DISEÑO DE SISTEMAS OPERATIVOS 4º. Septiembre de 2003

Ejercicio 1

a) [2 puntos sobre 10] Frente al esquema *tradicional* de preasignación de *swap*, donde se reservan a priori los bloques requeridos (por ejemplo, B_5, B_8 y B_{11}), se presenta una estrategia de preasignación alternativa en la que no se reservan bloques concretos, sino simplemente se compromete el número de bloques necesarios (en el ejemplo, sólo se anota una reserva de 3). Se pide responder razonadamente a las siguientes cuestiones:

a1) ¿Qué ventajas e inconvenientes tiene esta estrategia frente al esquema de preasignación tradicional?

a2) ¿Qué ventajas e inconvenientes tiene esta estrategia frente al esquema sin preasignación?

b) [4 puntos sobre 10] Supóngase que en un sistema con esta nueva estrategia de preasignación, se implantan dos cuotas máximas de uso de recursos por parte de un proceso:

- Máximo tamaño del espacio de direcciones (mapa) del proceso (similar al `RLIMIT_AS` de Linux).
- Máximo consumo del espacio de *swap* por el proceso (similar al *page file quota* de Windows 2000).

Analice razonadamente si en las siguientes operaciones sería necesario comprobar alguna de estas cuotas:

1. Rutina de tratamiento del fallo de página
2. Proyección (`mmap`) compartida de un fichero
3. Proyección (`mmap`) privada de un fichero
4. Proyección (`mmap`) privada de tipo anónima (sin soporte)
5. Llamada al sistema `fork`
6. Rutina de tratamiento del *copy-on-write*
7. ¿Habría que comprobar alguna de estas cuotas durante la gestión del *heap*? En caso afirmativo, ¿en qué rutina del sistema operativo?
8. ¿Habría que comprobar alguna de estas cuotas durante la gestión de la pila? En caso afirmativo, ¿en qué rutina del sistema operativo?

c) [4 puntos sobre 10] Cuando un proceso agota una cuota, normalmente se aborta su ejecución. Considérese un sistema operativo que no permite llamadas al sistema concurrentes (o sea, que los procesos en modo sistema no pueden ser expulsados) que se ejecuta en un uniprocesador. Supóngase que en este sistema se está ejecutando un programa *multithread* al que se le agota la cuota de tiempo de UCP (cuota que se aplica al proceso en su totalidad y no a cada *thread* del proceso) y, por tanto, resulta abortado (se abortan todos sus *threads*).

c1) Especifique en qué distintos puntos de su ejecución se podrá encontrar en ese instante el *thread* que causa que se agote la cuota, distinguiendo posibles estados del *thread*, si estaba en modo usuario o sistema y a qué código hacía referencia su contador de programa. ¿Podría estar el *thread* en medio de una llamada al sistema no bloqueante?

c2) Especifique en qué distintos puntos de su ejecución se podrá encontrar en ese instante el resto de los *threads* del programa abortado, distinguiendo nuevamente los posibles estados, si estaban en modo usuario o sistema y a qué código hacía referencia su contador de programa. ¿Podría estar alguno de estos *threads* en modo usuario? ¿Y alguno podría estar en medio de una llamada al sistema no bloqueante?

c3) Si un *thread* se aborta en medio de una llamada al sistema, puede dejar incoherentes estructuras de datos del sistema operativo (al fin y al cabo, se puede considerar que en este caso un proceso en modo sistema ha sido expulsado). Plantee una forma de evitar este tipo de problemas de coherencia.

Solución

a1) Con el nuevo esquema, la preasignación requiere simplemente descontar del número total de bloques de *swap* disponibles los requeridos por la región implicada. Para llevar la gestión de esta información, bastará con usar una variable que mantenga el número de bloques libres. La reserva real de los bloques de *swap* se pospone hasta que sea estrictamente necesario: en el momento en el que se expulsa por primera vez una página modificada que requiere ser escrita en *swap*. Esta política corresponde con un modo de operación "perezoso", característico de muchas otras operaciones del sistema operativo (la paginación por demanda, el *copy-on-write*, el enlace de bibliotecas dinámicas, etc.), y ahí está la principal ventaja de este método: si una página no es expulsada nunca al *swap*, lo que es bastante probable, no se incurre en la sobrecarga de reservar un bloque.

