Universidad Carlos III de Madrid	**UNIVERSIDAD CARLOS III DE MADRID** **DEPARTAMENTO DE INFORMÁTICA** **GRADO EN INGENIERÍA INFORMÁTICA.** **SISTEMAS OPERATIVOS**	

ATENCIÓN: Dispone de **3 horas** para realizar la prueba.

Ejercicio 1. Teoría [2,5 puntos]:

Pregunta 1.
¿Cuándo entra un proceso en estado zombie?
A.- Cuando muere su padre y él no ha terminado todavía.
B.- Cuando muere su padre sin haber hecho `wait` por él.
C.- Cuando él muere y su padre no ha hecho `wait` por él.
D.- Cuando él muere y su padre no ha terminado todavía.

Razone la repuesta.

Pregunta 2.
¿Cuál de las siguientes políticas de planificación es más adecuada para un sistema de multiproceso de tiempo compartido?
A.- Primero el trabajo más corto.
B.- Round-Robin.
C.- Prioridades.
D.- FIFO.

Razone la repuesta.

Pregunta 3.
¿Cuál es la diferencia entre nombre absoluto y relativo? Indique dos nombres relativos para `/ssoo/alumnos/examen/enunciado`. Indique el directorio respecto al que son relativos.

Ejercicio 2 [2,5 puntos]:

La **criba de Eratostenes** es un algoritmo para generar una serie de números primos consecutivos. Permite hallar todos los números primos menores que un número natural dado *N*. Para ello se forma una tabla con todos los números naturales comprendidos entre 2 y *N* y se van tachando los números que no son primos de la siguiente manera: cuando se encuentra un número entero que no ha sido tachado, ese número es declarado primo, y se procede a tachar todos sus múltiplos. El proceso termina cuando el cuadrado del mayor número confirmado como primo es mayor que *N*.

En este ejercicio se pide implementar la criba de Eratóstenes con ***procesos y pipes***.
El algoritmo debe funcionar de la forma siguiente:

- Un padre crea N hijos.
- El padre se conecta con el hijo 0.
- El hijo **i** se conecta con los hijos i-1 e i+1.
- El padre genera una serie de números consecutivos 2, 3, 4, 5,… y los envía uno a uno al hijo 0. La secuencia termina con el número n-1.

- Cada hijo almacena el primer número que recibe en una variable primo_local. Posteriormente, si el número que recibe no es múltiple de primo_local, lo reenvía al siguiente hijo.
- Cuando un hijo recibe un -1, lo pasa al hijo siguiente y termina.
- El padre espera a que terminen todos los hijos antes de terminar.

Ejercicio 3 [2,5 puntos]:

Se tiene un **array de 100** elementos en el que se quiere realizar varias **iteraciones** en cada una de las cuales se debe colocar en **cada casilla la media de la suma del contenido de esa casilla y sus dos adyacentes**.

Es decir que para cualquier casilla del array entre 1 y 98 el nuevo valor de la casilla será **v[i]= (v[i-1]+v[i]+v[i+1])/3**; para la casilla 0 se supondrá que su adyacente izquierda es la casilla 99 y para la casilla 99 se supondrá que su adyacente derecha es la casilla 0.

El procedimiento se aplicará durante 200 iteraciones y para optimizarlo se desea que **la mitad del array la procese un thread y la otra mitad otro thread.**

Las **operaciones de cálculo de los nuevos valores** se realizarán **en un array auxiliar** y sólo cuando los dos threads hayan terminado su iteración volcarán los valores de éste sobre el real para continuar con la siguiente iteración.

Por tanto el **procedimiento** que seguirá **cada thread** es:

1. **Copiar los nuevos valores** en el array auxiliar
2. Cuando haya terminado de copiarlos deberá **esperar a que termine el otro thread antes de volcar los datos del array auxiliar en el array real.** De esta forma no se modifican los datos de una iteración antes de que el otro utilice la casilla que tienen en común
3. Una vez copiados los datos en el array real deberá **esperar a que el otro thread termine también de copiar los datos auxilares sobre el array real** antes de proceder a la siguiente iteración del paso 1. Al igual que en el paso 2, hay que esperar para que las casillas comunes estén actualizadas.

