Sistemas Operativos

Nachos: Procesos y Scheduler

Octubre 2006

Profesor Jonathan Makuc // jmakuc@gmail.com

Índice

1. Introducción

1.1. Generalidades

A continuación se describe la forma de trabajo con los procesos de NachOS y la estructura del scheduler del sistema.

1.2. Scheduler

El scheduler es el componente del sistema operativo que determina que proceso debe entrar a la CPU para ser procesado. En forma estricta existen 3 niveles de scheduling:

- Corto Plazo: calendarización de procesos v/s CPU
- Mediano Plazo: calendarización sobre que procesos van a swap y cuales permanecen en memoria
- Largo Plazo: calendarización de que procesos a admitir en el sistema para ser procesados.

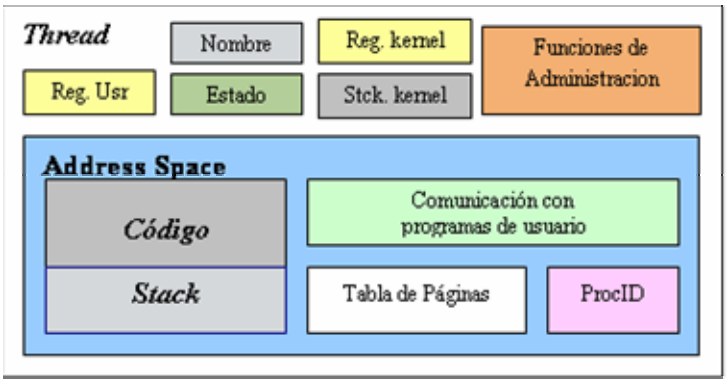
NachOS solo trae implementado scheduler de corto plazo.

2. Procesos en NachOS

2.1. Generalidades

Como se explica en el documento 1 de NachOS, los procesos están compuestos por 2 clases.

Por una parte se tiene el AddressSpace que contiene la parte dura de un proceso (código, stack, etc). Por otra se tiene Thread que se encarga de las labores administrativas para el manejo de procesos, que funciona como cáscara al AddressSpace al contenerlo.



2.2. Stacks y registros

Nachos trabaja con 2 grupos de registros por cada proceso. Uno colocado en la hebra (Thread) contiene el set de registros de la CPU correspondientes al programa de usuario; estos contienen los valores que este esta utilizando en la CPU como el program counter, stacktop, etc; los cuales deben ser almacenados al ocurrir un cambio de contexto para poder luego retomar la ejecución donde se dejo. Por otra parte se tiene el set de registros de CPU del lado del kernel, que almacenan los valores al momento que se ejecuta código del lado del kernel en la CPU. En un sistema operativo real estos registros también deben guardarse al producirse un cambio de contexto para poder retomar la ejecución en su punto anterior; pero debido a la arquitectura de NachOS, todo el código del lado del kernel NO corre en el MIPS simulado sino en la CPU real, por lo tanto es el Linux quien controla la ejecución de este código no siendo necesario el almacenamiento de estos registros por parte de NachOS.

2.3. Creación de Procesos

2.3.1. Proceso Inicial de NachOS

Como todo sistema operativo, NachOS levanta un proceso inicial al momento de partir desde el cual se podrán crear nuevos procesos. Este procedimiento se encuentra en el mismo método ***main*** en ***thread/main.cc.*** Una vez concluidos el parseo de la línea de comando y la inicialización del kernel, se siguen los siguientes pasos, si es que se ha proporcionado un nombre de archivo para ejecutar.

1. Creación del espacio de direcciones (cargado de código a memoria)
2. Inicialización de registros de lado de kernel. Se setea el Program Counter a cero y el limite del stack.
3. Se hecha a correr la maquina MIPS simulada: kernel->machina->Run();
4. El método Run es un ciclo infinito de 2 instrucciones: OneInstruction y OneTick. La primera (implementada en mipssim.cc) procesa 1 instrucción en assembler del código del programa de usuario y avanza el program counter. La segunda (implementada en interrupt.cc) avanza 1 tick de reloj en la máquina simulada, haciendo que se revise la lista de interrupciones pendientes. Aquí específicamente se busca por ***timeslice***; de existir uno pendiente, se representa seteando la variable ***yieldOnReturn*** a true provocando la salida del proceso actual de la CPU llamando a ***kernel->currentThread->Yield()***.

