UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática

1	2	3	4	test	extra	NOTA

Fundamentos de los Sistemas Operativos
Convocatoria especial - 17 de noviembre de 2014

Nombre y apellidos	**DNI/NIE**
SOLUCIONES	

DURACIÓN: Dispone de tres horas para realizar el examen (el test y estos cuatro ejercicios)

1 (1,25 puntos) Un sistema de memoria virtual paginada utiliza páginas de 1 KiB y direcciones lógicas de 32 bits. En un momento dado, en la tabla de páginas de un proceso se encuentra la siguiente información: la página 0 está cargada en el marco 5, la página 1 en el marco 4 y la página 3 en el marco 15.

Bajo estas condiciones, ¿con qué dirección lógica se corresponde la dirección física 1024?

La dirección física 1024 se corresponde con el byte 0 del marco 1 de memoria. Según la información proporcionada en el enunciado, no se nos da información sobre la página lógica que se encuentra cargada en ese marco de página, por lo que no es posible averiguar la dirección lógica con la que se corresponde dicha dirección física.

2 (1,25 puntos) En un sistema operativo se utiliza una estructura de *inodes* parecida a la de Unix. Los bloques de datos son de 1024 bytes. Las entradas en los *inodes* dedican 64 bits al tamaño del archivo y 16 bits a los punteros de los bloques. Un *inode* tiene ocho entradas de direccionamiento directo, una de direccionamiento indirecto simple y otra de direccionamiento indirecto doble.

Sobre esta descripción, indique si cada una de estas afirmaciones es verdadera o falsa. Debe justificar sus respuestas:

a) El tamaño máximo de un fichero es de 264 bloques.

La afirmación es incorrecta.

Se indican en el enunciado varios parámetros que pueden condicionar el tamaño máximo de un fichero. Por tanto, es conveniente analizarlos todos ya que el tamaño máximo estará condicionado por aquel parámetro más restrictivo:

· 64 bits del *inode* dedicados al tamaño del archivo: un archivo podría tener un máximo de 2^{64} bytes, es decir, 2^{54} bloques.

· 16 bits para los punteros a bloques: podremos direccionar hasta 2^{16} bloques diferentes.

· Según la estructura del *inode*, un fichero que ocupe todo el espacio tendrá 8+512+512+(512×512) bloques, es decir, aproximadamente 2^{18} bloques.

Cualquiera de estas cantidades supera con creces el tamaño de 264 bloques.

b) El número máximo de bloques asignados a un archivo en su *inode* es: 262 664.

La afirmación es incorrecta.

El número de bloques de datos que puede llegar a tener un archivo es de 8+512+512+(512×512)=262664. A esto hay que sumar los bloques asignados al archivo para mantener sus tablas de índices, que en el peor caso serían 1 indirecto simple + 1 maestro del indirecto doble + 512 indirectos dobles = 514 bloques de índices. La cantidad total, pues, es superior a la que indica el enunciado.

c) El tamaño máximo de un archivo que utiliza todo el disco es de 64 MiB.

Respuesta variable.

Para empezar, no se nos da el dato del tamaño del disco, con lo cual no se puede calcular una cifra. En segundo lugar, lo normal sería que las estructuras de control del sistema de ficheros se lleven una

parte del espacio físico, con lo cual no puede existir un archivo «que utiliza todo el disco», al menos en el sentido literal de la afirmación. Si se toman estas consideraciones, la respuesta debería ser «la afirmación no tiene sentido» o «no hay datos suficientes para calificar la afirmación».

Si por «disco» entendemos «espacio direccionable», serían 2^{16} bloques de 1024 bytes, por tanto 64MiB. A este espacio ocupado habría que descontarle los bloques necesarios para mantener los índices, que haría que la longitud del archivo esté por debajo de esos teóricos 64MiB. No obstante, si la expresión «tamaño de un archivo» se entiende como el «espacio ocupado por un archivo», sí podría considerarse correcta la afirmación.

---

3 (1 punto) A continuación se muestran las demandas futuras de CPU y E/S de un conjunto de procesos secuenciales que arriban simultáneamente a un ordenador con un solo procesador. En cada casilla se muestra el tiempo requerido para completar una petición de CPU o E/S, en unidades de tiempo arbitrarias. Las casillas vacías indican que no hay peticiones futuras.

