CS 111

Scribe Notes for 4/25/05

by Irina Litvin, Aleksandr Liber, Patrick Brophy, Alex Omoto

Synchronization II

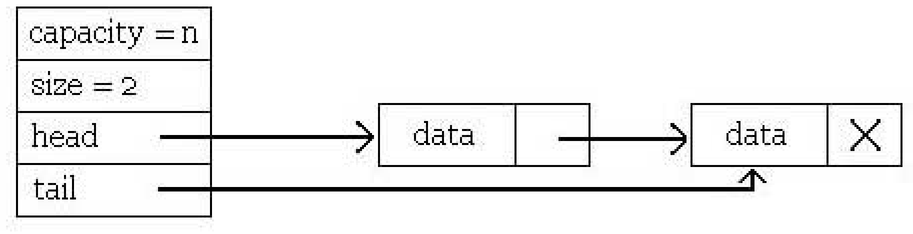
Queues

Let's take a look at synchronization with a very simple example: Basic Queue.

A queue is a first-In, first-Out (FIFO) list of data items. A bounded queue is a queue whose length is bounded by a certain capacity. There are two operations that can be performed on a queue:

- enqueue - puts a data item onto the 'tail' end of the queue
- dequeue - takes the data item from the 'head' end of the queue, removes it and returns it

A typical queue will look something like this:



Let's now take a look at the source code provided in the handout on page 1 and consider possible synchronization issues.

```
void queue_enq(queue_t *q, queuedata_t data)
{
        queueelem_t *qe = (queueelem_t *) malloc(sizeof(queueelem_t));
        assert(q->size < q->capacity);
        qe->data = data;
        qe->next = NULL;
        if (q->tail)
                q->tail->next = qe;
        else
                q->head = qe;
        q->tail = qe;
        q->size++;
}

queuedata_t queue_deq(queue_t *q)
{
        queueelem_t *qe;
        queuedata_t data;
        assert(q->size > 0);
        qe = q->head;
        q->head = qe->next;
        if (q->head == NULL)
                q->tail = NULL;
        q->size--;
        data = qe->data;
        free((void *) qe);
        return data;
}
```

Now, let's suppose that multiple processes may access the queue at the same time.

What is the problem with this code? The problem with this code is that if multiple processes access the queue concurrently, the integrity of the queue will be compromized, inserting incorrect/garbage values. What can we do to fix that? We can ensure that all access to the queue itself will be protected. To do that, we will will isolate the queue access code into a critical section and make sure that no one can enter the critical section once we're in there.

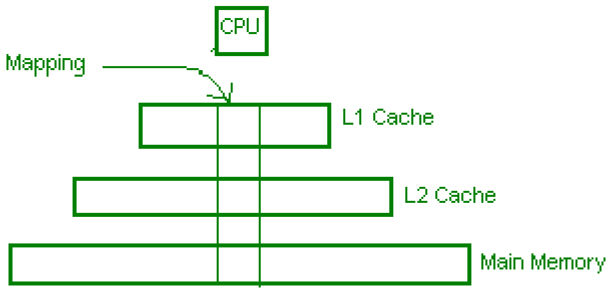
```
void queue_enq(queue_t *q, queuedata_t data)
{
        queueelem_t *qe = (queueelem_t *) malloc(sizeof(queueelem_t));
        assert(q->size < q->capacity);
        qe->data = data;
        qe->next = NULL;
        spin_lock(&q->lock);
        if (q->tail)
                q->tail->next = qe;
        else
                q->head = qe;
        q->tail = qe;
        q->size++;
        spin_unlock(&q->lock);
}

queuedata_t queue_deq(queue_t *q)
{
        queueelem_t *qe;
        queuedata_t data;
        assert(q->size > 0);
        spin_lock(&q->lock);
        qe = q->head;
        q->head = qe->next;
        if (q->head == NULL)
                q->tail = NULL;
        q->size--;
        spin_unlock(&q->lock);
        data = qe->data;
        free((void *) qe);
        return data;
}
```

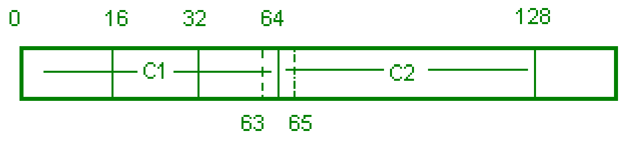
When creating a critical section of code, we must also worry about the following:

- interrupts - mutual exclusion
- multiple threads of control
    - threads/processes - user level
    - SMP - multiple processors

