Grupo: …… NIA: …… Nombre y apellidos: ……..

Ejercicio 1

Considérese un sistema operativo que usa un algoritmo de planificación de procesos *round-robin* con una rodaja de 100 ms. Supóngase que se quiere compararlo con un algoritmo de planificación expulsiva por prioridades en el que cada proceso de usuario tenga una prioridad estática fijada en su creación. Dado el siguiente fragmento de programa, se pide analizar su comportamiento usando el planificador original y, a continuación, hacerlo con el nuevo modelo de planificación planteado. Para cada modelo de planificación, se deberá especificar la secuencia de ejecución de ambos procesos (se tendrán en cuenta sólo estos procesos) hasta que, o bien un proceso llame a la función P2 o bien el otro llame a P4.

NOTA: La escritura en una tubería no bloquea al escritor a no ser que la tubería esté llena (situación que no se da en el ejemplo). Además, en este análisis se supondrá que a ninguno de los dos procesos se les termina el cuanto de ejecución.

```
...

f = open ("/dev/tty", RD_RDONLY);
pipe(p);

/* crea un hijo (en el caso del planificador modificado de menor
prioridad que el padre, en este caso la llamada sería:
fork(LOWERPRIORITY)) */

if (fork()==0){
      P1()   /* procesamiento de 20 ms.   */
      write   (p[1],   buf,   t);
      P2();
      ...
} else {
      /*   lectura del terminal   */
      read   (f, buf, t);   /*   estará disponible   en   5   ms   */
      P3();   /*  procesamiento de   2   ms   */
      read   (p[0] ,   buf,   t);
      P4();
      ...
}
```

Grupo: …… NIA: …… Nombre y apellidos: ……..

SOLUCIÓN

En primer lugar, se presenta la secuencia de ejecución para el algoritmo original:

- P. padre: Ejecuta fork (no hay cambio de contexto ya que no se bloquea), inicia la lectura del terminal y se bloquea ya que los datos no están disponibles.
- P. hijo: Comienza a ejecutar P1 (5ms.)
- Datos del terminal disponibles: se pone en estado de listo para ejecutar el proceso padre pero, a continuación, sigue ejecutando el proceso hijo.
- P. hijo: Continúa ejecutando P1 y después escribe en la tubería. Como no está llena no se bloquea y pasa a ejecutar P2.

Siguiendo las especificaciones del enunciado, con esto termina la traza de la ejecución.

A continuación, se muestra la secuencia de ejecución para el algoritmo modificado:

- P. padre: Ejecuta fork (no hay cambio de contexto ya que el hijo es de menor prioridad), inicia la lectura del terminal y se bloquea ya que los datos no están disponibles.
- P. hijo: Comienza a ejecutar P1 (5ms.).
- Datos del terminal disponibles: Se pone en estado de listo para ejecutar el proceso padre. Puesto que tiene mayor prioridad pasará a ejecutarse a continuación.
- P. padre: Ejecuta P3 y lee de la tubería vacía, por lo que se bloquea y continúa ejecutando el proceso hijo.
- P. hijo: Termina P1 y escribe en la tubería. Como no está llena, no se bloquea pero se des-bloquea al proceso padre que tiene mayor prioridad. Por lo tanto, después del tratamiento de la lectura pasa a ejecutar el proceso padre.
- P. padre: Ejecuta P4.

Siguiendo las especificaciones del enunciado se termina la traza con esta situación.

Grupo: …… NIA: …… Nombre y apellidos: ……..

Ejercicio 2

Se dispone de un computador multiprocesador con 4 CPUs. La interrupción de reloj se produce 100 veces por segundo. Se pide:

a) Diseñar un planificador de procesos incluyendo las estructuras de datos necesarias, los estados del proceso que se requieren y la descripción del algoritmo de cambio de contexto. Este planificador debe permitir lo siguiente:
   a. Poder ejecutar un proceso en cada CPU simultáneamente.
   b. No debe haber CPUs inactivas mientras haya procesos listos para ejecutar.
   c. Un proceso deben ejecutar siempre en la misma CPU salvo que otra CPU se encuentre inactiva.
   d. Los procesos deben seguir una política *round-robin* con una rodaja de 100 ms.

b) Dado la siguiente lista de procesos a ejecutar.

Proceso	Tiempo de ejecución	Instante de inicio
P1	100 ms	0 ms
P2	400 ms	0 ms
P3	400 ms	0 ms
P4	400 ms	0 ms
P5	300 ms	99 ms
P6	300 ms	99 ms
P7	300 ms	99 ms

Representar la traza de ejecución de dichos procesos que realizará el algoritmo de planificación diseñado en el apartado anterior. Utilice una tabla como la siguiente para representar dicha traza.

Tiempo	0 ms	…. ms	… ms
CPU 1	(*)		
CPU 2			
CPU 3			
CPU 4			

(*) : Proceso X, CPU Inactiva

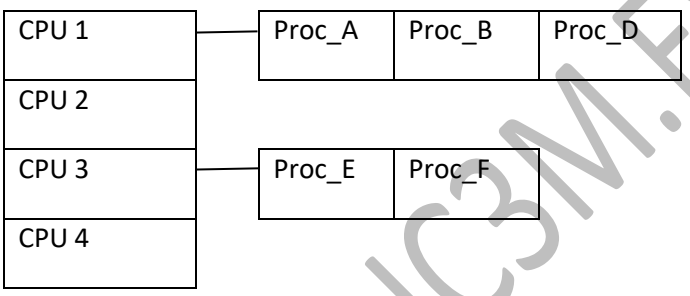
Grupo: …… NIA: …… Nombre y apellidos: ……..

SOLUCIÓN

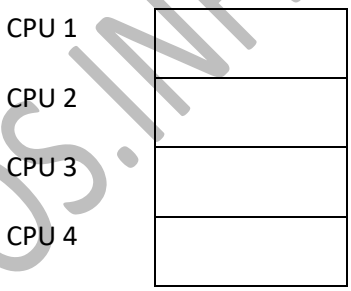
a) La interrupción de tiempo es cada 10ms, hay 10 interrupciones por rodaja.

Para el planificador pedido se precisa el diseño de las siguientes estructuras de datos:

1) Lista de procesos listos para ejecutar: consta de una lista por cada CPU



2) Lista de procesos ejecutando: es un vector de 4 posiciones (una por cada CPU)



3) Los estados del proceso: al menos: listo para ejecutar, ejecutando, bloqueado.

El cambio de contexto ocurre cuando uno de los procesos ejecutando termina o se bloquea o cuando termina su rodaja de tiempo (en ese caso se pone en la cola de listos de su CPU al final). En ese momento se coge el primer proceso de la cola de listos de esa CPU y se pone a ejecutar. Si no hay procesos en dicha cola se coge el primer proceso de otra de las colas (según el orden de CPU).

Grupo: …… NIA: …… Nombre y apellidos: ……..

b)

Tiempo	100ms	200ms	300ms	400ms	500ms	600ms
CPU 1	P1	P5	P5	P5	P6	x
CPU 2	P2	P6	P2	P6	P2	P2
CPU 3	P3	P7	P3	P7	P3	P3
CPU 4	P4	P4	P4	P4	P7	x
	(1)	(2)	(3)	(4)	(5)	

(*) u= código usuario; k=código kernel

(1) Los 1º cuatro procesos se reparten entre las CPU
(2) Aparecen 3 procesos que se reparten en las listas de listos y termina la rodaja de todos, se inicia los tres nuevos y se repite el p4.
(3) P1 termina. El resto terminan sus rodajas y se cambian salvo P5 que repite.
(4) Terminan su rodaja y se cambian salvo P5 que repite.
(5) P4 y P5 terminan y los 4 procesos que quedan se reparten entre las CPU y se repiten hasta que terminen.

Grupo: …… NIA: …… Nombre y apellidos: ……..

Ejercicio 3

Considérese que se distinguen las siguientes operaciones internas del sistema operativo:

1. Reservar y liberar una entrada de la tabla de procesos.
2. Rellenar/actualizar el BCP (se debe explicar en qué consiste la actualización).
3. Insertar y eliminar un proceso de una cola de procesos.
4. Cambiar de contexto.
5. Planificar.
6. Leer e interpretar un ejecutable.
7. Crear pila (se debe especificar cuál es su contenido inicial).
8. Crear una región de memoria (privada o compartida, asociada a un ejecutable o sin soporte).
9. Compartir y duplicar una región.
10. Eliminar una región de memoria.

Se pide especificar, basándose en las anteriores operaciones, cómo se llevan a cabo las siguientes llamadas:

a) FORK.
b) EXEC
c) EXIT y WAIT (tenga en cuenta la sincronización asociada a la terminación de procesos en UNIX).

SOLUCIÓN

a) Las operaciones básicas asociadas al FORK serían las siguientes:
   - Reservar una entrada de la tabla de procesos.
   - Rellenar el BCP reservado copiando los valores del padre (esta copia incluye información como los registros salvados, los descriptores de ficheros o el tratamiento de las señales). Sin embargo, algunos campos del BCP deben tomar valores específicos para el hijo, tales como su identificador o la información relacionada con la contabilidad sobre el uso de recursos de cada proceso. Se pondrá al proceso en estado de listo para ejecutar.
   - Con respecto a la gestión de memoria, por cada región de memoria del mapa del proceso padre:
     - Si es de carácter compartido, se comparte entre el padre y el hijo.
     - Si es de carácter privado, se duplica en el hijo, ya sea de manera inmediata o usando *copy-on-write.*

- Como resultado de las operaciones de memoria anteriormente comentadas, se ha creado implícitamente una nueva pila cuyo contenido es un duplicado de la pila del padre.
- Insertar el nuevo proceso al final de la cola de listos.

b) Las operaciones básicas asociadas al EXEC serían las siguientes:

- Leer e interpretar la cabecera del ejecutable para obtener los datos de cada región. Esta operación de lectura conlleva el bloqueo del proceso mientras que se lee del disco la cabecera del ejecutable, a no ser que el bloqueo correspondiente estuviera almacenado en la caché del sistema de ficheros. Este bloqueo implicaría a su vez las siguientes operaciones:
  - Poner al proceso en estado bloqueado.
  - Eliminar el BCP de la cola de listos e insertarlo en la cola de procesos bloqueados esperando la finalización de una operación sobre el disco.
  - Planificar y cambiar de contexto.
  - Cuando ocurra la interrupción que indica que ha terminado la operación del disco, el proceso será movido de la cola de bloqueados a la de listos y, cuando sea posteriormente elegido por el planificador, continuará procesando esta llamada EXEC.
- Eliminar las regiones del mapa actual, salvando previamente la información que corresponde con los argumentos y el entorno que recibirá el nuevo programa.
- Crear las regiones especificadas en el ejecutable de acuerdo con sus características específicas:
  - Código: Permiso de lectura y ejecución, de carácter compartido y asociado al ejecutable.
  - Datos con valor inicial: Permiso de lectura y escritura, de carácter privado y asociado al ejecutable.
  - Datos sin valor inicial: Permiso de lectura y escritura, de carácter privado y sin soporte.
- Crear pila como una región con permiso de lectura y escritura, de carácter privado y sin soporte, cuyo contenido inicial serán los argumentos y el entorno pasados al nuevo programa.
- Actualizar el BCP del proceso especificando, entre otras cosas, la nueva información sobre el mapa de memoria y los nuevos valores del contador de programa (primera instrucción del nuevo programa) y del puntero de pila (apuntando a la nueva pila).