A continuación se da la estructura básica del programa. **Se pide añadir la sincronización en los apartados donde se indica**. ¶ **y** • ¸ de los hilos 1 y 2·¸

```
#define TAM 100
#define NUMITER 200
pthread_attr_t attr;
```

```
pthread_t idth[2];
float v[TAM];

// ¶ AÑADIR LAS VARIABLES QUE SE NECESITEN

void rellenarArray(){  //Dependiente del programa. No lo tiene que rellenar el alumno  }

void *hilo0(void *num) {
  int i,j;
  float vaux[TAM/2];
```

- **Hilo 0**/AÑADIR LAS VARIABLES QUE SE NECESITEN

```
  for (j=0; j<NUMITER; j++) {
          vaux [0]= (v[TAM-1]+v[0]+v[1])/3;
          for (i=1; i<TAM/2 ; i++)
           vaux [i]= (v[i-1]+v[i]+v[i+1])/3;
```

ی **Hilo 0**//REALIZAR LAS OPERACIONES DE SINCRONIZACIÓN NECESARIAS Y LA COPIA DE LOS DATOS AL ARRAY REAL

```
         }

  pthread_exit(0);

}

void *hilo1(void *num) {
   int i,j;
   float vaux[TAM/2];
```

- **Hilo 1**//AÑADIR LAS VARIABLES QUE SE NECESITEN

```
   printf ("Hilo 1\n");

    for (j=0; j<NUMITER; j++) {

        for (i=TAM/2; i<TAM-1 ; i++)
            vaux [i-TAM/2]= (v[i-1]+v[i]+v[i+1])/3;
         vaux [TAM/2-1]= (v[TAM-2]+v[TAM-1]+v[0])/3;
```

ی **Hilo 1**//REALIZAR LAS OPERACIONES DE SINCRONIZACIÓN NECESARIAS Y LA COPIA DE LOS DATOS AL ARRAY REAL

```
       }

    pthread_exit(0);
```

```
}

int main(){

   int i;
   rellenarArray();
  pthread_mutex_init (&mtx1, NULL);
   pthread_mutex_init (&mtx2, NULL);
   pthread_attr_init(&attr);
   pthread_create(&idth[0],&attr,hilo0,NULL);
   pthread_create(&idth[1],&attr,hilo1,NULL);
   for (i=0; i<2; i++)
       pthread_join(idth[i],NULL);
return(0);

}
```

Ejercicio 4 [2,5 puntos]:

Suponiendo que estamos en un sistema de ficheros UNIX., cuyos i-nodos dispone de:

- Punteros directos a bloques (16).
- Punteros indirecto simple (2).
- Punteros indirecto doble (2).

Además, el tamaño de bloque es de 1024 bytes y el tamaño de un índice es de 2 bytes. Suponga para todos los casos que sólo se encuentra precargado en memoria el i-nodo.

Responda a las siguientes preguntas:

a. ¿Cuantos accesos a disco son necesarios para leer completamente un fichero cuyo tamaño es de 10 MB? Asuma que una vez leído un bloque, este se encuentra en cache.

b. Considerando únicamente la estructura de i-nodos ¿Cuál es el tamaño máximo de un fichero en este sistema de ficheros?

c. ¿Cual es el tamaño máximo de un dispositivo de almacenamiento usando este sistema de ficheros?

d. ¿Qué ocurriría en este sistema de ficheros al aumentar el tamaño del bloque de datos? Explique las ventajas e inconvenientes de esta modificación.

SOLUCIONES

TEORIA 1.

La solución correcta es la C.

Cuando el proceso hijo termina, debe informar de este hecho a su proceso padre. Para que el padre pueda tratar este evento debe estar esperando la terminación de ese hijo concreto mediante `waitpid()` o el de cualquiera de sus hijos mediante `wait()`. Cuando el hijo envía su estado de terminación al padre se liberan todos sus recursos.

Si por algún motivo, cuando el hijo termina el padre no esta esperando, es decir no ha hecho `wait()` por él, surge el problema de que el hijo no puede transmitirle su estado de terminación y debe seguir existiendo hasta que el padre ejecute el `wait()`. En este caso se dice que el proceso hijo esta en estado zombie. Puede ser un problema en el caso que el padre no ejecute nunca el `wait()` ya que el proceso hijo no se liberara jamás, ocupando espacio en memoria inútilmente.

En el caso de que el padre termine sin haber ejecutado `wait()`, surge el problema de que hacer con sus hijos, tanto los que están en estado zombie como los que no. Este problema se soluciona asignando a todos los procesos huérfanos un proceso padre que se sabe seguro que existe. En el caso de POSIX, la norma es asignarlos como padre al proceso INIT.

TEORÍA 2.

