Grupo 7

Trabajo Práctico Nro 2

Núcleo de un S.O. y estructuras de administración de recursos

Integrantes:

Brula, Matías	*Legajo 58639*
Tallar, Julián	*Legajo 59356*
Vuoso, Julián	*Legajo 59347*

Introducción

Para la realización de este trabajo práctico, tomamos como base el trabajo práctico realizado en Arquitectura de Computadoras, el cual ya tenía las secciones kernel y userland vinculadas por syscalls, un driver de video en modo gráfico y un manejo de interrupciones básico. A esa base se le agregó:

- Administrador de memoria física (que puede ser manejado con free list first fit o con buddy system)
- Manejo de procesos
- Scheduler con context switching, capaz de cambiar el proceso en ejecución a partir de un algoritmo de round robin con prioridad
- Sincronización por medio de mutexes y semáforos
- Comunicación entre procesos por medio de pipes
- Aplicaciones necesarias para mostrar la funcionalidad agregada

Decisiones tomadas

ADMINISTRADOR DE MEMORIA FÍSICA

Como dirección inicial del Memory Manager, tomamos la última dirección de módulo cargada y lo alineamos al tamaño de página (4KB) preexistente en el kernel original. Esta dirección, junto con el tamaño total de espacio de memoria elegido para administrar se envían como parámetros para crear el administrador durante la inicialización del kernel. El tamaño total es fijo, seteado con una variable global en kernel.c y elegimos un tamaño de 512 MB.

Para elegir entre los dos algoritmos en tiempo de compilación, utilizamos un flag -DBUDDY, que se agrega dentro del Makefile de la carpeta kernel, en la línea 11:

MM = -DBUDDY para correr con Buddy
MM = para correr con First Fit Free List

Esto permite usar un ifndef para ver en tiempo de compilación qué funciones tomar: las del Buddy o las del First Fit Free List. Si no se especifica nada, se utiliza el algoritmo de First Fit Free List.

A continuación, detallamos las decisiones tomadas específicas del algoritmo de First Fit Free List.

En primer lugar, guardamos una estructura estática con la información relevante del administrador: puntero a lista de bloques libres y usados, cantidad de bloques libres y usados y tamaño de bloque.

Previo a cada bloque a malloquear, se guarda un nodo con la información del bloque creado. El nodo contiene dirección de memoria de comienzo de bloque, cantidad de bloques requeridos, y direcciones de nodos previo y siguiente en la lista correspondiente. Cada uno de estos bloques está alineado a 64 bits y su tamaño es de 2 veces el tamaño del nodo. Es decir, si un proceso pide una cantidad de bytes menor al tamaño de bloque, se le asignará un bloque entero. Al retornar, se devuelve la dirección de memoria a

partir de la cual puede guardar datos (dirección de bloque + tamaño de nodo).

Cuando se hace un malloc, se busca un nodo con el tamaño suficiente para satisfacer el pedido. Si se lo encuentra, se lo subdivide (en caso de tener bloques de más), se lo extrae de la lista de bloques libres y se lo inserta en la lista de bloques ocupados. En caso de hacer un free, se busca el nodo en la lista de bloques usados, y si se lo encuentra, se lo quita de la misma, se inserta en la lista de libres y se hace un merge en la misma en caso de ser haber bloques libres contiguos en memoria.

Con respecto a la implementación del algoritmo de Buddy, decidimos implementarlo por medio de un árbol binario por la naturaleza del algoritmo. En este caso, decidimos guardar los nodos en un vector estático al inicio del espacio asignado para el administrador de memoria al inicializar el kernel.

La altura del árbol se define en base al espacio restante dentro del asignado. Se va reduciendo la altura desde el valor máximo posible hasta que quede espacio suficiente para guardar TODOS los nodos posibles del árbol. Los espacios recibidos serán convertidos a potencia de 2.

En cuanto a las funciones de reserva y liberación de memoria, el subárbol izquierda tendrá prioridad respecto del derecho. Es decir, primero chequea si tiene espacio del lado izquierdo, sino irá al derecho. Con respecto a la liberación de nodos, no solo se busca un nodo que apunte a la dirección recibida, sino también uno que sea hoja (no tenga hijos) o que ambos hijos estén liberados.

PROCESOS

Todos los procesos ejecutables serán cargados en memoria física al inicializar el kernel en las direcciones o entrypoints especificados en moduleAddresses.h. De esta manera, si queremos crear un proceso, se busca cuál de las direcciones corresponde al nombre recibido.

