UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática

1	2	3	test	extra	NOTA

Fundamentos de los Sistemas Operativos
Examen parcial - 10 de abril de 2015

Nombre y apellidos **DNI/NIE**

SOLUCIONES	

DURACIÓN: *Dispone de dos horas para realizar el examen.*

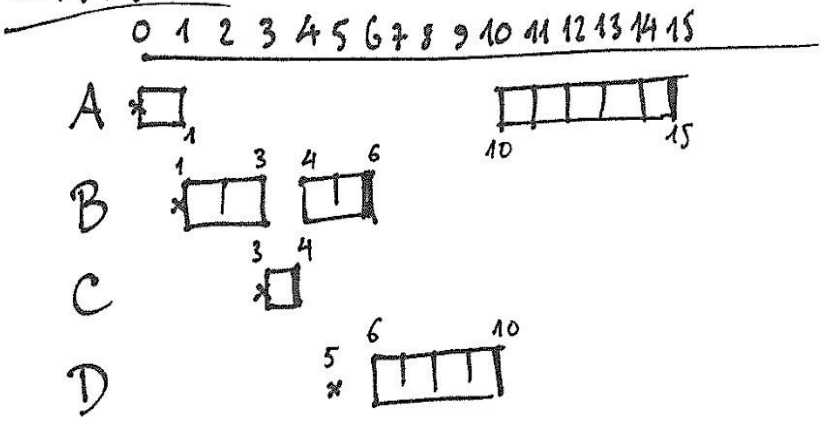
Lea las instrucciones para el test en la hoja correspondiente.

1 (1'25 puntos) A un planificador de CPU llegan cuatro procesos, según el cuadro adjunto. Aplique las dos políticas SRTF y RR (q=2) y, para cada una de ellas, obtenga lo siguiente:

- Diagrama de Gantt o similar con la planificación.
- Tiempo de espera y de retorno de cada uno de los procesos.
- Número de cambios de contexto realizados durante la planificación.

proceso	llegada	duración
A	0	6
B	1	4
C	3	1
D	5	4

SRTF

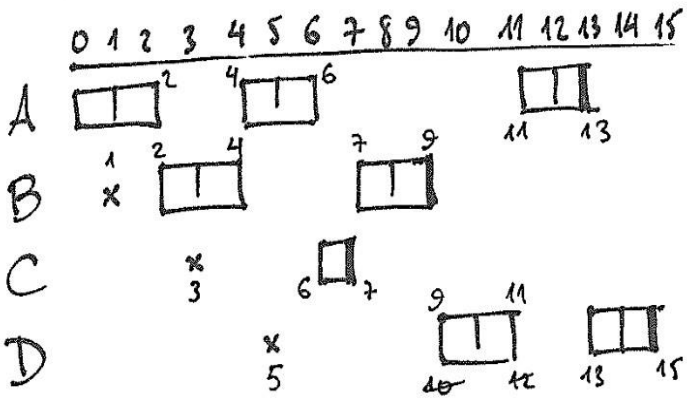


Tiempo de espera
A = 9
B = 1
C = 0
D = 1

Tiempo de retorno
A = 15
B = 5
C = 1
D = 5

cambios de contexto = 5

RR q=2



Tiempo de espera
A = 7
B = 4
C = 3
D = 6

Tiempo de retorno
A = 13
B = 8
C = 4
D = 10

cambios de contexto = 7

2 (1 punto) Los diseñadores de sistemas operativos, cuando idean políticas para gestionar el acceso a un recurso, no pueden en general encontrar un algoritmo que optimice al mismo tiempo el rendimiento, la seguridad y la justicia en el reparto del recurso: deben encontrar algoritmos que lleguen a un balance entre todos estos criterios.

TAREA. Ilustre este problema con un ejemplo concreto. No use más de 200 palabras en su exposición.

Hay varios casos que se han visto en la asignatura y que se pueden aprovechar para responder a esta cuestión. Algunos ejemplos:

Lo que ocurre cuando queremos planificar la CPU: si queremos optimizar el tiempo de espera medio, el mejor algoritmo es el SJF, pero a cambio perjudica enormemente a los procesos de larga duración. Si queremos dar un reparto justo y equilibrado, podemos intentar un Round Robin, pero incurre en un mayor número de cambios de contexto y por tanto disminuye el rendimiento de la CPU respecto a SJF. No hay una política que sea óptima en rendimiento y en justicia a la vez.

También está el caso de los algoritmos de sección crítica, que los podemos diseñar muy sencillos, pero sin ninguna garantía de que se cumpla la exclusión mutua. En este caso, la seguridad nos hace aumentar la complejidad de nuestras políticas. En el caso de la instrucción "test-and-set" y similares, es fácil diseñar un algoritmo de sección crítica que cumpla sólo la exclusión mutua, pero asegurar la espera limitada exige un algoritmo más complejo y que tarda más en ejecutarse.

