DISEÑO DE SISTEMAS OPERATIVOS. Septiembre de 2004

Ejercicio 1

Responda razonada, pero brevemente, a las siguientes cuestiones vinculadas con la planificación de procesos:

a) En numerosos sistemas operativos, cuando el planificador elige el próximo proceso que debe ejecutar, además de la prioridad de cada proceso listo para ejecutar, tiene en cuenta otros factores de cada proceso candidato. Estos factores pueden otorgarle un incremento de su prioridad a la hora de ser elegido de cara a mejorar la eficiencia del sistema. Para los siguientes casos, analice si sería razonable dar un incremento al proceso afectado:

a1) El proceso listo es hijo del que deja el procesador.

a2) En un S.O. con *threads*, el *thread* listo está asociado al mismo proceso que el que deja el procesador.

a3) En un sistema multiprocesador, el proceso listo ejecutó su última racha en el mismo procesador que el proceso que deja de ejecutar.

a4) Una última pregunta relacionada: En Windows 2000 se le otorga un incremento a los procesos que están en ese momento en "foreground". Dadas las características de este sistema, ¿cuál cree que es la interpretación de este término en el mismo (o sea, cuándo se considera que un proceso está en *foreground*)? Asimismo, en Windows 2000 se usa un tamaño de rodaja de ejecución diferente en distintas configuraciones del sistema operativo. Teniendo en cuenta el uso al que va destinado cada sistema, ¿cuál debería tener una menor rodaja, una versión *Professional*, orientada al uso personal, o una de tipo *Server*, ideada para la configuración de servidores?

b) Supóngase un hipotético dispositivo X de uso compartido, de solo lectura y dirigido por interrupciones. El manejador incluye 2 funciones: LeerX, que recibe la dirección de usuario especificada por el proceso donde desea leer el dato, y la rutina de interrupción. Se plantea el esqueleto de 4 versiones simplificadas, 2 de ellas erróneas, del manejador de este dispositivo (en el código, *Bloquear* incluye al proceso actual al final de la lista, *Desbloquear* pone como listo al primer proceso de la lista, *ResMem* reserva memoria del núcleo y *LibMem* la libera, *InserPet* incluye una petición al final de una cola de peticiones incluyendo la dirección donde copiar el dato y *ExtrPet* extrae la primera petición devolviendo dicha dirección):

Versión 1

```
tipo_lista_proc espera;
tipo_lista_proc list_opera;
dato *destino;

LeerX (dato *dir_usu) {
  Si (list_opera!= NULL)
    Bloquear(espera);
  Programar_disp_X();
  destino=dir_usu;
  Bloquear(list_opera);

  Si (espera != NULL)
    Desbloquear(espera);
}

InterrupciónX {
  *destino=reg_datos;
  Desbloquear(list_opera);
}
```

Versión 2

```
tipo_lista_proc espera;
tipo_lista_proc list_opera;
dato buf_int;

LeerX (dato *dir_usu) {
  Si (list_opera!= NULL)
    Bloquear(espera);
  Programar_disp_X();

  Bloquear(list_opera);
  *dir_usu=buf_int;
```

```
    Si (espera != NULL)
      Desbloquear(espera);
  }

  InterrupciónX {
    buf_int=reg_datos;
    Desbloquear(list_opera);
  }
```

Versión 3

```
  tipo_lista_proc lista;
  tipo_cola_peticiones cola;
  LeerX (void *dir_usu) {
    dato *p_buf_int;
    p_buf_int=ResMem();
    InserPet(cola,p_buf_int);
    Si (lista != NULL)
      Bloquear(lista);
    Sino {
      Programar_disp_X();
      Bloquear(lista);
    }
    *dir_usu=*p_buf_int;
    LibMem(p_buf_int);
  }
  InterrupciónX {
    dato *p_buf_int;
    p_buf_int=ExtrPet(cola);
    *p_buf_int=reg_datos;
    Desbloquear(lista);
    Si (lista != NULL)
      Programar_disp_X();
  }
```

