Workshop 2 - Semaphores, list API and Usage

Team Cardinal
CSCI 3453 Spring 2017

March 3, 2017

1 Semaphores

1.1 Overview

Semaphores and locks are two examples of synchronization primitives, objects used to ensure that multiple threads can access shared data without introducing race conditions. Although locks are ubiquitous in general-purpose resource management, semaphores are still necessary to efficiently deal with the interrupt handler. PintOS synchronization is built entirely off of semaphores and its lists. To understand how PintOS synchronization routines can be extended to support priority scheduling, this report presents the necessary background and pseudocode to tackle the upcoming assignments.

1.2 Background - Architecture

Semaphores always have non-negative values, which controls how many threads can access the protected resource at a given time. A semaphore also has three atomic functions: down(), up(), and init(). No other operations are allowed on a semaphore. In particular, no thread can directly read the current value of the semaphore.

Calling down(), sometimes referenced as P(), on a semaphore is equivalent to asking for permission to access its protected resource. If the resource is available instantly (the semaphores value is positive), down() will decrement the value and return. A semaphore value of 0 signifies that the resource is unavailable. In this case, the thread calling down() will be added to a list of threads waiting for the semaphore, and then block until the resource is available.

Calling up(), sometimes referenced as V(), on a Semaphore is equivalent to forfeiting access to a resource. When a thread calls up(), it will increment the value of the semaphore and, if there are threads waiting, unblock the first thread on the waiting list.

The init() function initializes a semaphore with the non-negative integer value. This value will control how many threads can down() the semaphore before a calling thread will need to block for access. For example, say a semaphore is initialized with a value of 2. When Thread 1 and Thread 2 call down(), the calls will return immediately because they decrement the count from 2-1 and 1-0 respectively. However, if Thread 3 calls down() before Thread 1 or Thread 2 call up(), it will be added to the waiting list and block until it is woken by a call to up(). In this way, A semaphore can be used to control access to the number of threads allowed to access a resource concurrently.

1.3 Uses

1.3.1 Mutual Exclusion

Mutual Exclusion (Locks) - Semaphores initialized with a value of 1 will function as a lock, since only one thread will be allowed access at a time. In Pintos, Locks are implemented as a structure containing a Semaphore and a pointer to the owning thread. See Appendix A for an example usage of mutual exclusion.

1.3.2 General Waiting

General Waiting - Semaphores are also utilized for general waiting purposes, such as waiting for another thread to execute a conditional look. In this case, the semaphore is initialized to 0 and each call to down() waits for a corresponding thread to call up(). (If up() is called first, down() returns immediately.) See Appendix B for and example usage of general waiting.

1.4 Pintos Implementation

In PintOS, semaphores are internally built out of disabling interrupt and thread blocking/unblocking. Each semaphore maintains a list of waiting threads - this is implemented via linked list in /lib/kernel/list.c. Thread and lock structures, themselves, are implemented in /src/threads/synch.h.

1.5 Limitations of Semaphores

Like condition variables, semaphores can be used to wait for a thread to finish a process. For example, if a thread wants to create another thread, and wait until the created thread has finished some task, the first thread may initialize a semaphore to 0, pass it to the created thread, and call down() immediately after. The value of the semaphore will be zero, so the first thread will block until the next call to up(). When the created thread has finished its task, it should call up() to wake the first thread. In this way, semaphores can be used like condition variables. However, semaphores cannot replace condition variables because they cannot test any conditions other than the current value of the semaphore. With the Lock and Condition Variable model, threads can test a predicate value, and then continue to wait if the predicate is not true. This is possible because a call to wait() will release the held lock before stopping the thread. When signal() is called, wait() will reacquire the lock, which allows access to shared state.

For this reason, Locks and Condition Variables are preferred to Semaphores. Pintos implements Condition Variables through Semaphores. However, because Condition Variables involve blocking the thread until a value becomes true, they cannot be used inside interrupt handlers like Semaphores can.

1.6 Priority Scheduling Using Semaphores

The priority scheduling will be important in determining which threads should occupy the CPU. Priority introduces a layer of complexity: how can threads be inserted into the waiting

list? The priority (either dictated by the thread itself or otherwise) will control the location within the waiting list; this location (priority) will be reflected in this absolute order. This can be achieved either by reordering the list as a whole or maintaining the appropriate order by inserting the thread in the correct place in the already ordered list. The semaphore (lock) will need to properly handle priority donation and the many scenarios (including, but not limited to, nested donation).

2 Pintos List

2.1 List API

The PintOS kernel provides an implementation of a doubly linked list in */lib/kernel/list.h* and *lib/kernel/list.c* which can be used in a number of different contexts within PintOS. Lists are used to manage the ready_list of threads waiting to execute on the processor and semaphores maintain their own list of threads waiting for access to resources and memory allocation utilizes lists to keep track of free blocks of memory. The API for the PintOS list is similar to the list in the C++ standard template library and contains many of the same operations. The PintOS list provides functions for traversing a list (forwards and backwards), inserting elements into the list, removing elements from the list, returning the size of the list, returning the maximum or minimum elements from the list and sorting the list. The most pertinent list functions and structures are summarized below. For an exhaustive list please see */lib/kernel/list.h* and *lib/kernel/list.c*.