Evidentemente, cuando se expulsa por primera vez una página modificada que requiere ser escrita en *swap*, habrá que hacer la reserva en *swap* que estaba pendiente. En ese punto reside la desventaja de este esquema: la rutina que realiza esta expulsión (la rutina de fallo de página en el caso de un sistema sin *buffering* de páginas o el demonio de paginación si se trata de un sistema con *buffering*) será menos eficiente en esta estrategia alternativa ya que requiere llevar a cabo la reserva del bloque de *swap*, lo que no se necesita con la preasignación clásica pues ya se había hecho previamente.

En cualquier caso, esta nueva estrategia es ventajosa con respecto a la preasignación clásica. En el peor de los casos, que correspondería con una situación en la que todas las páginas de la región privada se modifican y resultan expulsadas, las dos estrategias realizarían el mismo número de reservas, aunque en distintos momentos: la preasignación clásica lo haría durante la creación de la región, mientras que la nueva alternativa lo realizaría en el momento de la expulsión. En este caso extremo, sería un poco peor la nueva estrategia ya que, además de las reservas, requeriría la actualización del contador de espacio libre en *swap* en la creación de la región. Sin embargo, esta sobrecarga es despreciable. Además, hay que tener en cuenta que se trata de un caso extremo, puesto que lo normal es que haya una cantidad apreciable de páginas que no tengan que escribirse en *swap*.

a2) Las ventajas y desventajas de esta nueva estrategia frente a un esquema sin preasignación son las mismas que se producen entre este último y un sistema de preasignación clásico, a saber:

- Como ventaja, los esquemas con preasignación, sean del tipo que sean, aseguran que cuando se expulsa una página siempre tiene espacio en *swap*, bien sea porque ya tiene reservado un bloque específico, en el caso de una preasignación convencional, o bien porque se ha comprometido el espacio de *swap* correspondiente a un bloque, si se trata del esquema alternativo. Por tanto, el agotamiento del espacio de *swap* siempre se detecta en una operación de creación o expansión de una región, por tanto, de una forma síncrona. El tratamiento consistiría simplemente en no realizar la operación devolviendo un error (por ejemplo, al proyectar un fichero o al solicitar memoria dinámica). Con un esquema sin preasignación puede ocurrir que, cuando se intenta expulsar una página de memoria para habilitar espacio para otra, no haya espacio en *swap*. En este caso, por tanto, la detección de que no hay espacio en *swap* se realiza de forma asíncrona, y habrá que abortar la ejecución de algún proceso.
- La desventaja de los esquemas con preasignación es que con ellos el espacio total disponible para las regiones privadas de los procesos está limitado al tamaño del *swap*, mientras que en las estrategias sin preasignación este límite corresponde con la suma del tamaño del *swap* más la memoria física, pudiendo, por tanto, conseguir un mayor grado de multiprogramación.

b) Para afrontar esta cuestión, conviene tener en cuenta los dos siguientes aspectos:

- Cada vez que se crea una nueva región o se expande el tamaño de una ya existente aumenta el espacio ocupado en el mapa de direcciones del proceso, por lo habrá que comprobar la cuota del tamaño máximo del espacio de direcciones.
- Cada vez que se crea una región privada o se expande el tamaño de una región de este tipo ya existente aumenta la cantidad de espacio comprometido en *swap*, ya que este tipo de regiones requiere soporte en *swap*. Por lo tanto, en estas situaciones habrá que comprobar la cuota del consumo máximo de espacio de *swap* por el proceso.

b1) *Rutina de tratamiento del fallo de página*. Un fallo de página no va a influir en ninguna de estas dos cuotas ya que no implica la creación de una región ni su expansión, a excepción del caso de que se trate de un fallo vinculado con la expansión de la pila, como se analizará en el apartado *b8*. Nótese que, aunque un fallo de página puede causar que se escriba una página en el *swap*, esto no afecta al consumo de espacio de *swap*, puesto que ya tiene espacio preasignado.