Versión 4

```
  tipo_lista_proc lista;
  tipo_cola_peticiones cola;
  LeerX (void *dir_usu) {
    dato *p_buf_int;
    p_buf_int=ResMem();
    InserPet(cola,p_buf_int);
    Si (lista != NULL)
      Bloquear(lista);
    Sino {
      Programar_disp_X();
      Bloquear(lista);
    }
    *dir_usu=*p_buf_int;
    LibMem(p_buf_int);
  }
  InterrupciónX {
    dato *p_buf_int;
    p_buf_int=ExtrPet(cola);
    *p_buf_int=reg_datos;
    Desbloquear(lista);
    Si (lista != NULL) {
      Programar_disp_X();
      Bloquear(lista); }
  }
```

b1) Comparando las dos primeras versiones, ¿cuál es incorrecta y por qué? (diferencias marcadas en cursiva)
b2) Comparando las dos últimas versiones, ¿cuál es incorrecta y por qué? (diferencias marcadas en cursiva)
b3) Compare la eficiencia de las dos versiones correctas analizando cuál genera menos cambios de contexto. Para ello, puede plantearse una traza de ejecución de varios procesos que leen de X simultáneamente.
b4) ¿Cómo se podría usar la técnica de diferir interrupciones mediante la interrupción software en la segunda solución correcta? Muestre en pseudo-código cómo se utilizaría esta técnica en ese caso.

Solución

a) Antes de pasar a analizar los distintos casos planteados en el enunciado, hay que tener en cuenta dos aspectos a la hora de elegir un proceso en un cambio de contexto de manera que se pueda mejorar la eficiencia del sistema:

- Seleccionar un proceso de manera que se reduzca el tiempo que conlleva el cambio de contexto del proceso que deja el procesador al proceso elegido. Hay que resaltar que uno de los factores que más influyen en el coste del cambio de proceso es el tiempo requerido por la invalidación de la TLB (en aquellos procesadores que no incluyen información del proceso en la misma o tienen que reutilizarla). Por tanto, sería bueno evitar esta operación de invalidación, en el caso de que fuera posible.
- Seleccionar un proceso tal que el estado del sistema dejado por el proceso que abandona el procesador pueda influir positivamente en la ejecución del proceso elegido y, en general, en la ejecución de todo el sistema.

Asimismo, conviene resaltar que el incremento de prioridad que se le otorga a un proceso en el caso de que concurra alguna de las circunstancias que se analizarán en los siguiente apartados, no significa que el proceso correspondiente pase a tener una alta prioridad si previamente no la tenía. Se trata de un pequeño incremento que puede hacer que un proceso con una prioridad original ligeramente inferior a la de otro pueda adelantarle a la hora de ser ejecutados. En resumen, este mecanismo de ajuste no desvirtúa el mecanismo general de prioridades.

a1) En sistemas operativos con un modelo de creación de procesos tal que dicha operación implica que se crea un nuevo proceso que ejecuta un nuevo programa (como, por ejemplo, en Windows), el que uno de los procesos listos para ejecutar sea hijo del proceso que abandona el procesador no tiene ninguna implicación a la hora de establecer un posible incremento de la prioridad del mismo. El estado del sistema que queda cuando deja de ejecutar el proceso padre no permite acelerar la ejecución del proceso hijo, puesto que los dos procesos no comparten ninguna información entre sí. Tampoco comparten el mapa de memoria, por lo que habrá que invalidar la TLB en el cambio de contexto. A efectos del cambio de proceso, es como si se tratara de procesos sin ninguna relación, por lo que no tiene sentido proporcionar un incremento de prioridad al proceso hijo.
En sistemas operativos con interfaz POSIX, la creación de un nuevo proceso genera un duplicado del padre, de manera que ambos tienen inicialmente un mapa de memoria idéntico, que paulatinamente se va diferenciando según cada uno de ellos ejecuta y va modificando las páginas de su propio mapa. Por tanto, en principio, podría ser razonable plantear un incremento de prioridad en el hijo siempre que ambos procesos hayan ejecutado muy poco después de la operación de creación, puesto que después de la ejecución del padre podrían quedar residentes en memoria páginas que todavía siguen siendo idénticas en el hijo al no haberlas modificado ninguno de los dos procesos. De manera similar, podría haber información válida en la memoria caché del procesador. Por tanto, si se da esta circunstancia, la ejecución del hijo puede ser más rápida y requerir menos servicio del sistema operativo (un menor número de fallos de página), Sin embargo, sería necesario invalidar la TLB en el cambio de contexto para evitar problemas de coherencia, ya que los mapas son distintos.
De todas formas, la efectividad de esta estrategia es bastante discutible, ya que después de un cierto plazo de tiempo los mapas serán significativamente diferentes y, sobretodo, porque en la mayoría de los casos, después de la creación del hijo, éste realiza un EXEC que crea un nuevo mapa de memoria. En este último caso, no tiene sentido aplicar la técnica puesto que se trata de la mismo situación que en un sistema con un modelo de procesos como el de Windows.