Grupo: …… NIA: …… Nombre y apellidos: ……..

c) Las operaciones básicas asociadas al EXIT serían las siguientes:

- Eliminar las regiones del mapa actual y liberar otros recursos usados por el proceso (cerrar ficheros abiertos, liberar semáforos usados, etc.).
- Actualizar el BCP del proceso para reflejar esas operaciones y poner al proceso en estado *Zombie*. Asimismo, habría que actualizar el BCP de los procesos hijos para especificar que pasan a ser hijos directos del proceso *init*.

Las operaciones básicas asociadas al WAIT serían las siguientes:

- Si no hay ningún proceso hijo en estado *Zombie*:
  - Poner al proceso en estado bloqueado.
  - Eliminar el BCP de la cola de listos e insertarlo en una cola de espera.
  - Planificar y cambiar de contexto.
- Recoger información dejada por el proceso hijo ya terminado en su BCP (estado de terminación, contabilidad de su uso de recursos, etc.).
- Liberar la entrada de la tabla de procesos usada por el proceso hijo.

Grupo: …… NIA: …… Nombre y apellidos: ……..

Ejercicio 4

Se desea implementar un planificador basado en prioridades. Los procesos pueden tener 2 tipos distintos de prioridades:

- Prioridad Alta.
- Prioridad Baja.

Cada prioridad tiene su propia planificación:

- La política de planificación de los procesos de **prioridad alta** será **FIFO**.
- La política de planificación de los procesos de **prioridad baja** será **Round-Robin**, empleando en este caso una rodaja de tiempo de **100 milisegundos.**

Los procesos en la cola de **prioridad alta** se ejecutan en orden estricto de llegada (*FIFO*). Un proceso de **prioridad alta** se ejecuta hasta que:

a) Finaliza el proceso completamente.
b) Se duerme.
c) Se bloquea.

Un proceso de **prioridad baja** abandona el estado de ejecución cuando:

a) Finaliza su rodaja de tiempo.
b) Finaliza el proceso completamente.
c) Se duerme.
d) Se bloquea.

Se pide diseñar e indicar que funciones y estructuras de datos son necesarias para implementar el planificador.

Grupo: …… NIA: …… Nombre y apellidos: ……..

SOLUCIÓN

a) Para implementar la planificación, es necesario modificar o crear los siguientes eventos:

- Modificar la interrupción del reloj: el manejador debe activar una interrupción software que permita funcionar las rodajas de tiempo de procesos con prioridad *Round_Robin*.
- Modificar o revisar todos los eventos, que hasta ahora han utilizado la lista de listos. Por ejemplo, despertar procesos dormidos.

Además es necesario incorporar o modificar las siguientes estructuras de datos globales:

- Dividir la lista de listos actual en dos: Lista de procesos de alta prioridad y Lista de procesos de baja prioridad.
- Añadir en el BCP un campo llamado rodaja que contendrá el valor de rodaja actual.
- Añadir en el BCP un campo llamado prioridad que contendrá la prioridad.

El pseudocódigo de los eventos a implementar es el siguiente:

Pseudocódigo Manejador_interrucion_reloj()

- Ticks = Ticks +1;
- Insertar_Interrupcion_Software(Tratar_Rodaja)
- Generar_Interrupcion_Software();

Pseudocódigo Tratar_Rodaja()

- Si no (hay proceso_en_ejecucion o su prioridad es alta)
  - Terminar función
- ProcesoActual->rodaja = ProcesoActual->rodaja - 1
- Si ProcesoActual->rodaja == 0
  - ProcesoActual->rodaja= TICKS_POR RODAJA (100 milisegundos)
  - ProcesoActual->estado = LISTO
  - InsertarAlFinal (lista_procesos_listos_baja_prioridad, ProcesoActual)
  - ProcesoAnterior = ProcesoActual;
  - ProcesoActual = Planificador ()
  - Si (ProcesoActual != NULL)
    - ProcesoActual->estado = EJECUTANDO
    - Activador(ProcesoAnterior, ProcesoActual) // Cambio de contexto

Pseudocódigo Planificador ()

- Si No_vacia (lista_procesos_listo_alta_prioridad)
  - return Extraer_primer_proceso(lista_procesos_listo_alta_prioridad)
- Si no
  - return Extraer_primer_proceso(lista_procesos_listo_baja_prioridad)

Pseudocódigo llamada al sistema crear_proceso_prioridad (proceso, prioridad)

- Crear proceso (*proceso*): crear BCP, colocar estado en listo.
- Cambiar prioridad de nuevo proceso a *prioridad*.
- Añadir al nuevo proceso a la cola de listos de su prioridad

Grupo: …… NIA: …… Nombre y apellidos: ……..

Ejercicio 5

Se desea implementar un planificador *round-robin* que usa prioridades basado en una única cola de procesos listos. Para manejar esta cola únicamente se dispone las siguientes funciones:

- **BCP * devolverPrimeroListo():** Devuelve el primer elemento de la lista de listos.
- **insertarFinalListo(BCP *):** Inserta el elemento al final de la cola de procesos listos.

El planificador debe impedir la inanición de los procesos en el acceso a la CPU. Los procesos no pueden cambiar de nivel de prioridad.

Un proceso que se encuentre ejecutando en la CPU no puede ser expulsado hasta que:

a) Finaliza su rodaja de tiempo.
b) Finaliza el proceso completamente.
c) Se duerme.
d) Se bloquea.

Se pide diseñar e indicar que funciones y estructuras de datos son necesarias para implementar el planificador.

Grupo: …… NIA: …… Nombre y apellidos: ……..

SOLUCIÓN

a) Para implementar la planificación, es necesario modificar o crear los siguientes eventos:
   - Modificar la interrupción del reloj: Además del código anterior, el manejador debe activar una interrupción software que permita funcionar las rodajas de tiempo de procesos con prioridad Round_Robin.

Además es necesario incorporar o modificar las siguientes estructuras de datos globales:

- Un campo en el BCP que indique la prioridad (entero). Se considerará que un proceso A tiene más prioridad que otro B si A.prioridad > B.prioridad. El valor puede estar comprendido entre [0 .. n]
- Un campo en el BCP que indique la rodaja de tiempo (entero).
- Todos los procesos usan TICKS_POR_RODAJA para saber el tiempo mínimo que debe un proceso estar usando la CPU (entero).

Pseudocódigo Manejador_interrucion_reloj()

- Ticks = Ticks + 1;
- Insertar_Interrupcion_Software(Tratar_Rodaja)
- Generar_Interrupcion_Software();

Pseudocódigo Tratar_Rodaja()

- ProcesoActual->rodaja = ProcesoActual->rodaja - 1
- Si ProcesoActual->rodaja == 0
  - ProcesoActual->rodaja = TICKS_POR_RODAJA
  - ProcesoActual->estado = LISTO
  - **insertarAlFinal** (ListaListos, ProcesoActual).
  - ProcesoAnterior = ProcesoActual
  - ProcesoActual **= planificador()**
    NOTA: si no hay, espera hasta que lo haya y extrae el primero de la lista de listos.
  - ProcesoActual->rodaja **= (ProcesoActual->prioridad + 1) *** TICKS_POR_RODAJA
  - ProcesoActual->estado = EJECUTANDO
  - Cambiar contexto entre ProcesoAnterior y ProcesoActual // uso de activador

Grupo: …… NIA: …… Nombre y apellidos: ……..

Ejercicio 6

Se dispone de un sistema operativo con planificación FIFO el cual se quiere modificar para añadir tareas de tipos Round-Robin además de las tareas FIFO. Ambos tipos de tareas se trataran de la misma forma y con la misma prioridad.

Dos llamada al sistema permitirá elegir el tipo de planificación para cada proceso una vez creado. Su interfaz será el siguiente.

setFIFOScheduler ()
setRRScheduler ()

Se pide:

a) Diseñar e indicar qué eventos y estructuras de datos serán necesarias usar, añadir o modificar para implementar dicha funcionalidad.
b) Implementar en pseudocódigo las estructuras de datos y funciones descritas en el apartado anterior (usando interrupciones software si es necesario).

Grupo: …… NIA: …… Nombre y apellidos: ……..

SOLUCIÓN

a) Los eventos involucrados son:
- Llamadas al sistema setFIFOScheduler y setRRScheduler
- Interrupción hardware de reloj
- Interrupción software:

Las estructuras de datos:

- Tabla de procesos, Lista de listos y proceso actual.
- Se añade al BCP dos campos: rodaja_inicial, isFIFO.

b) El pseudocódigo de los eventos pedido podría ser:

Pseudocódigo Manejador_interrupción_hw_reloj()
- Ticks++
- Insertar_Interrupción_Software(Planificar_Rodaja)
- Generar_Interrupció_Software()

Pseudocódigo Planificar_Rodaja()
- Si procesoActual == NULL
  - Return
- IF (!ProcesoActual->isFIFO) {
  - (procesoActual->rodaja)--
  - Si procesoActual->rodaja == 0
    - procesoActual->estado = LISTO
    - procesoActual->rodaja = RODAJA_POR_DEFECTO
    - Insertar(ListaLISTOS, procesoActual)
    - procesoAnterior = procesoActual
    - procesoActual = planificador()
    - Borrar(ListaLISTOS, procesoActual)
    - procesoActual->estado = EJECUTANDO
    - CambioContexto(procesoAnterior, procesoActual)

Pseudocódigo SetFIFOSCheduler()
- ProcesoActual->isFIFO = TRUE;
- procesoActual->rodaja = 0;
-

Pseudocódigo SetRRSCheduler()
- ProcesoActual->isFIFO = FALSE;
- procesoActual->rodaja = RODAJA_POR_DEFECTO;

Grupo: …… NIA: …… Nombre y apellidos: ……..