b2) *Proyección compartida de un fichero*. La proyección crea una nueva región, por tanto, hay que comprobar la cuota del tamaño máximo del espacio de direcciones. Sin embargo, al tratarse de una región compartida, como tiene soporte en el propio fichero, no hay que verificar la cuota de consumo de *swap*.

b3) *Proyección privada de un fichero*. La proyección requiere comprobar la cuota del tamaño máximo del espacio de direcciones. Además, al tratarse de una región privada, se debe verificar la cuota de consumo de *swap*.

b4) *Proyección privada de tipo anónima*. Al igual que en el caso anterior, requiere supervisar ambas cuotas. El hecho de que la proyección sea anónima no afecta a esta cuestión.

b5) *Llamada al sistema* `fork`. Cuando se crea un nuevo proceso, se está creando un nuevo mapa compuesto por el duplicado de las regiones del padre. Se están creando nuevas regiones para el hijo y, por tanto, se está consumiendo espacio de direcciones del mismo y también espacio de *swap* para aquellas regiones que sean de carácter privado. Sin embargo, no es necesario comprobar ninguna de las cuotas, ya que el hijo las hereda del padre y, por tanto, si el padre las cumplía, también lo hará el hijo.

b6) *Rutina de tratamiento del copy-on-write*. En esta rutina no hay que verificar ninguna de las cuotas ya que no se está creando ni expandiendo una región, simplemente se está duplicando una página de una región privada, y esta página ya tiene preasignado espacio en *swap*.

b7) *Gestión del heap*. Cuando se expande el *heap*, habrá que comprobar ambas cuotas por tratarse de una región de carácter privado. Esta comprobación se llevará a cabo dentro de la llamada al sistema que corresponda con la expansión del *heap* (en el caso de UNIX, las llamadas *brk* y *sbrk*, que son a su vez, invocadas por la rutina *malloc*).

b8) *Gestión de la pila*. De manera similar al *heap*, cuando se expande la pila, habrá que comprobar ambas cuotas por tratarse de una región de carácter privado. Como se explicó en el apartado *b1*, esta comprobación se llevará a cabo dentro de la rutina de tratamiento del fallo de página, cuando corresponde con una expansión de la pila.

c1) El *thread* que causa que se agote la cuota tiene que estar, evidentemente, en ejecución cuando se produce la interrupción de reloj que determina que se ha agotado la cuota del proceso. En cuanto a su modo de ejecución, puede estar en modo usuario o en modo sistema.

Si está en modo usuario, el contador de programa hará referencia a código del programa. En caso de que esté en modo sistema, puede encontrarse en distintas situaciones: en una llamada al sistema, bloqueante o no bloqueante, en el tratamiento de un fallo de página, en una rutina de interrupción de algún dispositivo cuya interrupción tenga menos prioridad que la del reloj, o en cualquier anidamiento factible de esta rutinas. El contador de programa en este caso hará referencia a código de la rutina del sistema operativo interrumpida por el reloj.

c2) Con respecto a los otros *threads* del proceso, al tratarse de un monoprocesador, no podrán en estar en ejecución y, por tanto, estarán todos ellos "parados" en modo sistema, Se presentan las siguientes posibilidades:

- *Thread* recién creado. Está en estado de listo para ejecutar y su contador de programa apuntará a la rutina del sistema operativo en la que se inicia la ejecución de los procesos.
- *Thread* bloqueado. El contador de programa hace referencia a la instrucción justo después del cambio de contexto vinculado al bloqueo del proceso (cambio de contexto voluntario), por donde reanudará la ejecución cuando se desbloquee y sea elegido por el planificador.
- *Thread* listo para ejecutar previamente bloqueado. El contador de programa estará igual que en el caso anterior. Nótese que entre este estado y el anterior el proceso no ha ejecutado, por tanto, el contador de programa no habrá cambiado.
- *Thread* listo para ejecutar previamente en ejecución. El contador de programa hace referencia a la instrucción justo después del cambio de contexto realizado dentro de la rutina de tratamiento de la interrupción software (cambio de contexto involuntario).