a2) Los *threads* de un mismo proceso comparten los recursos del mismo, incluyendo el mapa de memoria. Por ello, en un cambio entre dos threads del mismo proceso no es necesario invalidar la TLB. Asimismo, toda la información incluida en la caché y residente en memoria principal dejada por el *thread* que abandona el procesador puede ser útil para el nuevo *thread*. Por tanto, es razonable asignar un incremento de prioridad a todos los *threads* en estado de listos para ejecutar que pertenezcan al mismo proceso que el *thread* que abandona el procesador.
En el siguiente fragmento extraído del código fuente de una versión de Linux, se puede observar el incremento de la prioridad que se realiza en este sistema cuando un proceso candidato (*p*) comparte el mapa de memoria con el proceso que deja el procesador (*prev*):

```
/* .. and a slight advantage to the current MM */
if (p->mm == prev->mm)
        weight += 1;
```

Nótese que Linux no implementa realmente *threads* sino procesos de peso variable usando la llamada *clone*, pero a todos los efectos es aplicable la misma idea.

a3) En un multiprocesador, se comparte la memoria principal, pero cada procesador posee su propia memoria caché. Cuando un proceso ejecuta en un procesador, en la memoria caché del mismo se va almacenando información vinculada con dicho proceso (información correspondiente a sus últimos accesos a memoria principal). Si la próxima vez que ejecute ese proceso lo hace en el mismo procesador, podrá encontrar todavía en su memoria caché información relacionada con el proceso, agilizando su ejecución. Por tanto, parece razonable dar un incremento de prioridad a un proceso listo cuya última racha de ejecución haya sido en ese mismo procesador. A esta estrategia se le suele denominar *afinidad al procesador*, o sea, la tendencia a que un proceso ejecute siempre en el mismo procesador. Sin embargo, también hay que tener en cuenta que, según van ejecutando otros procesos en ese mismo procesador, esta información va siendo desalojada del mismo. Cuanto más tiempo haya pasado (y más pequeña sea la memoria caché) desde que el proceso realizara la ejecución de su última racha, menos probable será que haya información vinculada con el proceso en la memoria caché. Resumiendo, el incremento de prioridad parece razonable aunque deberían tenerse en cuenta los factores comentados (tiempo desde la última ejecución y tamaño de la memoria caché) a la hora de otorgar el incremento.
Usando de nuevo Linux como ejemplo, se incluye el siguiente fragmento donde se observa el incremento de prioridad otorgado usando esta estrategia:

```
#define PROC_CHANGE_PENALTY      15
/* Give a largish advantage to the same processor...   */
/* (this is equivalent to penalizing other processors) */
if (p->processor == this_cpu)
        weight += PROC_CHANGE_PENALTY;
```

a4) En general, el término *foreground* se refiere a un proceso que en ese momento está interaccionando directamente con el usuario. Dado el carácter gráfico basado en ventanas de Windows, este término se aplica a los *threads* (y, por extensión a los procesos) interactivos asociados a la ventana que tiene en ese momento el foco, puesto que se trata de la ventana con la que está dialogando el usuario.
Con respecto al tamaño de la rodaja en las dos configuraciones de Windows planteadas en el enunciado, hay que tener en cuenta qué aporta la rodaja en cada caso:

- En la versión *Professional*, hay múltiples procesos de carácter interactivo generados por el trabajo del usuario. El uso de una rodaja asegura que el tiempo de respuesta de estos procesos es pequeño. En el peor de los casos, si hay N procesos en la cola de listos, un proceso que pasa en ese instante a listo tendrá que esperar N multiplicado por el tamaño de la rodaja. Es necesario asegurar un tamaño de rodaja pequeño para tener un buen tiempo de respuesta, aunque siempre asegurando que no hay demasiada sobrecarga por los cambios de contexto involuntarios generados.
- En la versión *Server*, existen múltiples peticiones de clientes simultáneas, que pueden requerir unas necesidades muy variadas: desde unos pocos ciclos del procesador hasta un intervalo de tiempo muy elevado. En esta situación, la rodaja asegura que una petición no acapare el uso del procesador. Lo recomendable sería buscar un tamaño de rodaja tal que la mayoría de las peticiones pudieran satisfacerse sin consumirla. De esta manera, se reduciría la sobre carga debida a los cambios de contexto involuntarios, que podría afectar al rendimiento del servidor.