Para cada proceso, tendremos una estructura con sus propiedades, punteros al stack, estado y recursos utilizados. Su stack inicial se configura de manera tal que, al cambiar de contexto y popear su estado, los registros queden configurados adecuadamente cuando comience su ejecución. Además, en la parte más baja del stack, agregamos como dirección de retorno la dirección de la función kill_current, que matará al proceso al finalizar su ejecución. También configuramos los registros rdi y rsi para que los procesos reciban argumentos.

Además de recibir su entrypoint, reciben su nombre, contexto (Foreground o Background), parámetros (argc y argv igual al estándar de C), alias para STDIN y alias para STDOUT. A continuación, explicamos para qué se usa el contexto y los alias:

- El contexto determinará únicamente si el proceso que crea este proceso nuevo debe bloquearse (en caso de ejecutarse en foreground) o no.

- Los alias de STDIN y STDOUT se utilizarán para crear procesos conectados mediante | , de forma tal de poder indicarle que no debe leer de STDIN o escribir en STDOUT, sino en otro file descriptor.

Por último, cada proceso guardará una lista de file descriptors a los que tiene acceso junto con el alias de cada uno. La lista será actualizada al crear el proceso y al abrir y cerrar pipes.

SCHEDULING Y CONTEXT SWITCHING

Con respecto al context switching, cada vez que salta la interrupción del timer tick se llama a la función scheduler con el stack pointer como parámetro, y se actualiza el valor del mismo a partir del valor de retorno de la función. En el caso de las syscalls, únicamente se forzará una interrupción del timer tick en caso de que se bloquee o se mate al proceso actual.

Con respecto al scheduler, éste será inicializado durante la inicialización del kernel. Aquí decidimos crear un proceso Idle, que llama a la syscall halt en un loop infinito. La función scheduler retornará su stack pointer en caso de no haber otros proceso para correr o que no haya ninguno listo.

Por cada proceso creado, se crea un nodo dentro de una lista del scheduler donde se contabilizará la cantidad de veces que se le asignó el procesador de manera consecutiva. Cuando el proceso alcance la cantidad correspondiente o se bloquee, se le asignará el procesador al proceso siguiente (puede ser él mismo si es no se ha bloqueado o Idle en caso de no haber ninguno listo).

Respecto al algoritmo de round robin con prioridad utilizado, determinamos cuatro niveles de prioridad (0 a 3), siendo 0 la prioridad más alta. Cada proceso se ejecutará durante 2 ^ (3 - prioridad) veces, por lo que el ciclo de un proceso de prioridad 0 será 8 veces mayor que el de uno de prioridad 3 (siempre y cuando no se bloquee antes).

En cuanto al kill de procesos, se lo quitará de la lista y se liberarán todos los recursos ocupados por el mismo. Esto puede ser semáforos adquiridos, bloqueo de pantalla (en caso de haber sido ejecutado en foreground) o nodos de sleep.

En caso de no haber más procesos en la lista por eliminar a la shell, se deshabilitan las interrupciones y se ejecuta halt dentro de kernel.

SINCRONIZACIÓN

Decidimos implementar Mutexes (semáforos binarios) y Semáforos enteros a partir de Mutexes, los dos con NOMBRE. En ambos casos, al crear uno nuevo se busca si ya existía otro con su nombre. De ser así, decidimos retornar una referencia a este mutex existente, en lugar de devolver error. Esto se debe a que muchas aplicaciones realizadas crean semáforos.

Dentro de cada mutex, guardamos sus propiedades, el primer elemento de una lista de procesos bloqueados por este mutex y una variable Lock, que será modificada de manera atómica al realizar operaciones sobre el mutex. Si

el campo Lock está en 0, se lo puede adquirir, mientras que si el campo está en 1, ya ha sido adquirido.

Si un proceso hace un wait sobre un mutex y no está adquirido, lo adquiere y retorna, mientras que si el mutex ya había sido adquirido se lo agrega a una lista de procesos bloqueados por él y se lo bloquea. Si un proceso hace un post sobre un mutex que no ha bloqueado a ningún otro proceso, éste desbloquea el mutex (de ser necesario) y retorna. En caso contrario, desbloquea al primer proceso de la lista de bloqueados.

