de varios realizará un wait en el semáforo de cada proceso de los que dependa. Cuando un

proceso termina realizará tantos signals sobre su propio semáforo como procesos dependan de él, con lo que liberará a todos los procesos que se hayan colgado en su semáforo.

Cuando el padre ha terminado de crear a los procesos, escribirá en el pipe de cada uno en este orden:

- La carga (número de segundos de ejecución del hijo)
- El número de signals ( tantos signals como procesos dependan de él)
- El número de waits (tantos waits como procesos precedentes tenga)
- El índice de cada uno de los procesos precedentes (los índices sobre los que hacer wait)

Cuando un proceso hijo comienza, leerá de su pipe la carga, el número de signals y el número de waits. Si el número de waits es >0, entonces entrará en bucle para leer del pipe el índice y hará wait sobre el semáforo en ese índice. Cuando termine el bucle es porque se han hecho los signals que le permiten seguir.

Si el número de waits es 0 o ha terminado el bucle de waits, entonces puede ejecutar su tarea llamando al procedimiento DoJob pasándole los segundos de su carga.

Cuando termina su ejecución (DoJob), realizará tantos signals sobre su semáforo como haya leído anteriormente del pipe, con lo que libera a procesos que estén esperando que él acabe.

----------TOKENIZACION DE UN STRING EN BASH----------
Si tenemos un string donde sus elementos (numéricos o texto) están separados por espacio, con la siguiente instrucción podemos generar un array donde los componentes del array son los tokens que componen el string:

**Tokens de un string:**

```
$ my_string="uno dos tres"
$ my_array=( $my_string )    %Observar los espacios dentro de los paréntesis.

%my_array es un array que tiene 3 componentes
%   my_array[0]=uno
%   my_array[1]=dos
%   my_array[2]=tres

Otro ejemplo con números
$ my_string="1 3 2 5 67 3"
$ tokens=( $my_string )    %Observar los espacios dentro de los paréntesis.

%tokens es un array que tiene 6 componentes
%   tokens[0]=1
%   tokens[1]=3
%   tokens[2]=2
%   tokens[3]=5
%   tokens[4]=67
%   tokens[5]=3
```

----------RESUMEN USO DE ARRAYS EN BASH----------

**Declaración, inicialización y asignación:**

```
$ declare -a my_array   %Declaración explicita del array. No es obligatoria
$ my_array=(foo bar)    %Declara/Inicializa el array my_array con los elementos foo y bar
$ my_array[0]=foo       %Asigna el valor foo al índice 0 del array. Empiezan en 0.
```

**Referencia a los elementos del array:**

```
$ echo ${my_array[@]}     %Devuelve todos los elementos como strings independientes.
$ echo ${my_array[*]}     %Devuelve todos los elementos en un string.
       Ejemplos:
              $ for i in "${my_array[@]}";   %Iteramos por los elementos del array.
              do
                 echo "$i";
              done

              Salida:
              foo
              bar
              --------------------------
              $ for i in "${my_array[*]}";
              do
                 echo "$i";
              done

              Salida:
              foo bar
              --------------------------

       $ echo ${!my_array[@]}       %Devuelve los índices del array
       Ejemplo:
              $ my_array=(foo bar baz)
              $ for index in "${!my_array[@]}";
              do
                 echo "$index";
              done

              Salida:
              0
              1
              2
```

**Número de elementos del array:**

```
$ echo ${#my_array[*]}   %Devuelve el número de elementos del array.
```

**Añadir elementos al array:**

Con += podemos añadir elementos al final del array.

Ejemplos array indexados:

```
$ my_array=(foo bar)
$ my_array+=(baz)
$ echo "${my_array[@]}"
foo bar baz

$ my_array=(foo bar)
$ my_array+=(baz foobar)   %se pueden añadir varios a la vez
$ echo "${my_array[@]}"
foo bar baz foobar
```

**Iterar por los elementos de un string usando tokens y bucle for estilo C:**

```
$ my_string="10 11 12 13"
$ tokens=( $my_tring )
$ num_tokens=${#tokens[*]}
$ for (( i=0; i<num_tokens; i++ ))
  do
    echo ${tokens[$i]}
  done
```

**10**
**11**
**12**
**13**