Un sistema multiproceso de **tiempo compartido** permite ejecutar varios procesos de distintos usuarios en el computador de forma concurrente, con lo que los usuarios tienen la sensación de que tienen todo el computador para cada uno de ellos. Para poder implementar estos sistemas de forma eficiente es imprescindible tener un sistema multiproceso y un planificador que permita cambiar de un proceso a otro según los criterios de la política exigida. Ahora bien, los criterios de la política de planificación influyen mucho sobre el comportamiento del computador de cara a los usuarios, por lo que es importante decidir que política se debe usar en cada caso.

La política de **Round Robin** realiza un reparto equitativo del procesador dejando que los procesos ejecuten durante las mismas unidades de tiempo (rodaja). Con este mecanismo se

encarga de repartir el tiempo de UCP entre los distintos procesos, asignando de forma rotatoria intervalos de tiempo de la UCP a cada uno de ellos. Esto algoritmo es especialmente **adecuado para los sistemas de tiempo compartido** por que se basa en el concepto de rodaja de tiempo y reparte su atención entre todos los procesos, lo que al final viene a significar entre todos los usuarios. Los procesos están organizados en forma de cola circular y cuando han consumido su rodaja de tiempo son expulsados y pasan a ocupar el último lugar en la cola, con lo que se ejecuta otro proceso. Con este mecanismo, los usuarios tienen la sensación de avance global y continuado, lo que no está garantizado con los otros algoritmos descritos.

La política del **Primero Trabajo Mas Corto** consiste en seleccionar para ejecución el proceso listo con tiempo de ejecución mas corto, por lo que exige conocer a priori el tiempo de ejecución de los procesos. Este algoritmo no plantea expulsión: el proceso sigue ejecutándose mientras lo desee. Además tiene puede tener problemas de inanición de procesos. **No es adecuado** para tiempo compartido.

Un algoritmo de planificación **FIFO** ejecuta procesos según una política "primero en entrar primero en salir": un proceso pasa al final de la cola cuando hace una llamada al sistema que lo bloquea y si esto no ocurre el proceso se ejecutará de forma indefinida hasta que acabe. No se plantea la expulsión del proceso: el proceso ejecuta hasta que realiza una llamada bloqueante al Sistema Operativo o hasta que termina. Este algoritmo es **inadecuado** para tiempo compartido porque no hay ninguna seguridad en cuanto al tiempo de respuesta a los usuarios, que puede ser muy lento.

Con la política de **Prioridades** se selecciona para ejecutar el proceso en estado listo con mayor prioridad. Se suele asociar a mecanismos de expulsión para que un proceso abandone el procesador cuando pasa a listo un proceso de mayor prioridad. Puede tener problemas de inanición de procesos. **No es adecuado** para tiempo compartido si no se asocian políticas de rodaja de tiempo a las colas de prioridad y no se evitan los posibles problemas de inanición.

TEORÍA 3.

La especificación de un nombre de archivo en un árbol de directorios, o en un grafo acíclico, toma siempre como referencia el nodo raíz. A partir de este directorio es necesario viajar por los sucesivos subdirectorios hasta encontrar el archivo deseado. Para ello el sistema operativo debe obtener el nombre completo del archivo a partir del directorio raíz. Hay dos formas de obtenerlo:

- El usuario introduce el nombre completo del archivo (nombre absoluto).
- El usuario introduce el nombre de forma relativa (nombre relativo a algún subdirectorio)

El nombre absoluto proporciona todo el camino a través del árbol de directorios desde la raíz hasta el archivo. Es autocontenido pues proporciona toda la información necesaria para llegar al archivo, sin que se necesite ninguna información añadida por parte del entorno del proceso o interna al sistema operativo. Sin embargo, los nombres relativos sólo proporcionan un porción del camino, a partir de un determinado subdirectorio (el directorio actual de trabajo). Éstos no se pueden interpretar si no se conoce el directorio del árbol a partir del que empiezan, para ello existe el directorio actual o de trabajo, a partir del cual se interpretan siempre los nombres relativos. Para evitar errores es necesario que el usuario se coloque en el directorio de trabajo adecuado antes de usar nombres relativos al mismo. Por ejemplo:

- El nombre absoluto es `/ssoo/alumnos/examen/enunciado`
- El nombre relativo respecto al subdirectorio `/ssoo/alumnos` es `examen/enunciado`
- El nombre relativo respecto al subdirectorio `/ssoo/alumnos/examen` es `enunciado`

