TRABAJO PRÁCTICO 2
SISTEMAS OPERATIVOS

Martín Victory	56086
Marco Fallone	48179
Florencia Cavallin	56015
Florencia Monti	55073

ÍNDICE

Physical Memory Management

Con respecto al manejo de la memoria, el usuario cuenta con dos system calls posibles, **malloc** y **free**. **Malloc** es utilizada para reservar espacio de memoria mientras que *free* se usa para liberar memoria concedida.

Optamos por implementar la memoria física de la siguiente manera en el archivo encontrado en kernel llamado *'memorymanager.c'*. Al comenzar la ejecución, el kernel pide una cantidad específica de memoria, si este tamaño es mayor a una página (medida asignada por la consigna) se la divide en la cantidad de páginas que puede contener y se crea un bitmap para contabilizar las páginas en uso.

Cuando una tarea realiza la system call **malloc**, se pide la cantidad especificada con la función *allocate* y si el tamaño de memoria pedida es mayor a la disponible, no se hace la concesión. En caso de tener el espacio disponible, se le asigna la memoria al proceso que la pide y se marca en la variable usedMap la página que ahora está en uso.

En caso de que el usuario desee liberar la memoria y utilice la system call **free**. La función *deallocate* se encargará de encontrar el bit correspondiente en el usedMap y hacer el cambio a disponible.

Drivers

El keyboard driver consta de dos arreglos, uno representa el teclado simple y otro el teclado compuesto por shift y cualquier otra clave del teclado. En un buffer se van guardando los símbolos que ingresa el usuario.

Cuando se crea un proceso se le asigna una pantalla. Esta pantalla está implementada en *'screenLoader.c'*, cada una cuenta con el pid del proceso que la posee, lo que se escribió hasta el momento, la posición actual en la pantalla y las pantallas anteriores y posteriores.

Un proceso está en foco cuando se ve en pantalla lo que está escribiendo, es el único proceso que puede leer lo que escribe el usuario en pantalla. Para los demás procesos que escriban en pantalla, se guardará en cada una de sus pantallas asignadas lo que escribieron. Si se aprieta shift y las flechas atrás o adelante se cambia el proceso en foco, de esta forma podemos ver que escribe cada proceso y generar un input para ellos si es que lo necesitan para continuar su ejecución.

El driver del video cuenta con funciones más básicas como imprimir decimales, enters, tabs, caracteres, entre otros. También se encarga del cambio de las pantalla en foco ya que se debe backupear las pantallas de los procesos que poseen pantallas anteriores o posteriores a la que está en foco.

Procesos, Threads, Context Switching y Scheduling

Los procesos constan de una serie de variables individuales como: ppid (pid del padre), pid (el id del proceso), state (el estado del proceso, puede ser: *ready, running, blocked-io, blocked-mutex, blocked-pipe o dead*), un nombre descriptivo, una biblioteca de threads, entre otras. Todos los procesos tienen un hilo mínimamente, el hilo conductor.

Los hilos cuentan con un stack propio, un id y un estado. Los estados son los mismos que los que pueden tener los procesos.

Las system calls que puede realizar un usuario con respecto a los hilos y threads son:

- Si se desea crear un proceso, se utiliza **pcreate** pasándole como parámetro un *entry point*. El proceso es instantáneamente agregado al ciclo del scheduler.
- **pcreateBackground:** se crea un proceso en el *blackground*, es decir se crea un proceso y se lo agrega a la lista de procesos del scheduler sin perder el foco del proceso que estaba en foco.
- Si se desea finalizar el proceso en ejecución, se utiliza la syscall **exit**.
- Si se desea finaliza otro proceso que podría no ser necesariamente el que está en ejecución, se utiliza el **pkill**, pasando el pid correspondiente.
- **tcreate** es utilizado para crear un nuevo thread, se le deberá de pasar por parámetro el proceso al cual se desea que pertenezca y su *entry point* propio.
- para finalizar un thread particular de un proceso específico se utiliza la system call **tkill** pasándole por parámetros el pid del proceso y el id del thread.
- **getCurrentPid**, se utiliza para obtener el pid del proceso en ejecución.
- **ps:** lista todos los procesos del scheduler, con la cantidad de threads que tiene, su respectivo pid y su estado.

El scheduler se encuentra implementado en el archivo en kernel *'scheduler.c'*. Consta de un *Round Robin* con quantum 5 para los procesos y un *Round Robin* con quantum 1 para los threads. El mismo posee una cola de procesos, cada uno con su correspondiente librería de threads, un puntero al proceso actual, la cantidad de procesos que posee en su cola y el número de ticks hasta el momento (el número de ticks es reiniciado cada vez que se llega al quantum).

Se comienza con la función del *runScheduler*, en donde se contabilizan los ticks para mantenerse con el quantum de procesos deseado, al mismo tiempo se chequea la ejecución de los threads. Antes de retornar de esta función se remueven todos los procesos con el estado *dead* y se libera la memoria que estaban ocupando. Se actualiza el estado del proceso que estaba en ejecución, si este está *running* ahora está *ready* para la próxima vez que aparezca en la cola del scheduler para correr.

El scheduler ejecuta luego el próximo proceso de la cola que está en estado *ready*, pero no sin antes hacer el cambio de contexto que implica solo un cambio de stacks de

kernelStack al *userStack* del proceso en cuestión. Asimismo al finalizar el quantum asignado a un proceso se realiza un cambio de contexto de *userStack* a *kernelStack* y así sucesivamente entre las ejecuciones de los procesos.

