Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Licenciatura en Ciencias de la Computación

Arquitectura del computador

Trabajo Práctico final
Corrutinas avanzadas

Sotelo, Bruno

18 / 12 / 2017

1 Introducción

El scheduler implementado en el trabajo es del tipo Multilevel Feedback Queue, se utilizan varias colas donde cada una posee una cantidad de quantums diferente. Este tipo de scheduler ha demostrado funcionar muy bien en sistemas en general ya que permite separar fácilmente a las tareas que hacen gran uso del procesador y las que se centran en I/O. Al contrario del Multilevel Queue Scheduler, donde a las tareas se les asigna una prioridad y son insertadas en la cola con la prioridad correspondiente, en este scheduler las tareas pueden cambiar su prioridad, dando lugar a un mayor dinamismo ante cambios de comportamiento de las mismas, y evitando también starvations en las colas de más baja prioridad. Para facilitar su uso se implementó una interfaz similar a la de los Pthreads de C, por ejemplo: las funciones que serán alimentadas al scheduler tienen como tipo **void *f(void *)**, para activarlas se utiliza una función **create_task**, y una tarea puede esperar a que otra termine con **join_task**.
En cuanto a la comunicación entre tareas, se eligió el esquema de shared memory. La interfaz de I/O con este IPC es muy similar a la de un archivo, cada tarea puede crear una o varias conexiones con un segmento de memoria compartida y poseerá un puntero a esta en cada una de las conexiones. El puntero puede moverse libremente en el área escrita de un segmento, pero se encuentra la restricción de escribir secuencialmente (es decir, la parte escrita del área será una sola, sin espacios no escritos en el medio).
Para realizar los locks también se usó una interfaz similar a los locks de Pthreads, aunque muy simplificada. Se puede crear, bloquear, desbloquear y eliminar un lock.

2 Scheduler

Como se dijo al principio, el scheduler utiliza múltiples colas de tareas. Como además cada cola representa una prioridad de ejecución, una buena forma de crearlas y mantenerlas es la estructura de **cola de prioridad**. Esta se encuentra implementada en los archivos *pqueue.c* y *pqueue.h*. Es una implementación mínima pero suficiente para este scheduler, en ella se puede: insertar un nuevo nodo con la prioridad más alta (*pqueue_new_node*), reinsertar un nodo con una prioridad cualquiera (*pqueue_insert*), quitar y poseer el primer elemento de la cola (*pqueue_pop*) y mover todos los elementos a la mayor prioridad (*pqueue_lift*, una función específica para este scheduler). Esta implementación es bastante flexible y simple para ayudar a mantener así también el código del scheduler.
Para comenzar a utilizarlo, primero se deberá reservar espacio para una estructura Task. Esta contiene toda la información necesaria para administrar la ejecución de la tarea:

- Un buffer sobre el cual se ejecuta *setjmp* y *longjmp* para controlar el hilo de ejecución,
- Una variable *Task_State* que mantiene el estado actual de la tarea. Se consideran en este trabajo cuatro estados: activa (la tarea está utilizando el procesador), lista (desea utilizar el procesador pero debe esperar su turno), bloqueada (se encuentra a la espera de ser desbloqueada por alguna condición) y zombie (la tarea ya terminó y no utilizará más el procesador, se mantiene en este estado para conocer su resultado, y saber que terminó).
- El argumento que recibe la tarea al momento de ser ejecutada (le es entregado a través de *create_task*).
- Un puntero a la función que ejecuta la tarea.
- Un puntero al resultado de la función ejecutada (será NULL si no termina).
- Un bit (*queued*) que indica si la función se encuentra en alguna cola de espera del scheduler o no.
- Posición inicial del stack de memoria. Cada tarea posee un fragmento de stack sobre el cual puede actuar libremente. El tamaño de uno de estos fragmentos está parametrizado por una macro en *mm.h*.