Ejercicio 7

En la empresa en la que estamos trabajando disponemos de un sistema operativo básico (como el que está siendo introducido en pseudocódigo en la asignatura) que usa un *kernel* monolítico con planificación *Round-Robin*.

A dicho sistema operativo (descrito anteriormente) nuestra empresa nos pide que modifiquemos el planificador para añadir un mínimo comportamiento adaptativo que consiste en que si más de la mitad de los cambios de contexto de un proceso han sido voluntarios entonces la rodaja de tiempo que se usa en su ejecución será la mitad de la rodaja de tiempo usada por el resto de procesos.

Nos piden:

a) Diseñar e indicar qué eventos y estructuras de datos serán necesarias usar, añadir o modificar para implementar dicha funcionalidad.
b) Implementar en pseudocódigo las estructuras de datos y funciones descritas en el apartado anterior (usando interrupciones software si es necesario).
c) ¿Qué ventaja tiene el nuevo diseño que nos pide la empresa?
d) Implemente en pseudocódigo una nueva llamada al sistema que permita conocer el número de cambios de contexto voluntarios e involuntarios con la siguiente interfaz:

```
cambios ( int *voluntarios, int *no_voluntarios );
```

Grupo: …… NIA: …… Nombre y apellidos: ……..

SOLUCIÓN

a) Los eventos involucrados son:

- Llamada al sistema de crear proceso: inicializar nuevos campos del BCP.
- Interrupción hardware de reloj: sin modificar.
- Interrupción software: actualizar campos del BCP.

Las estructuras de datos:

- Tabla de procesos, Lista de listos y proceso actual.
- Se añade al BCP dos campos: cambios_totales y cambios_involuntarios

b) El pseudocódigo de los eventos pedido podría ser:

Pseudocódigo Manejador_interrupción_hw_reloj()

- Ticks++
- Insertar_Interrupción_Software(Tratar_Rodaja2)
- Generar_Interrupció_Software()

Pseudocódigo Tratar_Rodaja2()

- Si procesoActual == NULL
  - Return
- (procesoActual->rodaja)--
- Si procesoActual->rodaja == 0
  - **(procesoActual->cambios_involuntarios) ++**
  - procesoActual->estado = LISTO
  - procesoActual->rodaja = RODAJA_POR_DEFECTO
  - **Si (procesoActual->cambios_totales - procesoActual->cambios_involuntarios) > procesoActual->cambios_involuntarios**
    - **procesoActual->rodaja = 0.5 * RODAJA_POR_DEFECTO**
  - Insertar(ListaLISTOS, procesoActual)
  - procesoAnterior = procesoActual
  - procesoActual = planificador()
    // procesoActual=Primero(ListaLISTOS) + Borrar(ListaLISTOS, procesoActual)
  - procesoActual->estado = EJECUTANDO
  - CambioContexto(procesoAnterior, procesoActual) // activador

Pseudocódigo CambioContexto (procesoAnterior, procesoActual)

- **(procesoAnterior->cambios_totales) ++**
- **<Código anterior de CambioContexto>**

Grupo: …… NIA: …… Nombre y apellidos: ……..

Pseudocódigo Crear_proceso ()

- BCP->cambios_totales = 0
- BCP->cambios_involuntarios = 0
- <Código anterior de Crear_proceso()>

c) No tiene ventaja.

d) La nueva funcionalidad sería:

Pseudocódigo cambios (int * voluntarios, int * no_voluntarios)

- R0 <- código de la llamada cambios (por ejemplo 100)
- R1 <- voluntarios (dirección de memoria)
- R2 <- no_voluntarios (dirección de memoria)
- Generar Trap()
- Return R0

Pseudocódigo kernel_cambios ()

- Si procesoActual == NULL
  - R0 <- error
  - Return
- *R1 = procesoActual->cambios_totales- procesoActual->cambios_involuntarios
- *R2 = procesoActual->cambios_involuntarios
- R0 <- OK

Grupo: …… NIA: …… Nombre y apellidos: ……..

Ejercicio 8

Se quiere implementar un servicio que funcione como un buzón de mensajes para un sistema operativo básico multitarea. Sus características deben ser las siguientes:

- Cada buzón tendrá un identificador que lo represente.
- El buzón debe almacenar un número ilimitado de mensajes.
- Cualquier proceso puede guardar mensajes en el buzón.
- Cualquier proceso puede recoger mensajes del buzón.
- Los mensajes almacenados serán repartidos entre los procesos que esperan recogerlos en estricto orden de llegada (el primer mensaje que llego se entregará al primer proceso que solicito un mensaje).
- Si un proceso solicita un mensaje y no hay ninguno pendiente, entonces el proceso deberá esperar hasta que algún otro proceso envié un mensaje al buzón.
- Un proceso que envié un mensaje al buzón terminará dicha operación aunque no haya procesos esperando por dicho mensaje.

Se pide:

a) Diseñar las estructuras de datos necesarias para implementar los buzones de mensajes especificados anteriormente. Indicar que estructuras del sistema operativo que son necesarias para dicha labor y (si es necesario) qué modificaciones requieren.
b) Implementar en pseudocódigo las siguientes funciones:
    - **enviarMensaje (buzón, mensaje)**: Envía un mensaje al buzón para que sea entregado al 1º proceso que lo solicite.
    - **recogerMensaje (buzón, mensaje)**: Recoge el primer mensaje que haya en el buzón. Si no hay mensajes, el proceso esperará hasta que otro proceso envié un mensaje.

Grupo: …… NIA: …… Nombre y apellidos: ……..

SOLUCIÓN

a) Habrá una tabla en el SSOO con todos los buzones que se crean en el sistema, El identificador del buzón será la posición en dicha tabla.
La estructura de datos que componen el buzón está compuesta por dos campos:
   - Lista de mensajes almacenados por orden de llegada
   - Lista de punteros a BCP de los procesos esperando por mensajes.

b) Las funciones se implementan de la siguiente forma:
   - **enviarMensaje (buzón, mensaje):**
     - Introducir el mensaje en la lista de mensajes del buzón.
     - Si hay procesos en la lista de procesos bloqueados.
       - Obtener el primer proceso de la lista.
       - Cambiar su estado de bloqueado a listo.
       - Incluirlo en la lista de procesos listos para ejecutar.
   - **recogerMensaje (buzón, mensaje):**
     - Mientras que la lista de mensajes este vacía
       - Insertar el puntero del BCP del proceso actual en la lista de procesos del buzón.
       - Cambiar el estado del proceso actual de ejecutando a bloqueado.
       - Obtener el BCP del primer proceso de la lista de listos.
       - Poner dicho BCP en la lista de ejecutando
       - Cambiar contexto entre proceso actual y nuevo proceso ejecutando.
     - Devolver el primer mensaje de la lista.

Grupo: …… NIA: …… Nombre y apellidos: ……..

Ejercicio 9

Disponemos de un sistema con CPU monoprocesador y sistema operativo con un *kernel* monolítico expulsivo (como el que está siendo presentado en pseudocódigo en clase). En nuestra empresa se nos pide cambiar este sistema operativo para añadir una nueva llamada al sistema denominada Crear_proceso2 que permita al crear un proceso, indicar la rodaja de tiempo que tendrá durante toda su ejecución. Esta nueva llamada se unirá a la existente Crear_proceso que usará la rodaja por defecto.

Sin cambiar el planificador, se pide:

a) Indicar las estructuras de datos y funciones que se han de modificar para poder añadir una nueva llamada al sistema comentada anteriormente.

b) Indicar en pseudo-código las funciones del apartado anterior.

Grupo: …… NIA: …… Nombre y apellidos: ……..

SOLUCIÓN

a) Las estructuras de datos a modificar son: BCP
Las funciones a modificar o añadir son: la llamada Crear_proceso2 y la interrupción hw/sw de reloj.

b) El pseudocódigo de las funciones pedidas es:

Crear_Proceso2 (int rodaja)

- REG0 <- código de llamada de Syscall_Crear_proceso2
- REG1 <- rodaja
- Trap
- return //return REG0 si se devuelve 0 al hijo, pidHijo al padre y -1 si hay error

Syscall_Crear_Proceso2 ()

- Realizar lo mismo que Syscall_Crear_proceso() pero con
"BCP_procesoCreado->rodajaCompleta = REG1"
en lugar de
"BCP_procesoCreado->rodajaCompleta = TICKS_POR_RODAJA"
- Guardará en REG0 el pid del proceso hijo creado o -1 si no se ha podido crear. El hijo creado se crea colocando un 0 en el sitio para almacenar el valor de REG0 de su BCP.

Manejador_interrupción_reloj ():

- procesoActual->rodaja -- ;
- if (0 == procesoActual->rodaja)
    - procesoAnterior = procesoActual
    - procesoAnterior->estado = LISTO
    - procesoAnterior->rodaja = **procesoAnterior->rodajaCompleta**
    - Insertar(ListosParaEjecutar, procesoAnterior)
    - procesoActual = planificador()
    - procesoActual->estado = EJECUTANDO
    - cambio_contexto(procesoAnterior->context, procesoActual->context)

Grupo: …… NIA: …… Nombre y apellidos: ……..

Ejercicio 10

Una máquina monoprocesador tiene instalado un sistema operativo con núcleo no expulsivo que usa un algoritmo de planificación de procesos basado en prioridades estáticas con carácter no expulsivo.