En cuanto a su valor inicial, todo valor mayor o igual que 1 se interpretará como bloqueado. Cuando se mata a un proceso que había sido bloqueado por un mutex, se recorre la lista de mutexes buscando aquellos que habían sido adquiridos por él. De ser así, se restaura su valor inicial, es decir, si había comenzado desbloqueado, se le hace un post.

Con respecto a los semáforos enteros, fueron implementados utilizando dos mutexes y un contador. Con un mutex, protegemos al contador, evitando modificaciones concurrentes, mientras que con el otro bloqueamos al proceso en caso de ser necesario.

Los procesos ejecutados en Userland sólo tendrán acceso a los semáforos enteros, ya que consideramos innecesario crear todas las syscalls necesarias para que tengan acceso a ambos. No pierden funcionalidad ya que, de necesitar un mutex, pueden crear un semáforo entero de cuenta 0 o 1 y manejarlo de manera adecuada.

IPC

En primer lugar, decidimos mantener una misma estructura para todo file descriptor, ya sea de un pipe o uno estándar. Por cada uno de ellos, guardamos sus propiedades, un buffer circular de 255 caracteres y 3 mutexes: uno para escritura y dos para lectura. Los file descriptors estándar son creados por el kernel al inicializarse. Los pipes creados serán con NOMBRE y BIDIRECCIONALES.

Con respecto a los mutexes utilizados, cuando un proceso quiera leer o escribir, hará un wait sobre el mutex correspondiente, de forma tal que solamente uno pueda leer o escribir en ese file descriptor en un mismo instante. En el caso de la lectura, agregamos un mutex más para cuando se desee leer una cantidad de caracteres mayor a la disponible. Éste mutex será desbloqueado al escribir en el file descriptor, ya sea por alcanzar la cuenta pedida o por el envío de un EOF.

Respecto al buffer circular, decidimos que en caso de pasarse del límite de 255 caracteres, vuelva a escribir desde el inicio, por lo que al momento de leer caracteres, se leerá la cuenta pedida (o hasta EOF) desde la posición inicial. El EOF se enviará en caso de cerrar un pipe cuyo mutex de escritura no esté bloqueando a otro proceso, ya que en ese caso habría más información para guardar en el buffer.

Como dijimos anteriormente, cada proceso mantendrá una lista de file descriptors abiertos por el mismo, junto con el alias que le corresponda. Al momento de llamar a una syscall write o read en un file descriptor, se buscará

el alias correspondiente a ese número de fd. En caso de no existir, no podrá ejecutar la operación. Los procesos sólo podrán agregar entradas de pipes creando o abriendo pipes existentes. Sus file descriptors de entrada y salida estándar serán asignados al momento de su creación.

Respecto a la escritura de STDOUT y STDERR, decidimos directamente llamar a las funciones de impresión de la consola, ya que sino se debía modificar el funcionamiento de la misma. En el caso del teclado, éste sí guardará en un buffer los valores a medida que se ingresan, de la misma manera que sucede con los otros file descriptors.

APLICACIONES DE USERSPACE

En primer lugar, decidimos que varios de los comandos disponibles sean funciones built-in en lugar de crear un proceso correspondiente, ya que muchas de ellas no hacían más que imprimir en pantalla o cambiar un estado.

Los procesos disponibles para crear son Idle, Shell (estos dos solo pueden ser creados y modificados por el kernel), Sleep, Loop, Cat, Filter, Wc, Process A y B (para programa Sync) y Phylo, Phylo Process y Phylo View (para programa de filósofos. Estos procesos ya tienen una dirección de memoria asignada donde se cargan al ejecutar. Como no tenemos FyleSystem, no podemos cargar otros procesos a memoria.

Por otro lado, los comandos que imprimen estados de los distintos elementos (mem, ps, sem y pipe) imprimen solamente en STDOUT, aún si se ejecuta con pipes. Tomamos esta decisión porque teníamos que modificar los recorridos de la lista para la impresión y creemos que era mucho cambio en el código de impresión cuando no suma funcionalidad. Y si ejecutamos alguno con un pipe, el hecho de cerrar el pipe luego de ejecutar el comando manda un EOF, que indica que no hay nada para leer, por lo que no habrá problemas.

Respecto a los procesos ejecutados desde la terminal, agregamos la posibilidad de enviar una señal de EOF usando Ctrl + D o una señal de interrupción de proceso desde el teclado con Ctrl + C. El primero es útil en el caso de los programas que leen entrada desde STDIN ya que se puede indicar que terminó el ingreso. El segundo sirve para matar al proceso actual (siempre que no sea Shell).