Los mismos casos anteriores se pueden aplicar a cualquier recurso al que varios procesos quieran acceder, pero que se deba utilizar de forma excluyente. Por ejemplo, una impresora. Podemos gestionar la cola de acceso de manera que se saque más trabajo por unidad de tiempo, pero a costa de un reparto injusto. Etcétera.

Nombre y apellidos **DNI/NIE**

3 (1'75 puntos) En una agencia de meteorología tenemos un programa multihilo que se encarga de lanzar cinco hilos que intentan estimar por distintos métodos la temperatura que hará mañana en Las Palmas. En cuanto dos de los cinco hilos hayan terminado, otro hilo calculará la media de sus respectivas estimaciones y esa será la predicción definitiva.

Los hilos utilizan un vector global para anotar sus estimaciones, que serán leídas por el hilo predictor. El cuadro adjunto muestra el esquema general del código.

```
// variables globales
float estimaciones[5] = {∞,∞,∞,∞,∞};
float predicción = ∞;
```

```
hilo estimador(int i) { // i=0..4
  - Calcula la estimación por el método "i"
  estimaciones[i] = valor estimado
}
```

```
hilo predictor() {
  - Espera a que terminen dos estimaciones
  - Calcula la media de las dos estimaciones
  predicción = media calculada
}
```

TAREA. Tiene usted que añadir el código que sea necesario para sincronizar al hilo predictor con los hilos estimadores, utilizando semáforos como herramientas de sincronización.

Si lo considera necesario, puede crear nuevas variables o cambiar el código, siempre que se mantenga la arquitectura general aquí expuesta.

Para sincronizar los procesos, basta con utilizar un semáforo inicializado a cero, sobre el que el proceso predictor hace dos operaciones WAIT consecutivas. Cada proceso estimador, según termina, ejecuta una operación SIGNAL sobre el semáforo. De esa forma se garantiza que el predictor sigue adelante cuando dos estimadores han finalizado.

Se adjunta un ejemplo de código que implementa esta estrategia. También se implementa la recogida de las estimaciones desde el vector compartido.

```
// Este semáforo sirve para señalizar que un estimador ha terminado
Semáforo terminado = 0;

hilo estimador(int i) {
  // Calcula la estimación por el método "i"
  estimaciones[i] = valor_estimado;

  // Avisa de que ha terminado
  SIGNAL(terminado);
}

hilo predictor() {
  // espera a que terminen al menos dos estimadores
  WAIT(terminado);
  WAIT(terminado);

  // calcula la media de las dos estimaciones
  // recorre la lista y encuentra dos estimaciones ya hechas
  // este código no es necesario para superar el ejercicio, aunque si se
  // gestiona la recogida de resultados, se obtendrá una major calificación
  // OJO – en este código se asume que no hay problemas
  // de acceso concurrente a estimaciones[]

  float suma = 0.0;
  int encontrados=0;
  for (int i=0;i<5;i++) {
    if (estimaciones[i] != ∞) {
      suma+=estimaciones[i];
      encontrados++;
      if (encontrados==2) break;
    }
  }
  float media = suma/2.0;

  predicción = media;
}
```

Nombre y apellidos	DNI/NIE

Otra alternativa, utilizando un contador de procesos terminados y protegiendo el acceso al vector estimados[] mediante un mutex:

```
// Este semáforo sirve para avisar al predictor
Semáforo puede_predecir = 0;
// Contador de estimadores terminados
int terminados = 0;
// Mutex para proteger el código crítico
Semáforo mutex = 1;

hilo estimador(int i) {
  // Calcula la estimación por el método "i"

  // Escribe el resultado y avisa al predictor (si es el caso)
  WAIT(mutex);
    estimaciones[i] = valor_estimado;
    terminados++;
    if ( terminados==2 ) SIGNAL(puede_predecir);
  SIGNAL(mutex);
}

hilo predictor() {
  // espera a que terminen al menos dos estimadores
  WAIT(puede_predecir);

  // calcula la media de las dos estimaciones
  // recorre la lista y encuentra dos estimaciones ya hechas
  // en este caso, aprovechamos el mutex para proteger el acceso al vector

  float suma = 0.0;
  int encontrados=0;
  for (int i=0;i<5;i++) {
    WAIT(mutex);
    float dato = estimaciones[i];
    SIGNAL(mutex);
    if (dato != ∞) {
      suma+=dato;
      encontrados++;
      if (encontrados==2) break;
    }
  }
  float media = suma/2.0;
  predicción = media;
}
```