Se quiere implementar en el *kernel* primitivas de sincronización: mutex. Un mutex es el mecanismo de sincronización de procesos ligeros más sencillo y eficiente. Los mutex se emplean para obtener acceso exclusivo a recursos compartidos y para asegurar la exclusión mutua sobre secciones críticas. Sobre un mutex se pueden realizar dos operaciones atómicas básicas:

- lock: intenta bloquear el mutex. Si el mutex ya está bloqueado por otro proceso, el proceso que realiza la operación se bloquea. En caso contrario, se bloquea el mutex sin bloquear al proceso.
  NOTA: Si un proceso ejecuta la operación lock dos veces consecutivas, la segunda debe dar error.

- unlock: desbloquea el mutex. Si existen procesos bloqueados en él, se desbloqueará a uno de ellos, que será el nuevo proceso que adquiera el mutex. La operación unlock sobre un mutex debe ejecutarla el proceso ligero que adquirió con anterioridad el mutex mediante la operación lock.
  NOTA: Solo el proceso que tiene bloqueado el mutex puede desbloquearlo. Si otro proceso ejecuta la operación unlock esta debe dar error.

La interfaz de los servicios de mutex va a ser la siguiente:

- `int crear_mutex(char *nombre)`: Crea el mutex con el nombre especificados. Devuelve un entero que representa un descriptor para acceder al mutex. En caso de error, devuelve un número negativo.
  NOTA: El mutex no se abre al crearlo, si el mismo proceso que lo crea quiere abrirlo, deberá realizar las dos llamadas (crear_mutex y abrir_mutex)
- `int abrir_mutex(char *nombre)`: Devuelve un descriptor asociado a un mutex ya existente o un número negativo en caso de error.
- `int lock(unsigned int mutexid)`: Realiza la típica labor asociada a esta primitiva, tal y como se comentó anteriormente. En caso de error, devuelve un número negativo.
- `int unlock(unsigned int mutexid)`: Realiza la típica labor asociada a esta primitiva, tal y como se comentó anteriormente. En caso de error, devuelve un número negativo.
- `int cerrar_mutex(unsigned int mutexid)`: Cierra el mutex especificado, devolviendo un número negativo en caso de error. Nótese que todas las primitivas devuelven un número negativo en caso de error.
  NOTA: El mutex no se borra al cerrarlo, únicamente se desvincula del proceso que realizó la llamada.

Se considera que el sistema sólo puede utilizar $NUM\_MUT\_TOTALES$ mutex a la vez. Si se supera este número, se producirá un error. Por simplicidad, no se permite la eliminación de mutex, es decir, una vez se crea un mutex, éste permanece en el sistema hasta la finalización de la ejecución. Un proceso podrá tener abiertos tantos mutex como existan en el sistema.

Se pide:

Grupo: …… NIA: …… Nombre y apellidos: ……..

a) Indique las estructuras de datos necesarias (incluyendo los estados de los procesos) para implementar la nueva primitiva de sincronización y el planificador de procesos requerido**.** Tenga en cuenta que el sistema debe soportar tener varias barreras abiertas al mismo tiempo.
b) Implementar en pseudocódigo los manejadores de las llamadas al sistema.

SOLUCIÓN

a)

Para implementar la funcionalidad de los mutex es necesario crear los siguientes eventos:

- Crear la llamada al sistema *crear_mutex().*
- Crear la llamada al sistema *abrir_mutex().*
- Crear la llamada al sistema *lock().*
- Crear la llamada al sistema *unlock().*
- Crear la llamada al sistema *cerrar_mutex().*

Además hay que modificar la llamada al sistema de terminar_proceso para que libere todos los mutex bloqueados por el proceso.

Es necesario incorporar las siguientes estructuras de datos globales:

- Añadir un array (Array_Mutex) de estructuras de datos mutex de tamaño *NUM_MUT_TOTALES,* de forma que cada estructura contenga:
  - Nombre del mutex.
  - Valor del mutex (0 o 1)
  - Id del proceso que posee el mutex.
  - Lista de procesos bloqueados por el mutex.
- Añadir en el BCP un array de mutex abiertos de tamaño *NUM_MUT_TOTALES* que contengan las referencias a la estructura del mutex correspondiente.

Grupo: …… NIA: …… Nombre y apellidos: ……..

b)

El pseudocódigo de los eventos a implementar es el siguiente:

Pseudocódigo llamada al sistema crear_mutex (nombre)

- Mutex = Obtener_Mutex_no_usado (Array_Mutex)
- Si no se encuentra
  - Devolver error.
- Si se encuentra
  - Mutex->nombre =n ombre;
  - Mutex->valor = 0;
  - Mutex->proc_Id = -1;
  - Inicializar (Mutex->Lista_procesos_bloqueados);

Pseudocódigo llamada al sistema abrir_mutex (nombre)

- Mutex = Encontrar_mutex(Array_Mutex, nombre)
- Si no se encuentra
  - Devolver error.
- Si se encuentra
  - Pos = Encontrar_ref_mutex_no_usada (Proceso_actual->Mutex)
  - Proceso_actual->Mutex[Pos] = Mutex
  - Devolver Pos

Pseudocódigo llamada al sistema lock (mutexid)

- Mutex = Proceso_actual->Mutex[mutexid]
- Si ((Mutex == NULL) || (Mutex->proc_Id == Proceso_actual->id))
  - Devolver error
- Si no
  - **Mientras** Mutex->valor == 1
    - Proceso_actual ->estado = bloqueado
    - InsertarProcesoActual (Mutex->lista_procesos)
    - proc = planificador() ;
    - while (proc==null) { esperarInterrupcion(); proc=planificador(); }
    - proc->estado=ejecutando;
    - cambioDeContexto(Proceso_actual ->contexto, proc->contexto) ;
- Mutex->valor = 1
- Mutex->proc_Id = Proceso_actual ->id

Grupo: …… NIA: …… Nombre y apellidos: ……..

Pseudocódigo llamada al sistema unlock (mutexid)

- Mutex= Proceso_actual->Mutex[mutexid]
- Si ((Mutex == NULL) || (Mutex->valor != 1))
    - Devolver error
- Si (Mutex->proc_Id != Proceso_actual->id)
    - Devolver error
- Si no
    - Mutex->valor = 0
    - Mutex->proc_Id = -1
    - Proc=ObtenerPrimerProceso(Mutex->lista_procesos)
    - Si existe Proc:
        - Proc->estado = listo
        - Borrar(Mutex->lista_procesos, Proc) ;
        - Insertar(lista_listos, Proc) ;

Pseudocódigo llamada al sistema cerrar_mutex (mutexid)

- Si (Proceso_actual->Mutex[mutexid] != NULL)
    - Mutex = Proceso_actual->Mutex[mutexid]
    - Si (Mutex->proc_Id == Proceso_actual->id)
        - Unlock(mutexid)
    - Proceso_actual->Mutex[mutexid] = NULL;

Pseudocódigo llamada al sistema terminar_proceso ()

- for (i=0; i < NUM_MUT_TOTALES; i++)
    - cerrar_mutex(i) ;
- <RESTO DE CÓDIGO para terminar_proceso>

Grupo: …… NIA: …… Nombre y apellidos: ……..

Ejercicio 11

Una máquina monoprocesador tiene instalado un sistema operativo con núcleo no expulsivo que usa un algoritmo de planificación de procesos basado en prioridades estáticas con carácter no expulsivo.

Se quiere implementar en el *kernel* una nueva primitiva de sincronización: la barrera. Una barrera es utilizada en computación paralela para asegurar que todos los procesos completen una fase de ejecución antes de pasar a la siguiente, el funcionamiento es el siguiente:

- Dada una barrera de tamaño N, todos los procesos que llamen a la barrera se bloquearán hasta que los N procesos realicen la llamada.
- Cuando todos los procesos hayan realizado la llamada continuarán ejecutándose.

Se pide:

a) Indique las estructuras de datos necesarias (incluyendo los estados de los procesos) para implementar la nueva primitiva de sincronización y el planificador de procesos requerido**.** Tenga en cuenta que el sistema debe soportar tener varias barreras abiertas al mismo tiempo.
b) Implementar en pseudocódigo los manejadores de los siguientes eventos:
   - Llamada al sistema `int crearbarrera (int n)`: Crea una barrera de tamaño n y devuelve el identificador único de la barrera creada.
   - Llamada al sistema `int liberarbarrera (int idBarrera)`: Destruye la barrera idBarrera y despierta a algún proceso que pudiera estar esperando para que continúe su ejecución. Devuelve el número de procesos notificados.
   - Llamada al sistema `void barrera (int idBarrera)`: Esta operación la invoca un proceso al llegar a la barrera. Tiene el efecto de bloquear un proceso hasta que los N procesos (n es el tamaño de la barrera) la hayan llamado, momento en el cuál se despertarán a todos los procesos que estuvieran bloqueados para que sigan su ejecución.

Grupo: …… NIA: …… Nombre y apellidos: ……..

SOLUCIÓN

a) Estructuras de datos involucradas:
- Campo en el BCP prioridad que será un entero, mientras menor sea este número mayor será la prioridad del proceso.
- Lista de procesos listos: cada entrada contiene un puntero a un BCP de los procesos listos para ejecución, está ordenada de mayor a menor prioridad.
- Estructura de datos tipo barrera, que contiene:
  - tamaño: tamaño de la barrera
  - bloqueados: Lista de procesos bloqueados en la barrera (cada entrada un puntero a un BCP)
- Lista de barreras:
  - Contendrá una estructura de datos tipo barrera por cada barrera abierta en el sistema
  - La posición se corresponderá con el identificador de la barrera
- Los estados de los procesos son al menos: listo, ejecutando, bloqueado.

b) Manejadores de eventos:
Llamada_al_Sistema_Crear_Barrera (int n):
- pos = BuscarPosicionLibre (listaBarreras);
- listaBarreras[pos].tamaño = n;
- listaBarreras[pos].bloqueados = NULO;
- devolver(pos);

Despertar_Bloqueados_Barrera (listaBloqueados):
- Proc = ObtenerPrimerProceso (lista_bloqueados);
- Mientras (Proc != NULO)
  - Cambiar su estado de bloqueado a listo.
  - Eliminar Proc de lista_bloqueados
  - Incluirlos en la lista de procesos listos para ejecutar por orden de prioridad.
  - Proc = ObtenerPrimerProceso (lista_bloqueados).