Una vez reservada una estructura *Task*, se debe llamar a la función *start_sched*, que recibe como argumento un puntero a *Task*, una tarea tratada especialmente porque representa el hilo de ejecución principal del programa creado. En la función se realizan varios ajustes y creaciones de variables necesarias para poner el scheduler en marcha. Primero se setea el manejo de señales del programa. La función *scheduler*, definida también en *scheduler.c*, actuará como handler de las señales RT *SIG_TASK_YIELD, SIG_TASK_NEW, SIG_TASK_END*, las cuales indicarán a este handler que la tarea activa le "devuelve" el procesador, que se crea una nueva tarea y que la tarea activa termina y no se le debe dar el procesador nuevamente, respectivamente. Se utilizan señales real-time ya que poseen más beneficios que las señales de POSIX-1. Luego se crean tres timers:

sched_yield_timer Que controlará el tiempo de ejecución que se le cede a una tarea; el scheduler pone en marcha este timer antes de poner la tarea en ejecución, y al llegar el timer a 0 mandará una señal *SIG_TASK_YIELD* que lo pondrá de nuevo en marcha para continuar otra tarea.

sched_lift_timer Que indica cada cuánto tiempo se llevan todas las tareas a la cola de mayor prioridad, lo cual evita el starving en las colas de menor prioridad. Está parametrizado por *SCHED_LIFT_SECS* y *SCHED_LIFT_NSECS*. Este timer no manda señales sino que es revisado en cada ejecución del scheduler.

sched_idletask_timer Especifica el tiempo que toma la tarea 'idle' cuando es ejecutada. Es una tarea ociosa, fue definida para los casos en que no quedan tareas por ejecutar en cola. Esta tarea se ejecutará por *SCHED_IDLETASK_NSECS* segundos.

Lo siguiente en *start_sched* es setear los valores de la estructura *maintask*. El miembro *mem_position* es iniciado con la función *take_stack*, esta es en realidad una macro definida que abstrae a *memory_manager*, la cual controla los segmentos de stack que posee cada tarea. Puede entregar a una nueva tarea un puntero al próximo segmento libre (que ninguna otra está usando) o una tarea puede indicarle que no utilizará más su segmento con la macro *release_stack*, y el manager lo marcará como libre para cederlo a otra tarea. El tamaño de uno de estos segmentos está parametrizado con la macro *MEM_TASK_SIZE*
Lo que sigue es crear la cola de tareas del scheduler. También está parametrizada la cantidad de colas (prioridades) que se utilizarán con la macro *QUEUE_NUMBER*. Observar que puede crearse un scheduler Round Robin si se define $QUEUE\_NUMBER = 1$. Finalmente se pone un checkpoint en el lugar con la macro *YIELD*, básicamente una llamada a *setjmp*, y se llama al scheduler.
El funcionamiento del scheduler en general es como sigue:

1. El primer paso es desencolar la próxima tarea a ejecutar. Si primero se encola la tarea que se estaba ejecutando, es probable que esa tarea tenga una alta prioridad, y vuelva a ser elegida por el scheduler, lo que llevaría a un starving de las demás tareas. Esto hace necesaria la existencia de la tarea 'idle', ya que puede suceder que no haya otra tarea en cola, y no habría nada que ejecutar luego. Cuando se desencola una tarea, se debe asegurar que no esté bloqueada. Luego se explicará el mecanismo de bloqueo.

2. Si la señal que llega no es *SIG_TASK_END* y *qelem_current* no es *NULL*, la tarea que estaba en ejecución se guardará nuevamente en la cola de tareas listas y se hará un checkpoint (*YIELD*) al cual volver cuando el scheduler elija otra vez esta tarea, y que implica retornar del handler para continuar la ejecución que se estaba dando. Observar que la segunda condición será falsa sólo en la primera llamada a *scheduler*, y al finalizar una ejecución de la tarea 'idle'. En cuanto a la primera, su razón es que esa señal sólo será enviada por una tarea al terminar a través de la función *stop_task*, con la macro *FINALIZE*, por lo cual no se le debe entregar otra vez el procesador, y no se debe encolar.