2.3.2. Syscall exec

La syscall exec realiza los siguientes pasos en la creación de un proceso. Su detalle se encuentra en la función ***ExecHandler*** en ***exception.cc.***

1. Recuperación del nombre de archivo: a ejecutar desde espacio de usuario.
2. Obtención de PID: desde la tabla de procesos del kernel.
3. Creación del AddressSpace: Detallado en el documento 4.
4. Creación de la hebra: Constructor Thread.
5. Llamado a la función Fork: esta función recibe como parámetro el puntero a la función que se debe ejecutar y agrega la hebra a la readyList del scheduler. Por defecto Nachos llama a la

función de inicialización *ProcessStartup* que inicializa los registros de la maquina simulada del lado del usuario y hecha a correr la presente hebra, siguiendo el paso 3 en delante de 2.3.1.

2.4. Control de procesos

Los procesos en NachOS son controlados a través de los métodos que provee la clase Thread. A continuación se detalla el funcionamiento de cada uno de ellos.

Thread	Constructor recibe el nombre que tendrá la presente hebra. Inicializa los parámetros para luego llamar a Fork con la función a correr.
~Thread	Libera el stack
Fork	Llama a stackAllocate entregando el puntero a la funciona a ejecutar una vez iniciado el proceso. Añade la hebra al final de la readyList del scheduler.
Yield	Provoca que la hebra actual ceda la CPU al siguiente proceso en la readyList. Si no existe otro proceso listo para correr, sigue ejecutándose la misma hebra.
Sleep	Provoca la salida de la hebra actual de la CPU por término del programa o bloqueada (I/O o sincronizacion). Si no existe ningún otro proceso listo, llama al proceso IDLE hasta que ocurre la siguiente interrupción I/O provocando la entrada a la readyList de un proceso. Recibe como parámetro un flag que indica si poner a dormir el proceso para siempre (destruirlo) o hasta su siguiente turno.
Begin	Dealoca la hebra anterior (si corresponde) y habilita las interrupciones. Este método es llamado justo antes de *ProcessStartup* que inicializa los registros y pone a correr la hebra.
Finish	Llamado luego de que el programa ejecuta su última instrucción, provoca la salida del proceso de la CPU y su destrucción.
StackAllocate	Aloca un espacio de memoria contiguo para el stack y setea los valores de los registros de inicialización para la hebra.
SaveUserState	Guarda los registros de la CPU del lado de usuario.
RestoreUserState	Restaura los registros de la CPU del lado de usuario.

2.5. Destrucción de procesos

La destrucción de un proceso se realiza mediante el método sleep de la hebra pasando TRUE como parámetro.

1. Puesta a dormir para siempre: kernel->currentThread->Sleep(true);
2. Siguiente Hebra a correr: kernel->scheduler->Run(nextThread, true)
3. Cambio de contexto: antes de realizarse el cambio de contexto se setea el puntero ***toBeDestroyed*** a la hebra actual.
4. Ocurre el cambio de contexto (SWITCH)
5. Al retornar del cambio de contexto se verifica si esta hebra, la llamada en 1), debe ser destruida en el método checkToBeDestroyed(), quien borra el objeto de la hebra.

2.6. Sincronización de Procesos

Nachos trae implementadas las siguientes primitivas de sincronización en synch.{h,cc}:

Clase Semaphore (Semáforo): implementación fiel de los semáforos de Dijkstra, con los métodos P() – down o wait – y V() – up o signal – contiene una lista enlazada de las hebras que están bloqueadas esperando.

Clase Lock: mutex simple con los métodos Acquire y Release. Implementado con un semáforo en su interior, la llamada Acquire bloquea si no esta disponible.