Llamada_al_Sistema_Liberar_Barrera (int barreraId):
- /* Si hay algún proceso bloqueado en la barrera lo debemos despertar */
- listaBloqueadosBarrera = listaBarreras[barreraId].bloqueados;
- numProcesosBloqueados= ContarElementos(listaBloqueadosBarrera);
- Si (numProcesosBloqueados>0)
  - Despertar_Bloqueados_Barrera(listaBloqueadosBarrera);
- LiberarElemento(listaBarreras[barreraId]);
- Devolver(numProcesosBloqueados);

Grupo: …… NIA: …… Nombre y apellidos: ……..

Nota: la llamada a Despertar_Bloqueados_Barrera se realiza como una función normal ya que si lo hiciéramos incluyendo una interrupción software podría liberarse la barrera sin que se hayan despertado los procesos bloqueados en la barrera, dada la baja prioridad de las interrupciones software.

Llamada_al_Sistema_Barrera (int barreraId):

- /* ¿Somos el último proceso que ha alcanzado la barrera? */
- barrera = listaBarreras[barreraId];
- SI (ContarElementos(barrera.bloqueados) == barrera.tamaño – 1)
    - Insertar_Interrupcion_Software(Despertar_Bloqueados_Barrera, listaBarreras[barreraId].bloqueados);
    - Generar_InterrupcionSoftware();
- SI_NO
    - InsertarOrdenPrioridadProcesoActual (barrera.bloqueados);
    - Cambiar el estado del proceso actual de ejecutando a bloqueado.
    - Obtener el BCP del primer proceso de la lista de listos.
    - Poner dicho BCP en la lista de ejecutando.
    - Cambiar el contexto entre el proceso actual y nuevo proceso ejecutando
      /* Aquí se bloquea el proceso*/
- FIN_SI

Grupo: …… NIA: …… Nombre y apellidos: ……..

Ejercicio 12

Se dispone de un sistema hardware que incluye un dispositivo de reloj, el cual genera una interrupción con cada *tick* del reloj.

Un sistema operativo básico (como el que está siendo presentado en pseudocódigo en clase) que es multitarea tiene prevista (a falta de la implementación) la llamada al sistema:

```
Dormir (int segundos);
```

Que permite dormir un proceso el tiempo en segundos determinado.

Se pide:

a) Implementar en pseudocódigo la funcionalidad necesaria del kernel del sistema operativo.

Grupo: …… NIA: …… Nombre y apellidos: ……..

SOLUCIÓN

a) Los eventos involucrados son:
   - Interrupción del reloj
   - Llamada al sistema ObtenerFecha
   - Llamada al sistema Dormir

   Después las funciones a ejecutar en cada evento.

   Manejador_interrupción_reloj ():

   - Ticks = Ticks +1;
   - Insertar_Interrupcion_Software(Despertar_dormidos)
   - Generar_Interrupcion_Software();

   Despertar_dormidos ():
   - Proc = ObtenerPrimerProceso (lista_procesos_dormidos);
   - Mientras (Proc.DespertarEnTick < Ticks)
     - Cambiar su estado de bloqueado a listo.
     - Incluirlo en la lista de procesos listos para ejecutar.
     - Proc = ObtenerPrimerProceso (lista_procesos_dormidos);

   LLamada_al_Sistema_Dormir (Segundos):

   - Proceso_actual.DespertarEnTick = Ticks + SegundosATicks(Segundos);
   - Insertar el puntero del BCP del proceso actual en la lista de procesos dormidos, ordenadamente de menor a mayor valor de *DespertarEnTick*.
   - Cambiar el estado del proceso actual de ejecutando a bloqueado.
   - Obtener el BCP del primer proceso de la lista de listos.
   - Poner dicho BCP en la lista de ejecutando.
   - Cambiar contexto entre proceso actual y nuevo proceso ejecutando.

   Las Variables globales necesarias son:

   - Ticks: Números de ticks de reloj desde el arranque de la máquina.
   - Lista de procesos dormidos (cada entrada un puntero a un BCP)
   - Campo en el BCP *DespertarEnTick* que indica el valor absoluto de Ticks en el que debe despertar.

Grupo: …… NIA: …… Nombre y apellidos: ……..

Ejercicio 13

Una máquina monoprocesador tiene instalado un sistema operativo básico (como el que está siendo presentado en pseudocódigo en clase) con núcleo no expulsivo que usa un algoritmo de planificación de procesos basado en prioridades estáticas con carácter no expulsivo.

Se quiere implementar en el kernel dos nuevas primitivas:

```
int dormir2 (unsigned int segundos, pid_t pid);
int despertar (pid_t pid);
```

La primera primitiva permite a un proceso dormir tantos segundos como se indique y despierta a un proceso indicado por el parámetro pid, en caso de que este esté dormido.

La segunda primitiva despierta a un proceso que se encuentre dormido cuyo PID se indica por parámetro.

Se pide:

a) Indique las estructuras de datos necesarias (incluyendo los estados de los procesos) para implementar la nueva primitiva, así como las llamadas internas para poder volver al estado original. Tenga en cuenta que el sistema debe soportar tener varios procesos durmiendo al mismo tiempo.

b) Implementar en pseudocódigo las siguientes llamadas al sistema, así como las llamadas auxiliares:
   - Llamada al sistema `int dormir2 (int n, pid_t pid)`
   - Llamada al sistema `int despertar (pid_t pid)`

SOLUCIÓN

a) Estructuras de datos involucradas:
- Campo en el BCP pid.
- Campo en el BCP Despertar_en_ticks.
- Lista de procesos listos.
- Lista de procesos dormidos: cada entrada contiene un puntero a un BCP de los procesos dormidos, ordenada de menor a mayor número de Despertar_en_ticks.
- La interrupción de reloj
- Los estados de los procesos son al menos: listo, ejecutando, bloqueado y dormido.

Grupo: …… NIA: …… Nombre y apellidos: ……..

b)

Pseudocódigo llamada al sistema dormir2 (unsigned int segundos, pid_t pid)

- Proceso_buscado = BuscarProceso(Pid)
- Si (Proceso_buscado->estado == dormido)
    - Proceso_buscado->estado = listo
    - BorrarDeLista(lista_procesos_dormidos, Proceso_buscado)
    - InsertarLista(lista_procesos_listos, Proceso_buscado)
- Proceso_actual->Despertar_en_ticks = Tick + SegundosPorTicks*segundos
- Proceso_actual->estado = dormido
- InsertarLista(lista_procesos_dormidos, proceso_actual)
- Proceso_nuevo = planificador() ;
- while (Proceso_nuevo ==null) { esperarInterrupcion();Proceso_nuevo =planificador(); }
- Proceso_nuevo ->estado = ejecutando;
- cambioDeContexto(Proceso_actual->contexto, Proceso_nuevo->contexto) ;

Pseudocódigo Manejador_interrucion_reloj()

- Ticks=Ticks +1
- Insertar_Interrupcion_Software(Despertar_dormidos)
- Generar_Interrupcion_Software()

Pseudocódigo Despertar_dormidos()

- Proc=ObtenerPrimerProceso(lista_procesos_dormidos)
- Mientras(Proc.DespertarEnTicks < Ticks)
    - Cambiar el estado de Proc de bloqueado a listo
    - Mover Proc de la lista de dormidos a la de listos para ejecutar
    - Proc=ObtenerPrimerProceso(lista_procesos_dormidos)

Pseudocódigo llamada al sistema despertar (pid_t pid)

- Proceso_buscado = BuscarProceso(Pid)
- Si (Proceso_buscado.estado == dormido)
    - Proceso_buscado.estado = listo
    - BorrarDeLista(dormidos,Proceso_buscado)
    - InsertarLista(lista_procesos_listos,Proceso_buscado)

Grupo: …… NIA: …… Nombre y apellidos: ……..

Ejercicio 14

Se dispone de un sistema hardware monoprocesador que incluye un dispositivo de reloj, el cual genera una interrupción con cada *tick* del reloj.

En dicho sistema hardware se ejecuta un sistema operativo básico expulsivo con planificación *Round-Robin* (similar al que se va describiendo en pseudocódigo a lo largo de la teoría). Los procesos pueden bloquearse por acceso a un dispositivo de E/S que contiene una sola lista de espera, por tanto los procesos pueden estar en ejecutando, en la cola de listos o en la de bloqueados

Los administradores de la máquina nos proponen que cambiemos el código para que poder conocer los siguientes datos:

- El tiempo (en segundos) que un proceso lleva desde su inicio de ejecución mediante la ejecución de la siguiente llamada al sistema:

  int tiempoEjecutando (int id_proceso) ;

- El tiempo (en segundos) que un proceso ha estado en CPU ejecutando, mediante la llamada al sistema:

  int tiempoCPU (int id_proceso) ;

Se pide para ambos casos indicar:

a) Las estructuras de datos requeridas o modificadas,
b) La interfaz (e implementación) de las funciones desarrolladas y los eventos utilizados. Implementando en pseudocódigo la funcionalidad propuesta.

NOTAS: la constante TICKS_PER_SECONDS indica el número de ticks de reloj en un segundo. Considere que no hay que hacer una interrupción software para el reloj y que siempre existe un proceso idle o nulo.

Grupo: …… NIA: …… Nombre y apellidos: ……..

SOLUCIÓN

a) Se necesitan tener los siguientes datos a nivel global:

Variables globales existentes:

- **ticks**: Ticks desde el arranque del S.O.
- **procesoActual:** Variable que apunta al BCP del proceso que está actualmente ejecutando en la CPU o a NULL sino hay ninguno disponible

Nuevos campos en el BCP:

- **ticksEnCPU**: se guarda los ticks que el proceso ha pasado en CPU ejecutando
- **ticksInicio:** campo que guarda el instante en ticks en el que se fue creado el proceso

b) Los eventos involucrados son:

- Interrupción del reloj
- Llamada al sistema **int ticksCPUOcupada ()**
- Llamada al sistema **int ticksEjecutandoProcesoActual ()**
- Llamada al sistema **int crearTarea()**

Las funciones a ejecutar en cada evento:

interrupción_hw_reloj (void)

- ticks++;
- if (procesoActual != NULL)
    - procesoActual.ticksEnCPU++ ;

int crearTarea ()

- ticksInicio = ticks ;
- return crearProcesoOriginal() ; // se llama a la rutina de creación de proceso.

int ticksCPUOcupada (int pid)

- return (tablaProcesos[pid].ticksEnCPU / TICKS_PER_SECONDS) ;

int ticksEjecutandoProcesoActual (int pid)

- return (ticks - tablaProcesos[pid].ticksInicio / TICKS_PER_SECONDS) ;

Sería necesaria también la función de la biblioteca que a nivel de usuario empaqueta en los registros los parámetros y realizar el 'trap' para invocar a las llamadas al sistema.