3. Antes de encolar la tarea que estaba en ejecución se decide en qué nivel de prioridad se lo hace. Si la tarea utilizó todo el tiempo dado normalmente, se bajará en 1 su prioridad, ya

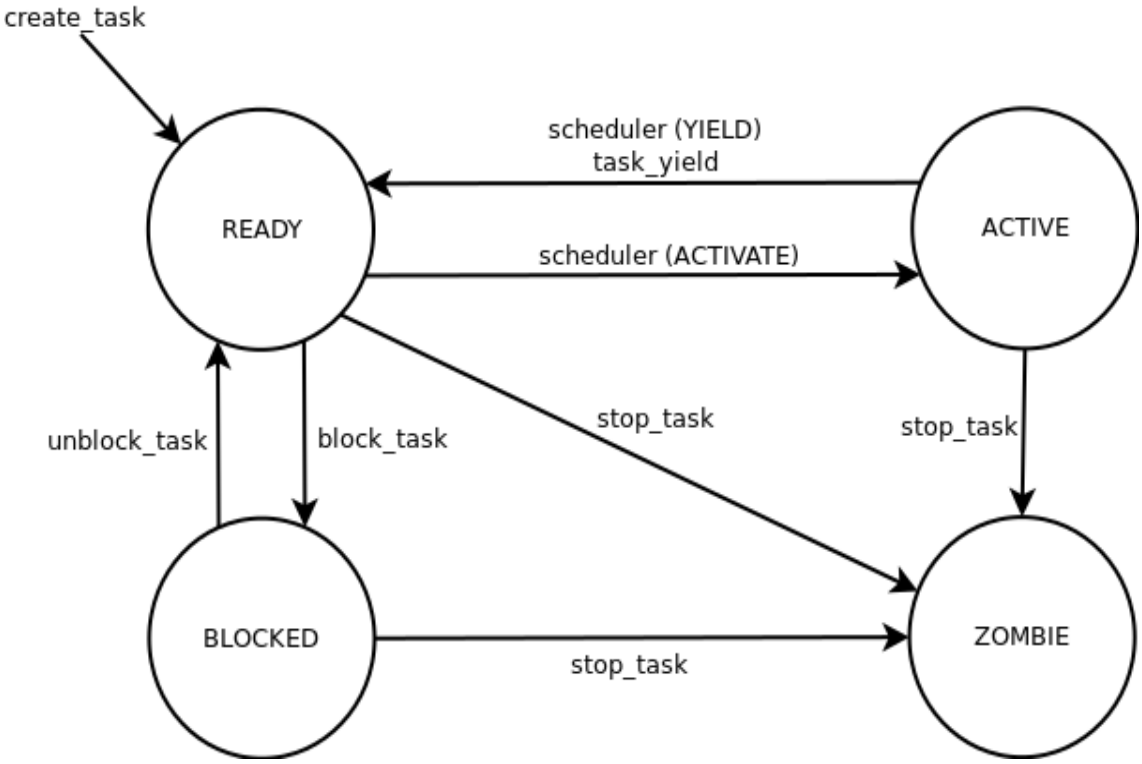
que es probable que sea una tarea que usará más tiempo de CPU. En cambio, si la tarea utilizó *task_yield*, que devuelve el control al scheduler antes de finalizar su segmento de tiempo, tiene que ser porque se encuentra esperando I/O, un lock, o que alguna otra condición se cumpla para seguir su ejecución, entonces se deja a la tarea en su prioridad actual.

4. Si la señal que llegó es *SIG_TASK_NEW*, se deberá introducir, al final de la cola de espera de mayor priorida, la nueva tarea. Su puntero (*Task **) se pasa como argumento en *data -¿ si_value.sival_ptr*.