Por ejemplo, el proceso 3 demanda 2 unidades de tiempo de CPU, luego pedirá 4 unidades de E/S, tras lo que necesita 1 unidad de tiempo de CPU, atendida la cual finaliza.

	CPU	E/S	CPU
Proceso 1	7	1	6
Proceso 2	4	4	4
Proceso 3	2	4	1
Proceso 4	1	7	1

Obtener el diagrama de Gantt, el tiempo medio de retorno y el tiempo medio de espera al aplicar Round-Robin con cuanto igual a 3 unidades de tiempo como política de planificación.

Ver solución al final del documento.

---

4 (1,5 puntos) Queremos procesar una imagen de tamaño N×N, siendo N una potencia de 2. El objetivo es aplicar cierta operación matemática a cada uno de los píxeles de la imagen. Para ello disponemos ya de una función **`Process_Pixels(…)`**, descrita más abajo. El procesamiento se hará concurrente, mediante M hilos (M es una potencia de 2). Cada hilo ejecutará la misma rutina, llamada **`ImageThread()`**, sin argumentos, que debe usted implementar. La rutina debe estar escrita de forma que cada hilo procese un subconjunto de píxeles, garantizando que todos los píxeles de la imagen quedan procesados y que no se procesa un mismo píxel varias veces. Utilice semáforos en caso de que sea necesario aplicar alguna sincronización entre hilos.

Considere que tanto M como N son cantidades constantes y conocidas de antemano.

`Process_Pixels (row,col,npixels)` procesa *npixels* consecutivos de la fila *row* a partir de la columna *col* de la imagen. Esta función no tiene ningún control de concurrencia interno.

Nombre y apellidos	DNI/NIE

SOLUCIÓN

Esta es una de las soluciones más simples para el problema. Consiste en que cada hilo toma un bloque de filas contiguas de la imagen. El número de filas por hilo es N/M, que es un número exacto según se ha explicado en el enunciado. Aquí está escrita en lenguaje C.

```
// constantes M y N. Los valores son ejemplos.
#define N 512
#define M 8
#define RowsPerThread (N/M)

// Variable global que señala la primera fila que está
// pendiente de procesarse
int current_row=0;

// Semáforo para garantizar exclusión mutua en la
// manipulación de current_row
Semaforo mutex=1;

// Rutina que ejecutan los hilos
void ImageThread() {

      // Toma la fila inicial que le toca a este hilo
      wait(mutex);
      int start_row = current_row;
      int end_row   = current_row + RowsPerThread;

      // Actualiza la fila inicial para el siguiente hilo que elija
      current_row = end_row;
      signal(mutex);

      // Procesa las filas que le tocan al hilo
      for ( int row=start_row; row<end_row; row++ ) {
        Process_Pixels (row,0,N);
      }
}
```

Esta es otra solución más sofisticada, en la que cada hilo va tomando bloques de *slice* píxeles, empezando desde la posición (0,0). Para simplificar los cálculos, se asume que el valor de *slice* es un divisor exacto de N, para que sean más simples los cálculos.

La solución aquí mostrada tiene varias ventajas sobre la anterior: no necesita conocer cuál es el número de hilos que participan y rinde mejor tiempo de ejecución si hay diferencias grandes de velocidad entre los hilos.

```
// Constantes del algoritmo. Los valores son ejemplos.
#define N 512
#define SLICE 64
#assert (N % SLICE) == 0

// Variables que indican el siguiente píxel de la imagen a procesar
int current_row=0, current_col=0;

// Variable que indica el número de píxeles que quedan por asignar
int pending_pixels = N*N;

// Semáforo para garantizar exclusión mutua en el acceso a las
// variables compartidas
Semaforo mutex=1;

// Rutina que ejecutan los hilos
void ImageThread() {
  while (true) {
    // Cogemos un trozo de imagen para procesar
    wait(mutex);
    // si ya no quedan píxeles pendientes, salimos
    if (pending_pixels==0) {
        signal(mutex);
        break;
    }

    // Este hilo tomará a partir de (current_row,current_col)
    int myrow = current_row;
    int mycol = current_col;

    // Actualizamos los valores de current_row y current_col
    if ( current_col + SLICE == N ) {
      current_row++;
      current_col = 0;
    }
    else {
      current_col += SLICE;
    }
    pending_pixels -= SLICE;

    signal(mutex);

    Process_Pixels(myrow,mycol,SLICE);
  }
}
```