En cuanto a la ejecución de procesos en background, se ejecutan colocando un espacio luego del comando (junto con sus argumentos) y el caracter '&'. Por ejemplo, podemos ejecutar loop &. Solo pueden ejecutarse en background aquellos procesos que no lean de STDIN. Si se ejecuta un comando built-in en background, será lo mismo que hacerlo en foreground, ya que lo único que modifica esto es si bloquea o no a la terminal al ejecutarse.

Luego de ejecutar este proceso en background, deja de leer argumentos, por lo que si se concatena este proceso con otro por medio de un pipe no lo ejecutará. Sí se puede usar pipes previo a la ejecución de un proceso en background.

Para redirigir la salida de un proceso a otro, se debe escribir un '|' entre ellos, con un espacio antes y después de la misma. Por ejemplo, podemos ejecutar help | cat.

Por último, los comandos built-in kill, block y nice se utilizan para controlar el estado de los procesos. Estos pueden aplicarse en cualquier proceso que no sea Shell o Idle. En el caso del nice, si se aplica con un número de prioridad mayor al máximo (3), se fijará el número máximo disponible.

Instrucciones de compilación y ejecución

Para **compilar** el código del proyecto completo, hay que dirigirse a la carpeta del mismo y ejecutar:

```
make all
```

Esto es válido tanto dentro de la imagen de docker provista por la cátedra como fuera de ella.

Para **ejecutar** el sistema operativo, hay que dirigirse a la carpeta del proyecto y ejecutar:

```
./run.sh
```

También se puede compilar y ejecutar en un mismo comando por intermedio de docker, ejecutando:

```
./docker.sh userName
```

Pasos a seguir para demostrar el funcionamiento de cada uno de los requerimientos

Al ejecutar el sistema operativo, se inicia la shell, con un mensaje de bienvenida. Presionando help, podremos ver todos los comandos que se pueden ejecutar.

Para probar el **algoritmo de scheduling y el manejo de procesos**, podemos ejecutar dos procesos loop en background y a continuación bloquearlos. Cada loop, al ejecutarse, mostrará su PID, por lo que podemos bloquearlos ejecutando **block PID.** Podemos imprimir el estado de los procesos para ver que se están corriendo dos procesos de nombre LOOP en background, que están bloqueados.

A continuación, cambiamos la prioridad de UNO de los procesos loop, ejecutando **nice PID PRIORITY.** Como prioridad, asignamos el **valor 0 o 1** (prioridades más alta) para poder ver la diferencia de ejecución entre uno y otro. Nuevamente imprimimos el estado de los procesos por medio de ps para ver que se haya modificado al valor deseado.

Nuevamente ejecutamos **block PID** para cada uno de los PID de los procesos para verlos correr nuevamente. Aunque no se llegue a distinguir por la velocidad de impresión, podemos ingresar **CTRL + C, CTRL + C (DOS VECES RÁPIDO)** para cortar la ejecución de ambos loop corriendo. Así, podremos ver según las últimas impresiones que el proceso al que le asignamos mayor prioridad se ejecutó más veces que el de menor prioridad (podría repetirse el

mismo ejercicio con las prioridades iniciales para verificar que antes se ejecutaban un número de veces similar).

Para verificar el funcionamiento de los pipes, podemos ejecutar el comando help, que imprime en pantalla la lista de comandos disponibles y pipearlo con uno o más comandos de análisis de texto, como pueden ser filter, cat o wc. Hay que tener en cuenta la limitación mencionada respecto al buffer de cada file descriptor. No se guardará el mensaje entero de help, sino las últimas líneas.

Por ejemplo, podemos ejecutar **help | filter | cat**. Filter tomará la salida de help, eliminando las vocales y se la mandará a cat, quien imprime lo recibido y lo replica al encontrar un Enter. La salida del cat será la mostrada en pantalla. También podemos ejecutar directamente filter o cat en primer lugar, ingresar nosotros (no se verá en salida estándar porque se redirige) y luego mandar un EOF con CTRL + D.

Si ejecutamos el comando pipe, podremos ver los pipes creados para cada uno de los | ingresados. Su estado será cerrado, ya que ningún proceso los tiene abiertos.