5. Cada cierto tiempo es prudente subir la prioridad de todas las tareas en la cola de espera a la más alta. Esto permite evitar starvations y mejora la dinámica del scheduler, haciendo que se adapte mejor a los comportamientos de las tareas. La acción se encuentra implementada como parte de la estructura *pqueue* con la función *queue_lift*.

6. Si la tarea desencolada es un puntero *NULL*, es porque la cola estaba vacía. En este caso se activará la tarea 'idle'.

7. Finalmente se configura el estado de la tarea elegida en *ACTIVE*, se calcula la cantidad de quantums que podrá usar para correr, se activa el timer con ese valor y se activa la tarea. Los tiempos de ejecución están parametrizados con las macros *TICK*, que es la duración en milisegundos de un quantum, y con *QUANTUM(x)* se puede especificar la cantidad de quantums que obtiene una tarea de nivel x. En este caso la función de quantums es, como es recomendado, exponencial en función del nivel de tarea.



Flujo de estado de las tareas

Sobre el bloqueo de tareas: Si una tarea necesita bloquear a otra debe utilizar la función *block_task*, la cual, dado un puntero a la tarea a bloquear, simplemente cambia el estado de esa tarea a *BLOCKED*, siempre que el estado anterior no sea *ZOMBIE*. La tarea eventualmente

será desencolada para ponerla en ejecución. Cuando esto sucede, si la tarea tiene estado *BLOCKED*, procederá a eliminar el qelem correspondiente (no la tarea) y a desencolar otra tarea que no esté bloqueada.
Para desbloquear una tarea se utiliza *unblock_task*. La función le devuelve el estado *READY* y diferencia entre dos casos: la tarea está en la cola de espera o no; si lo está, no necesita hacer nada más, el scheduler le entregará el procesador como si nada hubiera sucedido; si no lo está, es necesario crear un nuevo elemento en la cola que contenga a esta tarea, el scheduler lo hará al enviársela como si fuera una recién creada, y se la coloca con prioridad 0 en la cola.

3 Locks

Los locks se encuentran implementados en *locks.c*, en forma de spinlocks (por lo cual se los nombra *sp*). Los lock tienen por tipo *task_sp_t*, un sinónimo para *long long*, y pueden tener dos valores, *TASK_SP_LOCK* o *TASK_SP_UNLOCK*. Para crear un lock simplemente se declara una variable *task_sp_t* con el valor *TASK_SP_INIT*. Una tarea pide un lock con *task_sp_lock* pasando el puntero a la variable declarada, y luego lo suelta llamando a *task_sp_unlock*.
El algoritmo para adquirir un lock se basa en la instrucción de x86_64 *xchg*, la cual intercambia dos variables atómicamente. En la función para adquirir el lock, primero se declara una variable con valor *TASK_SP_LOCK*, para intercambiarla luego con el lock argumento, mediante *xchg*. Entonces se prueba el valor de la variable local, si sigue siendo *TASK_SP_LOCK* es porque el lock está tomado, entonces se suelta el procesador para volver a probar suerte en la próxima ejecución; si en cambio el valor ahora es *TASK_SP_UNLOCK*, es porque se adquirió el lock, entonces se puede empezar a trabajar en la sección crítica. Para soltar un lock, sólo se le debe devolver el valor *TASK_SP_UNLOCK*.
Este tipo de lock sufre algunas desventajas:

- Pueden darse casos de starvation cuando una tarea no llega a tomar un lock. Esto se debe a que no existe un orden al pedirlo, las tareas se encuentran en condición de carrera.
- Si una tarea mantiene el lock por un tiempo prolongado, se genera una pérdida de rendimiento debido a la gran cantidad de cambios de contextos innecesarios generados por la espera activa en la toma del lock.

Ambos problemas pueden ser solucionados agregando colas a la implementación de estos locks. Si una tarea no consigue el lock al intercambiar variables, no seguirá intentando conseguirlo activamente, sino que pasará a bloquearse y esperar en una cola hasta que el actual dueño del lock lo suelte, y llame al primero en la cola. Esto se encuentra implementado en los archivos *locks2.c* y *locks2.h*. Se debe mencionar que tampoco son infalibles ya que generan también una caída de rendimiento cuando los tiempos de retención de locks son muy cortos.