EJERCICIO 1.

```
#define N 10

int main()
{
  int p[N][2];
  int i;
  // CREATE PIPES
  for (i=0;i<N;i++)
    pipe(p[i]);

  for (i=0;i<N;i++){
    if (!fork()) {
      int local_prime, y;
      close(p[i][1]);
      close(p[i+1][0]);

      // FIRST NUMBER TO BE RECEIVED IS A PRIME NUMBER
      read(p[i][0], &local_prime, 4);
      printf(" CHILD %d received PRIME NUMBER %d\n",
getpid(), local_prime);

      while (1) {
             read(p[i][0], &y, 4);

             // FORWARD A NUMBER ONLY IF IT IS NOT MULTIPLE
OF THE LOCAL PRIME NUMBER
```

```
                if ((y%local_prime) &&(i < N-1))
                     write(p[i+1][1], &y, 4);
                // FINISH WHEN RECEIVING -1
                if (y == -1)
                     break;
          }
          exit(0);
        }
      }

      // SENDING A STREAM OF NUMBERS
      for (i=2;i<N*10;i++)
         write(p[0][1],&i,4);
      // SENDING -1 TO TERMINATE
      i=-1;
      write(p[0][1],&i,4);

      // WAIING FOR THE CHILDREN TO TERMINATE
      for (i=0;i<N;i++){
        int status,pid;
        pid=wait(&status);
        printf(" CHILD %d terminated. \n", pid);
      }
}
```

EJERCICIO 4.

```
// compilar con  gcc -lpthread mediavector.c
// José Manuel Pérez Lobato
pthread_mutex_t mtx1,mtx2;
pthread_cond_t varcond1,varcond2;
int contfin1=0, contfin2=0;
```

Hilo 0

```
  int yoinicializo1=0, yoinicializo2=0;
```

Hilo 0

```
   pthread_mutex_lock (&mtx1);
     contfin1++;
     //El que  va a esperar en el wait será el encargado de
inicializar la variable contfin1 para la siguiente iteración
     if (contfin1==1) yoinicializo1=1;
   //Esperar que termine el otro si no ha terminado
     while (contfin1!=2)
       pthread_cond_wait(&varcond1, &mtx1);
```

```
  for (i=0; i<TAM/2 ; i++)
    v[i]=vaux[i];
  pthread_cond_signal(&varcond1);
  if (yoinicializo1){
    contfin1=0;
    yoinicializo1=0;
  }
pthread_mutex_unlock (&mtx1);

pthread_mutex_lock (&mtx2);
  contfin2++;
  if (contfin2==1) yoinicializo2=1;
//Esperar  a que el otro copie los datos
  while (contfin2!=2)
       pthread_cond_wait(&varcond2, &mtx2);
  pthread_cond_signal(&varcond2);
  if (yoinicializo2){
    contfin2=0;
    yoinicializo2=0;
  }
pthread_mutex_unlock (&mtx2);
```

Hilo 1

```
int yoinicializo1=0, yoinicializo2=0;
```

Hilo 1

Este apartado es idéntico al del hilo 0 excepto en el bucle de copia de datos que será el siguiente:

```
for (i=TAM/2; i<TAM ; i++)
    v[i]=vaux[i-TAM/2];
```

EJERCICIO 5.

a. Calculamos el número de bloques de datos leídos:
   10 MB/1024 = (10*2^20)/2^10 = 10*2^10 bloques de datos

   Calculamos cuantos datos direcciona un bloque de direcciones simple:
   (1024/2)*1024= (2^10/2)*2^10 = 2^19 = 512 KB

   Calculamos cuantos datos direcciona un bloque de direcciones doble:
   (1024/2)*(1024/2)*1024 = (2^10/2)*(2^10/2)*2^10 = 2^28 = 256 MB

Ahora calculamos cuando bloques de direcciones se necesitan para un fichero de 10 MB:
10MB/512KB = (10*2^20)/2^19=20 bloques de direcciones

Lo que supone en total:

10*2^10 accesos a bloques de datos + 2 accesos a bloques de direcciones simples + 1 acceso un bloque de direcciones doble + 18 accesos a los bloques de direcciones simples del bloque de direcciones dobles

b. (16*2^10)+ 2*((2^10/2)*2^10) + 2*((2^10/2)*(2^10/2)*2^10) =
16KB + 2*512KB + 2*256MB = 16KB + 1MB + 512MB

c. En el sistema de ficheros se pueden direccionar hasta 2^16 bloques (el tamaño de las direcciones es 2 bytes). Luego el sistema de ficheros solo puede usarse en dispositivos de almacenamiento no mayores de 2^16*1KB = 64 MB

d. Consecuencias:
   - Aumentaría el tamaño ideal del fichero.
   - Aumentaría el tamaño del dispositivo donde se puede usar el sistema de ficheros.
   - Mejoraría el rendimiento (menos accesos a disco para leer la misma cantidad e datos)
   - Aumentaría la posibilidad de desperdicio de espacio (fragmentación interna).