Por tanto, ninguno de estos *threads* puede estar en modo usuario, puesto que un proceso que no ejecuta está parado en modo sistema, ni realizando una llamada al sistema, dado que no se permiten llamadas al sistema concurrentes y, por tanto, una llamada al sistema continúa hasta que termina o hasta que el proceso se bloquea dentro de ella.

c3) Para evitar estos problemas de sincronización, se puede utilizar el mecanismo de interrupción software para diferir la operación de abortar el *thread* que causa que se agote la cuota del procesador hasta que haya terminado todo el trabajo del sistema operativo que pueda estar anidado. Así, de manera similar a como se implementa el *round-robin*, la rutina de interrupción del reloj detectará que se le ha acabado la cuota de ejecución al proceso y activará una interrupción software marcando previamente la circunstancia acaecida. Podría ser algo similar a lo siguiente:

```
int_reloj()  {
      ...........
      Si (cuota de proceso actual agotada)
            thr_actual -> abortado =TRUE;
            activar_int_SW();
      }
      ..........
}
```

Y dentro de la rutina de tratamiento de la interrupción software se procedería a abortar el proceso:

```
int_sw()  {
      ...........
      fijar_nivel_int(NIVEL_MAX);
      Si (thr_actual -> abortado) {
      abortar_los_otros_threads ▪ Pendiente de analizar
            terminar_proceso(thr_actual);
      }
      ..........
      cambio_contexto(...);
      ..........
}
```

Sin embargo, quedan pendientes de resolver dos problemas:

- Si el *thread* estaba realizando una llamada bloqueante cuando llegó la interrupción de reloj y al reanudarla posteriormente se bloquea, no se producirá la operación de abortado del proceso, al menos hasta que vuelva a ejecutar y se vea involucrado en un cambio de contexto involuntario.
- ¿Qué ocurre con los otros *threads* del proceso?

Con respecto al primer problema, se podría comprobar siempre antes de bloquearse si el *thread* está marcado como abortado, abortándose en este caso la ejecución del mismo. Sin embargo, si este es el caso, no se podría abortar inmediatamente puesto que el *thread* está bloqueándose dentro de una llamada al sistema y puede haber dejado bloqueados recursos del sistema (por ejemplo, una llamada de escritura en un fichero puede bloquear el *inodo* correspondiente para asegurarse del carácter atómico de la escritura). Por tanto, habría que dejar que el *thread* libere estos recursos antes de ser abortado. Podría plantearse que la operación de bloqueo no se llevase a cabo si el proceso está marcado como abortado devolviendo un valor para indicarlo. Se podría usar un esquema como el siguiente:

```
int Bloquear()  {
      ...........
```

```
        fijar_nivel_int(NIVEL_MAX);
        Si (thr_actual -> abortado)
              return -1;
        ..........
        cambio_contexto(...);
        ..........
        return 0;
  }
```

Y la hipotética llamada al sistema bloqueante se comportaría de la siguiente manera:

```
  int llamada()  {
        ...........
        Establece cerrojos sobre recursos
        ..........
        Bajo ciertas condiciones la llamada invoca a Bloquear
        estado= Bloquear();
        Si (estado==-1) {// No se ha bloqueado; ha sido abortada
              liberar recursos
        abortar_los_otros_threads() ▪ Pendiente de analizar
              terminar_proceso(thr_actual);
        }
        Procede después del bloqueo
        ..........
  }
```

Por lo que se refiere a la segunda cuestión pendiente (¿cómo abortar los otros *threads*?), dependerá del estado en que se encuentre cada uno de ellos:

- *Thread* recién creado. Dado que todavía no ha ejecutado, se puede abortar directamente:

```
abortar_los_otros_threads ()  {
      Por cada thread T del proceso excluyendo el que estaba en ejecución
            Si el thread T está recién creado
                  terminar_proceso(T);
      ..........
}
```