También hay que añadir el puntero a la función **ticksCPUOcupada** a la tabla de llamadas al sistema y lo mismo para **ticksEjecutandoProcesoActual**

Grupo: …… NIA: …… Nombre y apellidos: ……..

Ejercicio 15

Se dispone de un sistema hardware que incluye un dispositivo de reloj, el cual genera una interrupción con cada *tick* del reloj.

Un sistema operativo básico (similar al que se va describiendo en pseudocódigo a lo largo de la teoría) tiene prevista (a falta de la implementación) la llamada al sistema:

```
int dormir2 (int como_despertar, int durmiendo);
```

Que permite dormir un proceso y especificar en qué lugar de la cola de listos se debe despertar, existen dos opciones: **principio** de la cola de listos o **final** de la cola de listos. En el caso de quererse despertar al principio, el tiempo que se pide que duerma deberá ser superior a 10 segundos. Si no es así, se introducirá al final de la cola de listos.

Se pide Implementar en pseudocódigo la funcionalidad necesaria del kernel del SSOO para ofrecer el servicio descrito. Indicar también:

- Las estructuras de datos requeridas o modificadas,
- La interfaz de las funciones implementadas y
- Los eventos utilizados.

Grupo: …… NIA: …… Nombre y apellidos: ……..

SOLUCIÓN

Los eventos involucrados son:

- Interrupción del reloj
- Llamada al sistema dormir2

Las variables necesarias son:

- Ticks: números de ticks de reloj desde el arranque de la máquina.
- Lista de procesos dormidos: cada entrada un puntero a un BCP
- Dos campos nuevos en el BCP: *ComoDespertar* que indica si se va a insertar al principio de la cola de listos o al final y *DespertarEnTicks* que indica el valor absoluto de Ticks en el que debe despertar.

Las funciones a ejecutar en cada evento:

Interrupción_hw_reloj (void)

- Ticks = Ticks +1;

- Proc = ObtenerPrimerProceso (lista_procesos_dormidos);
- Mientras (Proc.DespertarEnTicks < Ticks)
    - Cambiar su estado de dormido a listo.
    - Si (Proc.ComoDespertar == “principio”)
        - Insertar en lista de listos al principio.
    - En caso contrario
        - Insertar al final de listos.
    - Proc = ObtenerPrimerProceso (lista_procesos_dormidos);

int dormir2 (int comodespertar, int durmiendo)

- Proceso_actual.ComoDespertar = comodespertar;
- Proceso_actual.DespertarEnTicks = Ticks + SegundosToTicks(durmiendo);
- Si (durmiendo < 10)
    - Proceso_actual.ComoDespertar = “final”

- Insertar ordenadamente por campo DespertarEnTicks en lista_procesos_dormidos
- Cambiar estado a dormido en el BCP
- Pedir otro proceso al planificador
- Cambio de contexto.

Grupo: …… NIA: …… Nombre y apellidos: ……..

Ejercicio 16

Se dispone de un sistema hardware monoprocesador que incluye un dispositivo de reloj, el cual genera una interrupción con cada *tick* del reloj.

En dicho sistema hardware se ejecuta un sistema operativo básico expulsivo con planificación *Round-Robin* (similar al que se va describiendo en pseudocódigo a lo largo de la teoría) con una rodaja de 10 ticks. Los procesos pueden estar en ejecutando o en la cola de listos.

Los administradores de la máquina nos proponen que cambiemos el código para implementar una llamada que impida que un proceso pueda ser expulsado de la CPU cuando se termina su rodaja mediante la llamada al sistema:

noEsExpulsable ()

También se debe proporcionar la llamada al sistema

esExpulsable()

Que permite que el proceso pueda ser expulsable cuando finalice su rodaja

Durante el tiempo en el que el proceso está en estado no expulsable la rodaja de tiempo debe seguir contabilizándose

Si se puede expulsar si se bloquea y si se atienden interrupciones

Indicar:

a) Las estructuras de datos requeridas o modificadas,
b) La interfaz (e implementación) de las funciones desarrolladas y los eventos utilizados. Implementando en pseudocódigo la funcionalidad propuesta.

NOTAS: la constante TICKS_PER_SECONDS indica el número de ticks de reloj en un segundo. Considere que no hay que hacer una interrupción software para el reloj y que siempre existe un proceso idle o nulo.

Grupo: …… NIA: …… Nombre y apellidos: ……..

SOLUCIÓN

a) Se necesitan tener los siguientes datos a nivel global:

Variables globales existentes:

- **ticks**: Ticks desde el arranque del S.O.
- **tamañoRodaja**: ticks de la rodaja, por defecto, de cada proceso. Inicializada a 10
- **procesoActual:** Variable que apunta al BCP del proceso que está actualmente ejecutando en la CPU o a NULL sino hay ninguno disponible

Nuevos campos en el BCP:

- **expulsable** : se guarda con true si el proceso es expulsable y con false sino lo es

b) Los eventos involucrados son:

- Interrupción del reloj
- Llamada al sistema **int esExpulsable ()**
- Llamada al sistema **int noEsExpulsable ()**

Las funciones a ejecutar en cada evento:

interrupción_hw_reloj (void)

- ticks++;
- if (procesoActual != NULL)
  - procesoActual.rodajaActual++
  - if (procesoActual.expulsable)
    - if (procesoActual.rodajaActual >= tamañoRodaja)
      - procesoActual.rodajaActual=0
      - Cambiar el estado de procesoActual de ejecutando a listo.
      - Insertar procesoActual al final de lista_procesos_listos.
      - Proc=Obtener_primer_proceso(lista_procesos_listos)
      - Cambiar el estado de Proc a ejecutando
      - Proceso_en_ejecucion=Proc
      - Cambiar de contexto entre procesoActual y nuevo proceso ejecutando

int crearProceso ()

- procesoActual.rodajaActual= 0
- procesoActual.expulsable=false
- return crearProcesoOriginal(); // se llama a la rutina de creación de proceso.

Grupo: …… NIA: …… Nombre y apellidos: ……..

int esExpulsable (int n)

- procesoActual.expulsable=false

int noEsExpulsable (int n)

- procesoActual.expulsable=true

Sería necesaria también la función de la biblioteca que a nivel de usuario empaqueta en los registros los parámetros y realizar el 'trap' para invocar a las llamadas al sistema. También hay que añadir el puntero a la función **rodajaProceso** a la tabla de llamadas al sistema

Grupo: …… NIA: …… Nombre y apellidos: ……..

Ejercicio 17

Se dispone de un sistema hardware monoprocesador que incluye un dispositivo de reloj, el cual genera una interrupción con cada *tick* del reloj.

En dicho sistema hardware se ejecuta un sistema operativo básico expulsivo con planificación *Round-Robin* (similar al que se va describiendo en pseudocódigo a lo largo de la teoría). Los procesos pueden estar en ejecutando o en la cola de listos.

Los administradores de la máquina nos proponen que cambiemos el código para implementar un esbozo de prioridades de la siguiente forma. Se debe proporcionar la llamada al sistema:

aumentarRodaja (int n)

Donde n es un valor entre 1 y 10 que multiplica el valor de la rodaja.

Se supondrá que la rodaja por defecto es de 10 ticks. Si invoco a *aumentarRodaja* con un 3 como parámetro el proceso pasará a tener una rodaja de 30.

La rodaja no puede ser mayor de 100 y se puede aumentar varias veces.

Indicar:

a) Las estructuras de datos requeridas o modificadas,
b) La interfaz (e implementación) de las funciones desarrolladas y los eventos utilizados. Implementando en pseudocódigo la funcionalidad propuesta.

NOTAS: la constante TICKS_PER_SECONDS indica el número de ticks de reloj en un segundo. Considere que no hay que hacer una interrupción software para el reloj y que siempre existe un proceso idle o nulo.

Grupo: …… NIA: …… Nombre y apellidos: ……..

SOLUCIÓN

a) Se necesitan tener los siguientes datos a nivel global:

Variables globales existentes:

- **ticks**: Ticks desde el arranque del S.O.
- **tamañoRodaja**: ticks de la rodaja, por defecto, de cada proceso. Inicializada a 10
- **procesoActual:** Variable que apunta al BCP del proceso que está actualmente ejecutando en la CPU o a NULL sino hay ninguno disponible

Nuevos campos en el BCP:

- **rodajaActual**: se guarda los ticks que el proceso ha pasado en CPU ejecutando en la actual ejecución
- **rodajaProceso:** campo que guarda el número de ticks que tiene este proceso como rodaja

b) Los eventos involucrados son:

- Interrupción del reloj
- Llamada al sistema **int aumentarRodaja ()**

Las funciones a ejecutar en cada evento:

interrupción_hw_reloj (void)

- ticks++;
- if (procesoActual != NULL)
  - procesoActual->rodajaActual++
  - if (procesoActual->rodajaActual >= rodajaProceso)
    - procesoActual->rodajaActual = 0
    - Cambiar el estado de procesoActual de ejecutando a listo.
    - Insertar procesoActual al final de lista_procesos_listos.
    - Proc=Obtener_primer_proceso(lista_procesos_listos)
    - Cambiar el estado de Proc a ejecutando
    - Proceso_en_ejecucion=Proc
    - Cambiar de contexto entre procesoActual y nuevo proceso ejecutando

int crearProceso ()

- procesoActual->rodajaActual = 0
- procesoActual->rodajaProceso = 10
- return crearProcesoOriginal(); // se llama a la rutina de creación de proceso.

Grupo: …… NIA: …… Nombre y apellidos: ……..

void aumentarRodaja (int n)

- if (n>0 && n<=10)
  - procesoActual->rodajaProceso = procesoActual->rodajaProceso*n
- if (procesoActual->rodajaProceso > 100)
  - procesoActual->rodajaProceso = 100

Sería necesaria también la función de la biblioteca que a nivel de usuario empaqueta en los registros los parámetros y realizar el 'trap' para invocar a las llamadas al sistema.

También hay que añadir el puntero a la función **rodajaProceso** a la tabla de llamadas al sistema

Grupo: …… NIA: …… Nombre y apellidos: ……..