Para probar el funcionamiento de los semáforos, creamos una aplicación llamada **sync**, que cuenta con dos procesos. El proceso A cuenta con un contador global y crea N procesos B, enviándole a cada uno la dirección de memoria donde se encuentra este contador.

Cada proceso B espera por un semáforo y entra en un ciclo de M iteraciones, donde almacena el valor actual del contador, espera sin bloquearse en un ciclo de código y luego guarda el valor almacenado + 1 en el contador global. Cuando termina de iterar, realiza un post en el mutex abierto.

Como salida, veremos cómo se van creando los diferentes procesos contadores, junto con sus PID. Cada proceso imprime el valor en el que quedó el contador al finalizar, y finalmente el proceso A (luego de hacer un sleep) imprime el valor de la cuenta final. Si la sincronización fue correcta, debería verse como resultado N * M. En este caso, como N = 10 y M = 1000, se debe mostrar 10000 como cuenta final.

También existe la aplicación **phylo**, que implementa el problema de los filósofos comensales, que utilizan los mismos cubiertos para comer. Si ejecutamos la aplicación, se crean inicialmente 4 filósofos, que irán ciclando entre estados (trabajando, esperando para comer y comiendo). Cuando un filósofo trabaja o come, simplemente hace un sleep de un tiempo determinado.

Ingresando la tecla ‘a’ agregamos filósofos (MÁXIMO 15 ya que los vectores son estáticos). Por medio de la tecla ‘r’ removemos un filósofo. Y con la tecla ‘q’ salimos de la aplicación.

Hay un proceso padre, que crea los filósofos y controla la entrada de teclado; un proceso vista que imprime el estado de la mesa; y N procesos filósofos que se comportan como tales.

Si ejecutamos **sem**, podremos ver todos los semáforos en uso y sus estados, tanto los mutexes como los semáforos con cuenta.

El administrador de memoria entra en juego en cada uno de los procesos ejecutados, ya que cada nodo agregado en alguna lista hace un malloc, y cada proceso que termina se libera del scheduler. Si quisiéramos ver explícitamente cómo crece el espacio ocupado, podemos ejecutar **mem**, luego hacer **help | filter**, que creará un pipe junto con su nodo como file descriptor. Si ejecutamos **mem** nuevamente, veremos cómo aumentó la cantidad de memoria ocupada.

Limitaciones

La principal limitación tiene que ver con la escritura de file descriptors. Como dijimos anteriormente, al ser buffers circulares de 255 caracteres, el hecho de no realizar lecturas mientras se está escribiendo puede ocasionar que los índices peguen la vuelta y se tome parcialmente la información escrita. Esto se puede ver si ejecutamos comandos conectados con pipes. Como estos procesos se ejecutan en Foreground, el proceso siguiente no se ejecuta hasta no terminar la ejecución del primero. Por ejemplo, si ejecutamos en la shell help | cat , solo se imprimen las últimas líneas que imprime el comando help porque no entró todo en el buffer y pegó la vuelta.

Otra limitación que tenemos tiene que ver con el espacio asignado al stack de cada proceso. Este tamaño está fijado en 3000 bytes por proceso. Si un proceso comienza a ejecutar funciones recursivamente sin parar y llena su stack, nadie controla que se siga llenando, por lo que comenzará a pisar otra información y dejará de funcionar. Lo mismo sucede si se usa un malloc para reservar espacio y un proceso se pasa del espacio asignado. Nadie controla que pueda acceder a ese espacio de memoria.

Por último, no se alcanzó a probar completamente el funcionamiento de la aplicación phylo, suponemos que funciona, pero no podemos garantizarlo.

Problemas encontrados

El principal problema que encontramos fue al realizar el memory manager. Nos había faltado considerar un caso al realizar un malloc de memoria, por lo que en algún caso particular quedaban inconsistencias en las listas de usados. Esto generaba que el stack pointer quedara apuntando a una dirección fuera de nuestro control y se colgaba el problema.

Por otro lado, decidimos mantener la parte de excepciones como posibles comandos a ejecutar (igual que como estaba del trabajo práctico de Arquitectura de Computadoras), pero tuvimos inconvenientes en el retorno o rebooteo de la Shell, por lo que resolvimos las excepciones matando al proceso que la generó.

Otro inconveniente es que tuvimos problemas con la liberación de recursos al matar un proceso. Si matábamos un proceso que lee de STDIN con Ctrl + C, el mutex quedaba bloqueado indefinidamente y los procesos

nunca se desbloqueaban. Para solucionarlo, recorremos la lista de mutexes buscando cuáles fueron asignados al proceso matado.