Clase Condition (Variable Condicion): Implementadas también con semáforos, poseen los métodos Wait, Signal y Broadcast.

Sin embargo NachOS no trae programado las ***syscall*** para que los programas de usuario puedan hacer uso de estas clases. Una buena idea es colocar listas enlazadas de semáforos, o la primitiva que más le acomode, en el kernel y manejar las instancias con 3 syscalls: crearSemaforo(nombre, valorInicial), P(nombre), V(nombre).

3. Scheduler en NachOS

3.1. Generalidades

Tomando el esquema de calendarización en 3 niveles (largo plazo – admisión, media plazo – swap y corto plazo – CPU), NachOS solo implementa scheduling de CPU o corto plazo.

3.2. Scheduler de CPU

3.2.1. Round Robin

El scheduler de CPU de NachOS es expropiativo Round Robin. Este lo implementa via una lista enlazada de Hebras privada al interior de la clase Scheduler: ***readyList***.

3.2.2. Métodos

3.2.2.1. *ReadyToRun*

Entrega al scheduler una hebra que esta lista para corre y obtener así tiempo de CPU. En la implementación original encola a la hebra al final de la lista.

3.2.2.2. *FindNextToRun*

Método que retorna la siguiente hebra que tiene que entrar a la CPU. En la implementación original, la función remueve el proceso que se encuentra al inicio de la cola o nulo en caso de estar vacía.

3.2.2.3. *Run*

Pone a correr a la hebra pasada como parámetro. Este método siempre es llamado cuando se trata del segundo proceso en adelante, ya que realiza cambio de contexto (descrito en 3.4). Recibe además un booleano que de estar seteado en TRUE indica que la hebra (proceso) saliente debe ser destruido.

3.3. Time Slice

La implementación de la interrupción Timeslice es realizada por 2 componentes en NachOS. Por una parte se tiene la variable ***yieldOnReturn*** de la clase Interrupt que provoca la llamada ***Yield*** a la hebra que este en la CPU, como se vio en 2.3.1.

Quien realiza el seteo de esa variable a TRUE para provocar el Timeslice es la clase ***Alarm*** (thread/alarm.{h,cc}). Esta extiende a ***CallBackObj*** implementando en el método ***callback*** la generación del Timeslice. La ocurrencia de la llamada a ***callback*** es manejada por el ***timer*** interno de ***Alarm***, el cual puede ser inicializado para provocar los timeslices aleatoreamente (por defecto) o a un intervalo regular.

3.4. Cambio de contexto

El cambio de contexto en NachOS se produce cuando se llama al método ***Yield*** de una hebra (por timeslice o por el usuario) y existe otro proceso listo para correr en el scheduler. De no existir proceso listo para ejecutarse, continuará ejecutándose la hebra actual.

En el método Yield se busca la siguiente hebra a correr llamando a FindNextToRun del scheduler. De retornar un valor no nulo (existe siguiente proceso) la hebra se encola a si misma en los procesos listos para correr antes de provocar su salida de la CPU.

El cambio de contexto se realiza la función Run del scheduler. A continuación se analiza línea a línea el proceso completo.

La hebra actual será la hebra antigua.

Debemos asegurarnos de que las interrupciones de MIPS estén apagadas para realizar el cambio de contexto en un solo paso.

Si finishing es TRUE, entonces la hebra actual será destruida.

```
void
Scheduler::Run (Thread *nextThread, bool finishing)
{
    Thread *oldThread = kernel->currentThread;

    ASSERT(kernel->interrupt->getLevel() == IntOff);

    if (finishing) {
        ASSERT(toBeDestroyed == NULL);
        toBeDestroyed = oldThread;
    }
```

Se almacenen los registros del espacio de usuario

```
    if (oldThread->space != NULL) {
        oldThread->SaveUserState();
        oldThread->space->SaveState();
    }
```

Se verifica si la hebra antigua tiene algún overflow no detectado hasta este punto

```
    oldThread->CheckOverflow();
```