4 Comunicación

La comunicación entre procesos se realiza mediante shared memory, y está implementada en *shmem.c*. Para comenzar a utilizar un área o región de memoria compartida, la tarea debe iniciar una conexión usando *shmem_new* y dando el nombre del área (un caracter, lo cual limita el número de áreas a 127). Esta función es una macro que abstrae al manejador de memoria compartida *shmem_region_manager*, el cual puede crear o remover regiones. *shmem_new* devuelve una estructura *shmem_t*, que será utilizada como argumento para las funciones de interfaz de shared memory.
El tamaño de una región de memoria compartida está parametrizado por la macro *SHMEM_REGION_SIZE*. Aquí se utilizan 255 bytes por región de memoria.
La estructura *shmem_t* contiene un puntero (*) a la región de memoria compartida y un entero que representa el puntero de la tarea sobre esa región. Cada región puede ser accedida por cualquier cantidad de tareas, y cada tarea debería iniciar su propia conexión a la región, ya que

así tendrá su puntero sobre ella. Este puntero funciona de manera similar al que utilizan los archivos: si se leen 5 bytes de la región, el puntero sumará 5 para que la tarea continue leyendo desde el punto en que paró. Si se escriben 5 bytes, se los escribe a partir de la posición actual del puntero, y este se deja en el próximo byte a escribir. También se puede cambiar la posición del puntero con la función *shmem_seek*; esta toma como argumentos un *shmem_t **, un *unsigned char* que es la posición de puntero deseada, y un *char* que puede ser:

- *SEEK_ABS*, entonces la posición de puntero indicada es absoluta en la región (se indexa desde 0).
- *SEEK_OFFSET*, con el cual la posición indicada es relativa a la posición actual del puntero (se suma el argumento a la posición).

La nueva posición del puntero a la región debe estar dentro del límite escrito (para facilitar el control, la escritura se "serializa"), si lo está, se cambia su valor y se devuelve *SHMEM_OK*, de lo contrario es devuelto *SHMEM_INVALID*.
Las otras funciones que provee la interfaz de shared memory son:

- *shmem_read* que dado el puntero a la estructura *shmem_t*, un puntero a un buffer b y un *unsigned char* n, lee de la región n bytes a partir de la posición actual del puntero, y los guarda en b. Si no existen n bytes para leer, se lee todo lo posible, y se devuelve la cantidad de bytes leídos.
- *shmem_write*, como read, toma el puntero a la región de memoria, un puntero a un buffer b y un *unsigned char* n, y escribe n bytes de b en la memoria compartida. Si no hay espacio para escribir n bytes, escribe todo lo posible, y devuelve el número de bytes escritos.
- *shmem_poll* toma el puntero a *shmem_t* y devuelve *SHMEM_OK* si hay algo escrito más allá del puntero actual de la tarea, o *SHMEM_ INVALID* si no lo hay.
- *shmem_block_read* es una combinación entre *shmem_read* y *shmem_poll*; si no hay nada para leer en la región de memoria, se devuelve el control al CPU, para volver a consultar cuando se adquiera nuevamente. En cuanto haya algo para leer, se utiliza la función de lectura y se retorna.
- Observación: En todas las funciones mencionadas se chequea que la región aún esté activa antes de realizar cualquier cosa con la función *shmem_check*, si no lo está, se devuelve en todos los casos la bandera *SHMEM_CLOSED*.

Bibliografía consultada

- Andrew S. Tanenbaum. *Modern Operating Systems, third edition*. 2009.
- Gunnar Wolf, Esteban Ruiz, Federico Bergero, Erwin Meza. *Fundamentos de Sistemas Operativos, primera edición*. 2015.
- Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 2015.