Por último, la aplicación phylo. Tuvimos inconvenientes para encontrar el/los motivos en el incorrecto funcionamiento del mismo. Lo que sucedía es que no cumplía del todo bien la secuencia de repartija de recursos en la mesa de los filósofos, y a su vez, dada una cierta cantidad de filósofos agregados (problema hallado con 6) el programa se cuelga. Creemos que el problema ha sido solucionado, ya que el sistema no se cuelga más. De todos modos, la implementación y utilización de los semáforos se puede observar al ejecutar sync en la shell, tal como se explicó en los pasos.

Citas de fragmentos de código reutilizados

Para la realización del programa de filósofos, basamos nuestra implementación en la que se presenta en el libro de Sistemas Operativos de Tanenbaum (sugerido por la cátedra), que se encuentra en la página 166, en el capítulo 2.

166 PROCESOS E HILOS CAPÍTULO 2

```
#define N                   5                   /* número de filósofos */
#define IZQUIERDO           (i+N-1)%N           /* número del vecino izquierdo de i */
#define DERECHO             (i+1)%N             /* número del vecino derecho de i */
#define PENSANDO            0                   /* el filósofo está pensando */
#define HAMBRIENTO          1                   /* el filósofo trata de obtener los tenedores */
#define COMIENDO            2                   /* el filósofo está comiendo */
typedef int semaforo;                           /* los semáforos son un tipo especial de int */
int estado[N];                                  /* arreglo que lleva registro del estado de todos*/
semaforo mutex = 1;                             /* exclusión mutua para las regiones críticas */
semaforo s[N];                                  /* un semáforo por filósofo */

void filosofo(int i)                            /* i: número de filósofo, de 0 a N−1 */
{
     while(TRUE){                               /* se repite en forma indefinida */
          pensar();                             /* el filósofo está pensando */
          tomar_tenedores(i);                   /* adquiere dos tenedores o se bloquea */
          comer();                              /* come espagueti */
          poner_tenedores(i);                   /* pone de vuelta ambos tenedores en la mesa */
     }
}

void tomar_tenedores(int i)                     /* i: número de filósofo, de 0 a N−1 */
{
     down(&mutex);                              /* entra a la región crítica */
     estado[i] = HAMBRIENTO;                    /* registra el hecho de que el filósofo i está hambriento */
     probar(i);                                 /* trata de adquirir 2 tenedores */
     up(&mutex);                                /* sale de la región crítica */
     down(&s[i]);                               /* se bloquea si no se adquirieron los tenedores */
}

void poner_tenedores(i)                         /* i: número de filósofo, de 0 a N−1 */
{
     down(&mutex);                              /* entra a la región crítica */
     estado[i] = PENSANDO;                      /* el filósofo terminó de comer */
     probar(IZQUIERDO);                         /* verifica si el vecino izquierdo puede comer ahora */
     probar(DERECHO);                           /* verifica si el vecino derecho puede comer ahora */
     up(&mutex);                                /* sale de la región crítica */
}

void probar(i)              /* i: número de filósofo, de 0 a N−1 */
{
     if (estado[i] == HAMBRIENTO && estado[IZQUIERDO] != COMIENDO && estado[DERECHO] != COMIENDO) {
          estado[i] = COMIENDO;
          up(&s[i]);
     }
}
```

Para la realización de semáforos enteros en base a mutexes, tomamos como referencia la implementación propuesta por Barz que encontramos en el siguiente link:

https://www.cs.tufts.edu/comp/150FP/archive/john-trono/semaphores.pdf

```
type
        semaphore = record                      // assumes initvalue >= 0 for this implementation and is assigned
                mutex = 1 : binarysemaphore;    // when the semaphore variable is created.
                delay = min(1, initvalue) : binarysemaphore; // choose smaller of 1 and the specified initial value.
                count = initvalue : integer;    // start this counting semaphore out at the initial value.
        end;

procedure P(s : semaphore);
begin
        PB(s.delay);
        PB(s.mutex);
        s.count := s.count - 1;
        if s.count > 0 then
            VB(s.delay);
        VB(s.mutex);
end;
```

```
procedure V(s : semaphore);
begin
        PB(s.mutex);
        s.count := s.count + 1;
        if s.count = 1 then
            VB(s.delay);
        VB(s.mutex);
end;
```