Se cambia el puntero de "hebra actual" del kernel a la nueva hebra y se cambia si estado

```
    kernel->currentThread = nextThread;
    nextThread->setStatus(RUNNING);

    DEBUG(dbgThread, "Switching from: " << oldThread->getName()
<< " to: " << nextThread->getName());
```

CAMBIO DE LA CPU A LA OTRA HEBRA

```
    SWITCH(oldThread, nextThread);
```

En este punto hemos retornado a la hebra antigua.
(Ver explicación más abajo)

Interrupciones apagadas para terminar el cambio atómicamente

```
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    DEBUG(dbgThread, "Now in thread: " << oldThread->getName());
```

Checkeo si esta hebra debe ser destruida

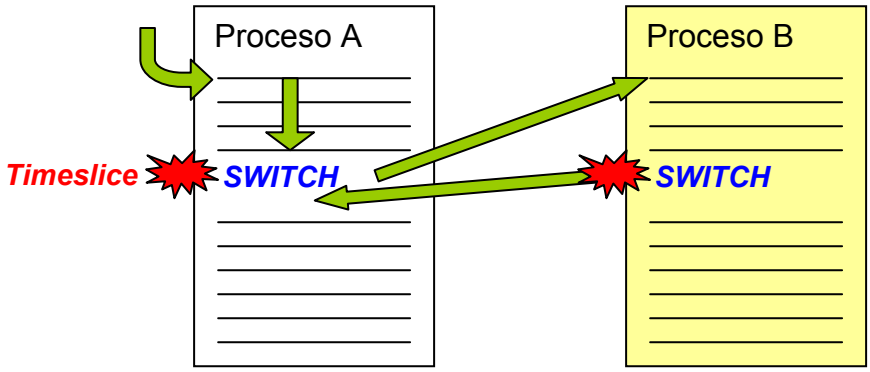
```
    CheckToBeDestroyed();
```

Si se llega a este punto, no se destruyo la hebra. Se restauran los registros de la CPU y se esta listo para reanudar la ejecución

```
    if (oldThread->space != NULL) {
        oldThread->RestoreUserState();
        oldThread->space->RestoreState();
    }
}
```

El cambio de contexto es un proceso a menudo confuso y no tan fácil de comprender. Una forma clara de explicar que ha sucedido en el código anterior es mediante el siguiente ejemplo.

Téngase 2 procesos A y B en NachOS ya instanciados en memoria con su código de programa de usuario.



Se comienza a ejecutar el Proceso A y existe un proceso B listo para ser ejecutado. El registro program counter entonces esta en cero y se comienza a leer desde la primera instrucción. En un punto de la ejecución se produce un ***timeslice*** que determina que debe entrar el siguiente proceso. Esta interrupción provoca un ***trap*** al sistema (paso desde modo de usuario a modo kernel). Hasta aquí seguimos con la CPU destinada al proceso A, pero en ejecutando código del kernel. Se realiza todo el proceso descrito anteriormente que desemboca en la llamada a **SWITCH** por parte del Proceso A para que se pase al Proceso B.

De la misma forma que A, B comienza a ejecutarse hasta que ocurre otro ***timeslice***. Aquí, análogamente a lo ocurrido en A, se produce un ***trap*** al sistema, pasando a modo kernel y la posterior llamada al método **SWITCH** por parte del Proceso B para pasar al Proceso A.

Dado que la última instrucción que se ejecuto cuando estábamos en el Proceso A fue **SWITCH**, entonces ahora que hemos retornado se debe ejecutar la siguiente instrucción (marcada en el código con ▶). Durante todo el periodo de ejecución del Proceso B, A no se entera de lo que ha sucedido ni puede interceder para que sea devuelto a la CPU. Funciona como una suerte de “Dimensiones Paralelas”.

Sin embargo, ya que en la llamada Yield la hebra se coloca a si misma en los procesos listos para correr del scheduler entonces llegará el momento en que se elija a esta hebra para entrar nuevamente a la CPU.