Cuando el usuario utiliza el system call:

- **pcreate**: el scheduler crea un proceso y dentro de él crea un hilo con los parámetros dados. Luego lo agrega a la cola de procesos del scheduler en la función *addProcess* con el estado *ready*. Devuelta el pid del proceso que se creó.
- **pkill**: el scheduler recorre la lista de procesos, encuentra el deseado y cambia su estado a *dead* mediante *removeProcess* y luego será eliminado del ciclo del scheduler.
- **exit**: el scheduler llama a pkill con el pid del proceso que actualmente está corriendo.
- **tcreate**: el scheduler crea un hilo y lo agrega al proceso especificado *createThread*, que asigna la memoria necesaria y alista el *stackFrame*. Devuelve el id del thread que se creó.
- **tkill**: el escheduler recorre la lista de procesos hasta el encontrar el deseado y luego recorre la biblioteca hasta encontrar el thread pedido y cambia su estado a *dead* y luego será eliminado.

Si un thread es bloqueado ya sea por el read de IO, read de pipe o un mutex/semáforo, se actualiza el estado de dicho thread con el estado correspondiente perteneciente al listado mencionado anteriormente. Si todos los threads de un proceso están bloqueados, entonces se bloquea el proceso y mediante *yieldSwitch* se procede a ejecutar el próximo proceso sin importar que sobre quantum para el proceso anterior que ahora esta bloqueado. Si algún thread de un proceso es desbloqueado, entonces el proceso está desbloqueado automáticamente.

Mutex y Semáforos

En el archivo de *'mutex.c'* encontramos la implementación de los mutex. Cada mutex tiene un id, un estado (*locked* o *unlocked*) y una lista de los threads que tiene bloqueados, cada thread posee su id y pid.

Los usuarios cuentan con las siguientes system calls para manipular los mutex:

- **createMutex:** crea un mutex asignandole memoria, un id, lo inicializa en *unlock* y lo agrega a la lista de mutex. Devuelve el id del mutex que se creó.
- **endMutex**: recibe por parámetro el id del mutex y lo elimina de la lista.
- **mutexUp**: recibe por parámetro el id del mutex y desbloquea todos los threads que se encuentren en la lista del mutex.
- **mutexDown**:recibe por parámetro el id del mutex y agrega a la cola de threads bloqueados el current threads que hizo down, luego hace *yield* para que el scheduler pase al siguiente thread porque el actual ya está bloqueado por el mutex.

En el archivo de *'semaphores.c'* encontramos la implementación de los semáforos. La misma tiene fuertes similitudes con la implementación de los mutex. La diferencia principal es que cuando se inicia el semáforo se debe especificar la cantidad de procesos que pueden utilizarlo sin bloquearse. A medida que más procesos hacen *down* sobre el semáforo, su valor baja en uno. Cuando se llega a cero se comienzan a bloquear los procesos que accedan de ahora en adelante y se agregan a la lista de threads bloqueados en el semáforo correspondiente.

Los semáforos cuentan con las siguiente system calls:

- **createSemaphore**: recibe la cantidad de procesos que pueden utilizar el semáforo sin ser bloqueado. Devuelve el id del semáforo que se creó.
- **endSemaphore:** recibe el id del semáforo que se borrar y se lo elimina de las lista de semáforos.
- **semaphoreUp:** aumenta en uno el valor del semáforo.
- **semaphoreDown:** decrementa el valor en uno, si el valor comienza a ser negativo entonces se agrega a la lista de threads bloqueados el thread que le hizo *down.*

IPC’s

En el archivo *‘pipefs.c’* se encuentra la implementación del envío y recepción de mensajes con recepción bloqueante. Para cada mensaje que se desea enviar de un proceso a otro, se crea un pipe con un nombre compuesto de ambos pids y dos punteros a char uno para write y otro para read.

El usuario consta de tres system calls para manipular la mensajería:

- **createPipe**: esta system call recibe dos parámetros, el pid del proceso que quiere mandar un mensaje y pid del proceso que lo va recibir. Devuelve el nombre compuesto por los pid (el nombre del pipe).
- **send:** recibe por parámetro el nombre del pipe y el mensaje. Se escribe en el buffer del pipe el mensaje para que luego el proceso receptor lo lea.
- **receive:** recibe por parámetro el nombre del pipe y el pid del proceso del cual espera el mensaje. Si no tiene ningún mensaje, entonces el proceso que está en ejecución se bloquea hasta que reciba el mensaje que espera.

Philosophers:

Realizamos la implementación de los philosophers en *‘shell.c’* para probar el buen funcionamiento de los mutex. Utilizamos un mutex y un thread por cada filósofo. Las siguientes system calls fueron utilizadas: **createMutex, mutexDown, mutexUp, tcreate.**

Producer-Consumer:

Para esta funcionalidad también localizada en *‘shell.c’* utilizamos un mutex y dos semáforos. Los semáforos fueron utilizados para determinar cuando puede producir el productor y cuando puede consumir el consumidor. Se usó un hilo para el consumidor y otro para el productor. Los system calls usados fueron: **createMutex, mutexDown, mutexUp, tcreate, createSemaphore, semaphoreUp, semaphoreDown.**