De las consideraciones anteriores, se puede concluir que en la versión *Professional* la rodaja debe de ser más pequeña: por defecto, es la mitad que la correspondiente a la versión *Server*.

b) Antes de pasar a analizar las cuestiones planteadas, conviene resaltar que se trata de una versión simplificada de un manejador que omite muchos detalles que aparecen en un manejador real. Entre otros muchos aspectos, no se preocupa de ir modificando el nivel de interrupción para asegurar el buen funcionamiento del manejador (por ejemplo, sería necesario prohibir las interrupciones del dispositivo desde que se va iniciar la programación del dispositivo hasta que se bloquee el proceso). Además, hay que hacer notar que, de forma implícita, se deduce que se trata de un manejador que corresponde con un sistema operativo con un kernel no expulsivo, ya que sino no serían válidas ninguna de las soluciones planteadas.

b1) Analizando las dos primeras versiones, se detecta que la primera es incorrecta, ya que en ella se intenta acceder la mapa del proceso que solicitó la lectura directamente desde la interrupción (en la variable *destino* se ha guardado esa dirección). Esto es erróneo puesto que cuando se ejecuta la rutina de interrupción el mapa de memoria activo es el del proceso actualmente en ejecución, que no tiene nada que ver con la petición de lectura, por lo que se estará intentando copiar en el mapa de dicho proceso. Nótese que el proceso lector ha especificado una dirección lógica de su mapa, pero desde la rutina de interrupción se está usando esa dirección lógica aplicada a otro mapa y, por tanto, a otra tabla de páginas.
b2) El análisis de las dos últimas versiones determina que la versión cuarta es la errónea, ya que en ella se invoca a la rutina de bloqueo desde una rutina de interrupción. Se estaría realizando un cambio de contexto dentro de una rutina de

interrupción, lo cual es erróneo, causando el bloqueo del proceso actual, que no tiene nada que ver con la operación en curso.

b3) Comparando las dos versiones correctas (segunda y tercera), se observa que la segunda genera más cambios de contexto. En esa versión, si un proceso encuentra el dispositivo ocupado se bloqueará en la lista de espera. Cuando termine la operación en curso, se desbloqueará ese proceso, que, en el momento que vuelva a ejecutar, programará el dispositivo y se volverá a bloquear. En la tercera versión, sin embargo, el proceso se bloquea sólo una vez, con independencia de si está libre el dispositivo o no, siendo la rutina de interrupción la encargada de programar el dispositivo para la siguiente operación pendiente.

El esquema de trabajo representado por la tercera versión es el más habitual, ya que, además, mantiene al máximo la ocupación del dispositivo, lo cual puede ser crucial si se trata de un disco. Para entender esta última afirmación, téngase en cuenta lo que ocurre en la segunda versión cuando un proceso de baja prioridad queda bloqueado en la lista de espera dado que el dispositivo está ocupado: cuando termina la operación se desbloquea, pero como es de baja prioridad tarda mucho en continuar su ejecución, dejando sin usar el dispositivo.

La única desventaja de la cuarta solución es la inclusión en la rutina de interrupción de una cantidad apreciable de operaciones, lo cual no es conveniente ya que se deben intentar minimizar la duración de las rutinas de interrupción para agilizar el funcionamiento del sistema. El próximo apartado aborda este problema y muestra lo que podría ser un manejador con un esquema comparable con los usados en los sistemas reales.

b4) Para reducir el tiempo que dura la rutina de tratamiento de la interrupción, se deben de dejar en ella sólo las operaciones urgentes, aplazando las restantes al tratamiento vinculado con una interrupción software que ejecutara con un nivel de interrupción mínimo. En muchos casos, las operaciones urgentes implican sólo leer del registro de datos del dispositivo para asegurarse de que el dato no se pierde por la llegada de otra interrupción. En este caso, se ha considerado como urgente la lectura del dato en un buffer interno, aplazando el resto de las operaciones.

```
InterrupciónX {
  dato *p_buf_int;
  p_buf_int=ExtrPet(cola);
  *p_buf_int=reg_datos;
  activar_int_software();
}
Tratar_interrupción_software {
  Desbloquear(lista);
  Si (lista != NULL)
    Programar_disp_X();
}
```