- *Thread* listo para ejecutar previamente en ejecución. De manera similar al caso previo, en esta situación también se podría abortar directamente el *thread* ya que está a punto de volver a modo usuario y no tendrá reservado ningún recurso del sistema operativo. Sin embargo, también se puede plantear una solución alternativa en la que es el propio thread el que aborta su ejecución:

```
abortar_los_otros_threads ()  {
      Por cada thread T del proceso excluyendo el que estaba en ejecución
            Si el thread T está recién creado
                  terminar_proceso(T);
            Si el thread T está listo por un c. cont. involuntario previo
                  T -> abortado =TRUE;
      ..........
}
```

Cuando vuelva a ejecutar, lo hará desde la rutina de interrupción software, donde el mismo terminará su ejecución. La nueva versión de esta rutina sería:

```
int_sw()  {
      ...........
      fijar_nivel_int(NIVEL_MAX);
      Si (thr_actual -> abortado) {
      abortar_los_otros_threads();
            terminar_proceso(thr_actual);
      }
      ..........
      cambio_contexto(...);
      // Va a volver a usuario, puede que esté abortado
      Si (thr_actual -> abortado)
            terminar_proceso(thr_actual);
      ..........
}
```

- *Thread* bloqueado. Como ocurría con el *thread* en ejecución en el caso de que se quedará bloqueado después de cumplirse la cuota, en este caso hay que realizar una terminación “cuidadosa” para asegurarse de que el *thread* libere todos los recursos que podía tener bloqueados antes de bloquearse. Previamente, habría que desbloquearlo, como se aprecia a continuación:

```
abortar_los_otros_threads ()  {
      Por cada thread T del proceso excluyendo el que estaba en ejecución
            Si el thread T está recién creado
                  terminar_proceso(T);
```

```
            Si el thread T está listo por un c. cont. involuntario previo
                  T -> abortado =TRUE;
            Si el thread T está bloqueado {
                  Ponerlo como listo quitándole de la cola de espera
                  T -> abortado =TRUE;
            }
      ..........
}
```

La rutina de bloqueo se modificaría para preguntar por la posibilidad del aborto después del cambio de contexto:

```
int Bloquear()  {
      ...........
      fijar_nivel_int(NIVEL_MAX);
      Si (thr_actual -> abortado)
            return -1;
      ..........
      cambio_contexto(...);
      Si (thr_actual -> abortado)
            return -2;
      ..........
      return 0;
}
```

Por último, la llamada al sistema hipotética que produce el bloqueo quedaría:

```
int llamada()  {
      ...........
      Establece cerrojos sobre recursos
      ..........
      Bajo ciertas condiciones la llamada invoca a Bloquear
      estado= Bloquear();
      Si (estado<0) {// No se ha bloqueado; ha sido abortada
            liberar recursos
            Si (estado==-1) // se trata del thread que agota la cuota
            abortar_los_otros_threads();
            terminar_proceso(thr_actual);
      }
      Procede después del bloqueo
      ..........
}
```

- *Thread* listo para ejecutar previamente bloqueado. Este caso es prácticamente igual al anterio, excepto que no hay que poner a listo el proceso. Con ello, se completan los casos posibles quedando así la rutina correspondiente:

```
abortar_los_otros_threads ()  {
      Por cada thread T del proceso excluyendo el que estaba en ejecución
            Si el thread T está recién creado
                  terminar_proceso(T);
            Si el thread T está bloqueado {
                  Ponerlo como listo quitándole de la cola de espera
                  T -> abortado =TRUE;
            Si el thread T está listo (sea cuál sea el estado previo)
                  T -> abortado =TRUE;
}
```

Nótese que hay cierta regularidad en los casos planteados: antes y después del cambio de contexto hay que comprobar si el proceso ha sido abortado, con independencia de si se trata de un cambio de contexto vinculado con la interrupción software (cambio involuntario) o con un bloqueo (cambio voluntario). Por tanto, podría especificarse una rutina común que englobará estas comprobaciones, junto con el cambio de contexto.