- *struct list_elem* - A list element, contains pointers to the previous and next list elements. Any structure that is a potential list must embed a *struct list_elem* member. All list functions operate on these list elements.
- *struct list* - Main list structure. Contains pointers to the head and tail of the list.
- *list_entry* - Because list elements are embedded within the actual structure we wish to access, list_entry allows conversion of the list_elem pointer to a pointer to the containing object.
- *list_begin()* - returns a pointer to the beginning of the list.
- *list_insert()* - Inserts a new list entry just before a given entry currently in the list.
- *list_push_front()* and *list_push_back()* - Insert an element into the front or back of list, respectively.
- *list_pop_front()* and *list_pop_back()* - Remove element from front or back of list.
- *list_size()* - Returns an integer, the number of elements in the list.
- *list_empty()* - Returns true if list empty, false otherwise.
- *list_sort()* - Sorts list into ascending order.
- *list_insert_ordered()* - Inserts new list entry into sorted list at the proper position.

2.2 Scheduling

Because a number of Pintos problems (like priority scheduling) rely on some implementation of a priority queue, the doubly linked list provided can be quite useful. Although a more complex data structure like a max-heap might be slightly more efficient for this purpose, the list API provides all the same functionality. The list_sort(), list_insert_ordered() functions are perhaps the most relevant here. If we need to manage a priority queue, the list_sort() function implements an O(n*lg(n)) sorting algorithm. The ordered list can be maintained using the list_insert_ordered() function which will place a new list element into the proper position. An important thing to note is that the list sorting algorithm sorts into increasing order and so servicing a list item with highest priority will require accessing the last item in the list. Pintos provides list_begin() and list_end() functions to allow iterating through a list front-to-back or back-to-front, respectively.

Open Questions

1. If the depth of nested priority donation d is clamped, what should happen to the locks of the chain of threads $t_{1:d}$ when a thread of greatest priority t_{d+1} yields in order to minimize, if not negate, starvation?

2. Would implementing our priority queues as a binary tree improve performance and/or simplify access to queues?

3. What are the trade-offs of implementing a single queue of ready to run threads vs. a single queue for each of the 64 priority levels?

Conclusion

Because thread preemption and external interrupts are components of Pintos, synchronization primitives are vital in ensuring that shared resources are being accessed appropriately and ultimately yielding deterministic results. The upcoming Pintos Project 1 assignment extends this functionality by implementing a priority scheduler which manipulates the threads locks and the semaphores that they are built upon. By understanding the propriety of semaphores and approaching the assignment from a top-down level, Pintos scheduling routines can be properly extended to handle, assign, and donate priority between threads.

Appendix A Mutual Exclusion Example

```
/*
Mutual Exclusion Use Case (Locks).

In this example, we use the Pintos lock structure (found in /src/threads/synch
    .c)
to provide mutual exclusion on a Resource.

Since Pintos locks are implemented using semaphores, this operation can also
    be performed by using a Semaphore with an initial value of 1. Calling
    sema_down() will decrement the value to 0. Any other threads calling
    sema_down() will be added to the waitlist and wait until the thread with
    access calls sema_up()
*/

/*
Part 1 - Lock Setup

This should only be done once per lock. This code should not run on every
    thread that
needs the lock.
*/

// Our lock structure.
struct lock *myLock = (struct lock*) malloc(sizeof(struct lock));

// Initialize the lock (This function calls sema_init for the lock's semaphore
     member).
lock_init(&myLock);

/*
Part 2 - Lock Usage
This is an example of code that would run in a thread. The 'Resource' class is
     just used to illustrate
the usage of a lock.
*/

// This will attempt to acquire the lock. If it is not available, the thread
    is
// added to the waiting list, and will sleep until the lock is available.
// Internally, lock_acquire() calls sema_down().
lock_acquire(&myLock);

// Perform some operation on the Resource we have access to.
modify(Resource);

// Release the lock once we have finished with the resource.
lock_release(&myLock);
```

Appendix B General Waiting Example

```
/*
Part 1 - Main Thread Setup.

In the main() function, we create and allocate space for a new semaphore, then
    initialize it.

We create a child thread, and then call sema_down() on our semaphore, which
    will add the current thread to the waiting list for the semaphore.
*/
main(){
        // Create a pointer to a semaphore structure.
        struct semaphore *my_semaphore = (struct semaphore*) malloc(sizeof(
            struct semaphore));

        // Initialize the semaphore with a value of 0. This means that the
            first thread to
        // call sema_down() will wait, because the semaphore's value is 0.
        sema_init(&my_semaphore,0);

        // Create a new 'child' thread, which our main thread will wait for.
        // We can use the 'aux' field of Pintos' thread_create() function to
            pass a reference
        // to the semaphore.
        thread_create("Child",PRI_DEFAULT,do_something,&my_semaphore);

        // Call sema_down() on the semaphore. This will wait for the child
            thread to call
        // sema_up().
        sema_down(&my_semaphore);

        // At this point, the child thread has completed.
        keep_working()
}
/*
Part 2 - Child Thread

In the child thread, we receive a pointer to the semaphore. We can do some
    work and then call sema_up(), which will remove the first thread from the
    waiting list and unblock it.

In our case, this is the main thread that called sema_down().
*/
do_something(void* aux){
        // Cast the void pointer back to a struct semaphore*.
        struct semaphore *my_semaphore = (struct semaphore*) aux;
        /*
        Code to do work here.
        */
        // When we're done working, we can wake up the parent thread by
            calling  //sema_up().
                sema_up(&my_semaphore);
}
```

References

- Pintos Source Code
  - /lib/kernel/list.h
  - /lib/kernel/list.c
  - /src/threads/synch.h
  - /src/threads/synch.c
- Operating Systems: Principles and Practices Second Edition by Thomas Anderson and Michael Dahlin
- Pintos Manual by Ben Pfaff