Ejercicio 18

En la empresa en la que estamos trabajando disponemos de un sistema operativo básico (como el que está siendo introducido en pseudocódigo en la asignatura) que usa un *kernel* monolítico con planificación *Round-Robin* con una rodaja de tiempo de 100ms y 3 niveles de prioridad (0 para el más alto y 2 para el más bajo)

Nuestra empresa nos pide que modifiquemos el planificador del sistema operativo indicado en el párrafo anterior para añadir un mínimo comportamiento adaptativo que consiste en que si un proceso lleva más de 10000ms en una cola de listos se le aumente en uno su prioridad. Una vez que el proceso ejecute se le devolverá a la cola de su prioridad inicial. Es decir si un proceso de prioridad 2 no ejecuta durante 10000ms pasará a prioridad 1, si estando en ese nivel pasan otros 10000ms y sigue sin ejecutar se le pasará a prioridad 0. Cuando ejecute volverá a prioridad 2.

Nos piden:

a) Diseñar e indicar qué eventos y estructuras de datos serán necesarias usar, añadir o modificar para implementar dicha funcionalidad.
b) Implementar en pseudocódigo las estructuras de datos y funciones descritas en el apartado anterior (usando interrupciones software si es necesario).
c) ¿Qué ventaja tiene el nuevo diseño que nos pide la empresa?

Grupo: …… NIA: …… Nombre y apellidos: ……..

SOLUCIÓN

a) Los eventos involucrados son:
   - Llamada al sistema de crear proceso: inicializar nuevos campos del BCP.
   - Interrupción hardware de reloj: colocar interrupción sw de asociada a la int. de reloj.
   - Interrupción software: actualizar campos del BCP y revisar la lista de procesos de las prioridades inferiores para proceder a los aumentos de prioridad.
   - Modificar planificador para considerar la utilización de 3 colas de procesos listos

   Las estructuras de datos:

   - Tabla de procesos, 3 colas de procesos listos (**ColaListos[3] (valores de 0 a 2))** y variable procesoActual con el proceso actual (BCP).
   - Se añade al BCP los campos: prioridad inicial, prioridad actual, tiempo (ticks) de la última vez que ejecutó (**ticksUltimaEjecución)**

   Llamada al sistema:

   - Crear_Proceso (int prioridad)

b) El pseudocódigo de los eventos pedido podría ser:
   RODAJA_POR_DEFECTO=pasarmsaticks(100)

   Pseudocódigo Manejador_interrupción_hw_reloj()
   - Ticks++
   - Insertar_Interrupción_Software(Recolocar_Colas)
   - Generar_Interrupció_Software()

Pseudocódigo Recolocar_Colas ()

- Si procesoActual == NULL //No debería producirse si existe el proceso idle.
  - Return
- (procesoActual->rodaja)—
- Si procesoActual->rodaja == 0
  - //Comprobamos si hay que aumentar la prioridad a los procesos sólo cuando se acaba la rodaja del proceso en ejecución en lugar de en cada tick de reloj para ahorrar tiempo. Sólo quitamos un proceso de ejecución cuando se acaba su rodaja, no cuando llega uno de mayor prioridad. No sería razonable que un proceso que lleva esperando ejecutar 20000ms sólo ejecutara durante 1 tick.
  - **prio=1;** //Los de prioridad 0 no pueden aumentar la prioridad
  - **while (prio<=2)**
    - **Proc=Primer_Proceso(ColaListos[prio])**
    - //Por la forma de insertar los procesos, éstos están ordenados por ticksUltimaEjecución
    - **while (Proc!=null) && (Proc.ticksUltimaEjecución+pasarmsaticks(10000)>Ticks)**
      - **Sacar_Cola_Listos (proc, ColaListos[prio])**
      - **Meter_Cola_Listos (proc, ColaListos[prio-1])**
      - **Proc.ticksUltimaEjecución =Ticks;**
      - **Proc.prioridadActual=prio-1;**
      - **Proc=Primer_Proceso(ColaListos[prio])**
    - **prio++**
  - procesoActual->estado = LISTO
  - procesoActual->rodaja = RODAJA_POR_DEFECTO
  - **Insertar(procesoActual, ColaListos[procesoActual.prioridadOriginal])**
  - **procesoActual.prioridadActual=procesoActual.prioridadOriginal**
  - **Proc. ticksUltimaEjecución =Ticks;**
  - procesoAnterior = procesoActual
  - procesoActual = planificador()
  - if (procesoActual!= null)
    - Borrar(ColaListos, procesoActual)
    - procesoActual->estado = EJECUTANDO
    - CambioContexto(procesoAnterior, procesoActual)

Pseudocódigo planificador() //podríamos suponer que siempre existe el proceso idle

- Además de la funcionalidad ya existente habría que hacer algo como:
- prio=0
- **Sacar_Cola_Listos (proc, ColaListos[prio])**
- **While (prio<=2 && proc== null)**
  - **Prio++;**
  - **Sacar_Cola_Listos (proc, ColaListos[prio]**
- **Return proc**

Grupo: …… NIA: …… Nombre y apellidos: ……..

Pseudocódigo Crear_Proceso (int prio)

- Además de la funcionalidad ya existente de añadir datos al BCP, añadir al proceso a la cola de listos que le corresponda, colocar estado, etc.
- If (prio <0 || prio >2) prio=2;
- **procesoCreado.prioridadOriginal=prio;**
- **procesoCreado.prioridadActual=prio;**
- **procesoCreado.ticksUltimaEjecución =Ticks**
- añadirAlFinal(procesoCreado, ColaListos[prio])

c) Se evita la hambruna de los procesos de baja prioridad.

Grupo: …… NIA: …… Nombre y apellidos: ……..

Ejercicio 19

En la empresa en la que estamos trabajando disponemos de un sistema operativo básico (como el que está siendo introducido en pseudocódigo en la asignatura) que usa un *kernel* monolítico con planificación con dos niveles de prioridad, usándose *Round-Robin* para los proceso de baja prioridad y FIFO para los procesos de prioridad alta.

Se pide indicar el diseño e implementación en pseudocódigo de los cambios que hay que hacer en el sistema operativo inicial para añadir la siguiente funcionalidad:

a) Añadir la llamada al sistema:
   void setpriorirty (int newpriority, int pid);
   Que cambia en el acto la prioridad del proceso cuyo identificador se indica en pid (índice en la tabla de procesos) y en el caso de pasar de prioridad alta a baja se encola el proceso en la lista de listos correspondiente y se pasa a ejecutar el que corresponda ser el siguiente.

b) Disminuir el efecto de la inanición de la siguiente forma: cuando un proceso de prioridad alta lleva ejecutando más de tres rodajas de reloj de un proceso de prioridad baja, dicho proceso pasa al final de la lista de prioridad baja, y se pasa a ejecutar el proceso que le correspondiera a ser el siguiente.
   NOTA: Utilice una única interrupción software.

c) ¿Qué efecto tiene esta propuesta contra la inanición si todos los procesos de prioridad alta duran más de tres rodajas?

Grupo: …… NIA: …… Nombre y apellidos: ……..

SOLUCIÓN

a) El pseudocódigo de los eventos pedido podría ser:

Llamada al sistema en espacio de usuario setpriority(int newpriority, int pid)

- R0 = SETPRIORITY
- R1 = newpriority
- R2 = pid
- TRAP
- Return R0

Llamada al sistema en kernel setpriority()

- oldpriority = tablaProcesos[i].priority
- tablaProcesos[i].priority = R1
- Si (oldpriority == ALTA) && (R1 == BAJA) &&
  (procesoActual != &(tablaProcesos[i])) // está en la lista de Alta
  - eliminar(listaListosAlta, &(tablaProcesos[i]))
  - insertar(**listaListosBaja**, &(tablaProcesos[i]));
- Si (oldpriority == ALTA) && (R1 == BAJA) &&
  (procesoActual == &(tablaProcesos[i])) // es el proceso actual
  - procesoActual->estado = LISTO
  - insertar(**listaListosBaja**, procesoActual);
  - procesoAnterior = procesoActual;
  - procesoActual = planificador()
  - procesoActual->estado = EJECUTANDO
  - activador(procesoAnterior, procesoActual)

b) La nueva funcionalidad sería:

Pseudocódigo Interrupción hardware de reloj ()

- Ticks ++
- InsertarTareaPendiente(comprobarRodajaPlus)
- ActivarInterrupciónSoftware()

Grupo: …… NIA: …… Nombre y apellidos: ……..

Pseudocódigo comprobarRodajaPlus()

- Si (procesoActual == NULL)
  - Return
- (procesoActual->rodaja)--
- Si procesoActual->rodaja == 0
  - procesoActual->estado = LISTO
  - Si (procesoActual->prioridad == ALTA)
    - procesoActual->rodaja = 3 * RODAJA_POR_DEFECTO
  - en otro caso
    - procesoActual->rodaja = RODAJA_POR_DEFECTO
  - Insertar(**ListaListosBaja**, procesoActual)
  - procesoAnterior = procesoActual
  - procesoActual = planificador()
  - procesoActual->estado = EJECUTANDO
  - activador(procesoAnterior, procesoActual)

Pseudocódigo Llamada a crear un proceso()

- <Igual que antes>
- Si proceso es de prioridad alta
  - nuevoProceso->rodaja = 3 * RODAJA_POR_DEFECTO
  - Insertar(ListaListosAlta, nuevoProceso)
- Si proceso es de prioridad baja
  - nuevoProceso->rodaja = RODAJA_POR_DEFECTO
  - Insertar(ListaListosBaja, nuevoProceso)

c) El efecto es el de tener *Round-Robin* para todos los procesos donde la duración de la rodaja es el triple para los procesos de prioridad alta.
Se evita inanición (en algún momento entran todos los proceso a ejecutar).

Grupo: …… NIA: …… Nombre y apellidos: ……..

Ejercicio 20

En la empresa en la que estamos trabajando disponemos de un sistema operativo básico (como el que está siendo introducido en pseudocódigo en la asignatura) que usa un *kernel* monolítico con planificación con dos niveles de prioridad, usándose *Round-Robin* para los proceso de baja prioridad y FIFO para los de prioridad alta.

Se desea añadir una llamada al sistema llamada "turbo" que permita ejecutar el proceso con prioridad alta durante un periodo de tiempo, pasado el cuál si el proceso era de prioridad baja volverá a la prioridad baja y se cambiará a ejecutar el que toque siguiente. Si es de prioridad alta se ignorará dicha llamada. Esta llamada tiene un único parámetro que es la duración del proceso.

Se pide:

a) Indique las estructuras de datos necesarias, modificaciones en el BCP e implementación en pseudocódigo de la llamada descrita únicamente.

b) Modifique el manejador de interrupción de reloj para que se pueda realizar el cambio de prioridad en caso de que sea necesario. Incluir la interrupción Software en el caso de que sea necesario.

c) Conteste razonando brevemente a las siguientes preguntas:
   1. ¿Evitaría la inanición esta propuesta?
   2. ¿Qué ventajas e inconvenientes encuentra respecto a la solución inicial?

Grupo: …… NIA: …… Nombre y apellidos: ……..

SOLUCIÓN

a) El pseudocódigo de los eventos pedido podría ser:

Llamada al sistema en espacio de usuario turbo (int duracion)

- R0 = TURBO
- R1 = duración
- TRAP
- Return R0

Llamada al sistema en kernel turbo()

- Si procesoActual->prioridad == ALTA
  - return
- BCP->duración = SegundosATics(R1)
- BCP->resto = BCP->duración
- BCP->prini = BCP->prioridad
- BCP->prioridad = ALTA

Las modificaciones en el BCP serían: añadir los campos prini, duración, y resto.

b) La nueva funcionalidad sería:

Pseudocódigo Interrupción hardware de reloj ()

- Ticks ++
- InsertarTareaPendiente(comprobarRodajaPlus)
- ActivarInterrupciónSoftware()

Grupo: …… NIA: …… Nombre y apellidos: ……..

Pseudocódigo comprobarRodajaPlus()

- Si procesoActual->prini == ALTA // crearProceso: BCP->prini = BCP->prioridad
  - return
- Si procesoActual->prioridad == ALTA
  - (procesoActual->resto) –
  - Si (procesoActual->resto != 0)
    - return
  - procesoActual->prioridad = BAJA
  - procesoActual->rodaja = 1
- (procesoActual->rodaja)--
- Si procesoActual->rodaja == 0
  - procesoActual->estado = LISTO
  - procesoActual->rodaja = RODAJA_POR_DEFECTO
  - Insertar(**ListaListosBaja**, procesoActual)
  - procesoAnterior = procesoActual
  - procesoActual = planificador()
  - procesoActual->estado = EJECUTANDO
  - activador(procesoAnterior, procesoActual)

c) Las preguntas pedidas:

(1) No porque si un proceso constantemente llama a la función tubo no dejaría ejecutar otros.

(2) Que permitiría una solución que cambie temporalmente la prioridad, no necesitando ser de prioridad alta todo el tiempo un proceso (solo el fragmento que lo precise).

Grupo: …… NIA: …… Nombre y apellidos: ……..

Ejercicio 21

En la empresa en la que estamos trabajando disponemos de un sistema operativo básico (como el que está siendo introducido en pseudocódigo en la asignatura) que usa un *kernel* monolítico con planificación con dos niveles de prioridad, usándose *Round-Robin* para los proceso de baja prioridad y FIFO para los procesos de prioridad alta.

Se pide indicar el diseño e implementación en pseudocódigo de los cambios que hay que hacer en el sistema operativo inicial para añadir la siguiente funcionalidad:

d) Añadir la llamada al sistema:
   int yield (void);
   De forma que el proceso que solicita este servicio pasa al final de la lista de listos y se ejecuta el siguiente proceso que corresponda. No cambia su prioridad y pero si es de baja prioridad el tiempo en CPU a la vuelta será el doble de una rodaja habitual (antes de cambiar a otro proceso) y devolverá 1.

e) Disminuir el efecto de la inanición de la siguiente forma: cuando un proceso de prioridad alta lleva ejecutando lo equivalente a más de tres rodajas de reloj de un proceso de prioridad baja, dicho proceso pasa al final de la lista de prioridad baja, y se pasa a ejecutar el proceso que le correspondiera a ser el siguiente.
   NOTA: Utilice una única interrupción software.

f) ¿Qué efecto tiene esta propuesta contra la inanición si todos los procesos de prioridad alta duran más de tres rodajas?

Grupo: …… NIA: …… Nombre y apellidos: ……..

SOLUCIÓN

a) El pseudocódigo de los eventos pedido podría ser:

Llamada al sistema en espacio de usuario yield (void)

- R0 = YIELD
- Trap
- R0 = 0
- If ProcesoActual->prioridad == BAJA
  - R0 = 1
- Return R0

Llamada al sistema en kernel yield ()

- procesoActual->estado = LISTO
- insertar(listaListosBaja, procesoActual);
- procesoAnterior = procesoActual;
- procesoActual = planificador()
- procesoActual->estado = EJECUTANDO
- activador(procesoAnterior, procesoActual)
- if (procesoActual->prioridad == BAJA)
  - procesoActual->prioridad = 2 * RODAJA_POR_DEFECTO

b) La nueva funcionalidad sería:

Pseudocódigo Interrupción hardware de reloj ()

- Ticks ++
- InsertarTareaPendiente(comprobarRodajaPlus)
- ActivarInterrupciónSoftware()

Pseudocódigo comprobarRodajaPlus()

- Si (procesoActual == NULL)
  - Return
- (procesoActual->rodaja)--
- Si procesoActual->rodaja == 0
  - procesoActual->estado = LISTO
  - Si (procesoActual->prioridad == ALTA)
    - procesoActual->rodaja = 3 * RODAJA_POR_DEFECTO
  - en otro caso
    - procesoActual->rodaja = RODAJA_POR_DEFECTO
  - Insertar(**ListaListosBaja**, procesoActual)
  - procesoAnterior = procesoActual
  - procesoActual = planificador()
  - procesoActual->estado = EJECUTANDO
  - activador(procesoAnterior, procesoActual)

Pseudocódigo Llamada a crear un proceso()

- <Igual que antes>
- Si proceso es de prioridad alta
  - nuevoProceso->rodaja = 3 * RODAJA_POR_DEFECTO
  - Insertar(ListaListosAlta, nuevoProceso)
- Si proceso es de prioridad baja
  - nuevoProceso->rodaja = RODAJA_POR_DEFECTO
  - Insertar(ListaListosBaja, nuevoProceso)

c) El efecto es el de tener *Round-Robin* para todos los procesos donde la duración de la rodaja es el triple para los procesos de prioridad alta.
Se evita inanición (en algún momento entran todos los proceso a ejecutar).

Grupo: …… NIA: …… Nombre y apellidos: ……..

Ejercicio 22

En la empresa en la que estamos trabajando disponemos de un sistema operativo básico (como el que está siendo introducido en pseudocódigo en la asignatura) que usa un *kernel* monolítico con planificación con dos niveles de prioridad, usándose *Round-Robin* para los proceso de baja prioridad y FIFO para los de prioridad alta.

Se desea añadir la siguiente funcionalidad: cambiar la planificación de los procesos de prioridad alta a SJF (*Shortest Job First*) donde los procesos se ordenan según la duración prevista y se ejecutarán en principio en este orden. Para ello la creación de los procesos se indican la prioridad y el tiempo estimado de ejecución. Cuando se supere ese tiempo estimado se penalizará al proceso pasando a prioridad baja y cambiando a ejecutar el proceso siguiente que toque.

Se pide:

a) Indique las estructuras de datos necesarias, modificaciones en el BCP y la modificación de la llamada al sistema de crear proceso necesarias para la funcionalidad descrita:

   int CrearProceso (int prioridad, int duracion);

   En el caso de procesos de prioridad baja, la duración no será tenida en cuenta.

b) Modifique el manejador de interrupción de reloj para que se pueda realizar el cambio de prioridad en caso de que sea necesario. Incluir la interrupción Software en el caso de que sea necesario.

c) Conteste razonando brevemente a las siguientes preguntas:
   3. ¿Existe inanición con esta propuesta?
   4. ¿Qué pasaría con un proceso que intente engañar con su duración?
   5. ¿Qué ventajas e inconvenientes encuentra respecto a la solución inicial?

Grupo: …… NIA: …… Nombre y apellidos: ……..

SOLUCIÓN

a) El pseudocódigo de los eventos pedido podría ser:

Llamada al sistema en espacio de usuario CrearProceso (int prioridad, int duracion)

- R0 = CREARPROCESO
- R1 = prioridad
- R2 = duración
- TRAP
- Return R0

Llamada al sistema en kernel CrearProceso()

- <resto de crear proceso>
- BCP->prioridad = R1
- BCP->duración = SegundosATics(R2)
- BCP->resto = BCP->duración

Las modificaciones en el BCP serían: añadir los campos prioridad, duración, y resto.

b) La nueva funcionalidad implicaría el siguiente pseudocódigo:

Pseudocódigo Interrupción hardware de reloj ()

- Ticks ++
- InsertarTareaPendiente(comprobarRodajaPlus)
- ActivarInterrupciónSoftware()

Pseudocódigo comprobarRodajaPlus()

- Si procesoActual->prioridad == ALTA
  - (procesoActual->resto) –
  - Si (procesoActual->resto ¡= 0)
    - return
  - procesoActual->prioridad = BAJA
  - procesoActual->rodaja = 1
- (procesoActual->rodaja)--
- Si procesoActual->rodaja == 0
  - procesoActual->estado = LISTO
  - procesoActual->rodaja = RODAJA_POR_DEFECTO
  - Insertar(**ListaListosBaja**, procesoActual)
  - procesoAnterior = procesoActual
  - procesoActual = planificador()
  - procesoActual->estado = EJECUTANDO
  - activador(procesoAnterior, procesoActual)

Grupo: …… NIA: …… Nombre y apellidos: ……..

c) Las preguntas pedidas:

(3) Si hay inanición de los procesos de prioridad baja puesto que mientras que haya procesos de prioridad alta previstos que cumplan los plazos, no entran a ejecutar.

(4) Que pasa a ejecutarse con prioridad baja, penalizando dichos procesos.

(5) Que se puede estimar el tiempo máximo de inanición de los procesos de baja prioridad y que al usar SJF para alta ofrece un menor tiempo promedio de ejecución.