Why do we need to consider these issues? To explain, lets look at an example implementation of locking:

```
spin_lock(lock_t *l)
```





```
{
while(*l > 0)
        /* spin */
        *l = 1;
}

spin_unlock(lock_t *l)
{
        *l = 0;
}
```

We still have a problem, since locking is not atomic! If multiple processes are trying to put a lock on the same thing, one process may be interrupted and the lock may be taken over by another process. Also, whereas before memory access was atomic, in current computers, that is not always the case. Take a look at the follwing instruction:

```
movl var, %eax          x = 1;     x = 10000;
```

What do you think will happen if we make the call this call?

```
printf("%d\n", x);
```

As it turns out, both 1 or 10000 have a chance to be displayed. The value printed to the screen can be either!

Cache lines

Mappings to cache lines usually happen atomically.

Is is possible for the mapping to not be atomic? What happens if the data (lets say an integer) is located at the tail of one cache line and the head of another cache line?

The integer is located between 63 and 65 which spans cache lines C1 and C2. This will mean that the integer's mapping will not be atomic. (Note: The x86 has 32-byte cache lines.)

How do we solve the problem of locks not being mutually exclusive? We need a better way to set and swap the lock status. The following functions are what is needed for lock and unlock.

(code in red is atomic)

```
int
test_and_set (int *addr, int val)
{
     int old = *addr;
     *addr = val;
     return old;
}
```

```
int compare_and_swap(int *addr, int test, int val)
{
    if(*addr == test) {
       *addr = val;
       return 1;
    }
   else
    return 0;
}
```

So the new spin_lock looks like this:

```
spin_lock(lock_t *l)
   while(test_and_set( l , 1) == 1)
       /* do nothing */;
```

This will work fine since lock can only be 0 or 1. The spin_unlock function does not need to be changed because its only instruction is already atomic. Are there any more problems with spin_lock? Well, it doesn't handle interrupts very well. Also, what if there is a bug in the code that tries to unlock without first locking? There is no way to prevent this in C, but there are techniques in other languages that attempt to prevent this.

What if 500 processes try to lock? They are all going to have to "spin" and wait. This will be fine if the critical section is short, but not if it is long. To solve this problem, the processes will have to be taken off the run queue until the lock is available. To do this we use "sleep" and "wake-up" functions.

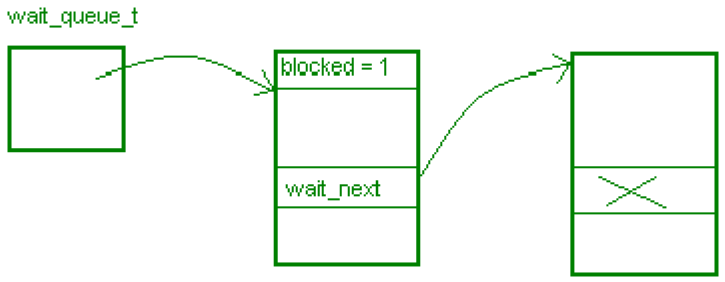
```
void
sleep(process_t *p, waitqueue_t *wq)
{
    p->waiting = wq;
    p->wait_next = wq->next;
    wq->next = p;
    p->blocked = 1;
    yeild(); //will not return until wq is woken up
}
```

```
void wakeup(waitqueue_t *wq)
{
    process_t *p;
    while((p = wq->next)) {
        wq->next = p->wait_next;
        p->waiting = 0;
        p->wait_next = NULL;
    }
}
```

If the lock is busy, the waiting processes will sleep until the lock is is not longer busy. The wait queue is associated with the lock function. Lock will work by signaling when a lock is available then waking up any writing processes. How should the processes be woken up? (FIFO? LIFO? All at once?) This is the choice for the programmer. The following modified functions now use sleep and wakeup calls.

```
spin_lock(lock_t *l)
{
    while (test_and_set(l, 1) == 1)
        sleep(self, &l->wq);
}
```

```
spin_unlock(lock_t *l)
{
```



```
    *l = 0;
    wakeup(&l->wq)
}
```

Is there a problem with this sleep and wakeup implementation? Yes there is. There are no critical sections so this can go wrong is numerous ways. In the worst case, it switches after the while loop in spin_lock over to spin_unlock. This will cause the processes in the wait queue to sleep forever. So what about putting the sleep and wakeup in critical sections? This will not work. It will protect the wait queue but not the higher stuff in spin_lock.

SYNCHRONIZATION OBJECTS:

Thus far we have only looked at the issues for a locking policy, which includes mutual exclusion and read/write exclusion.

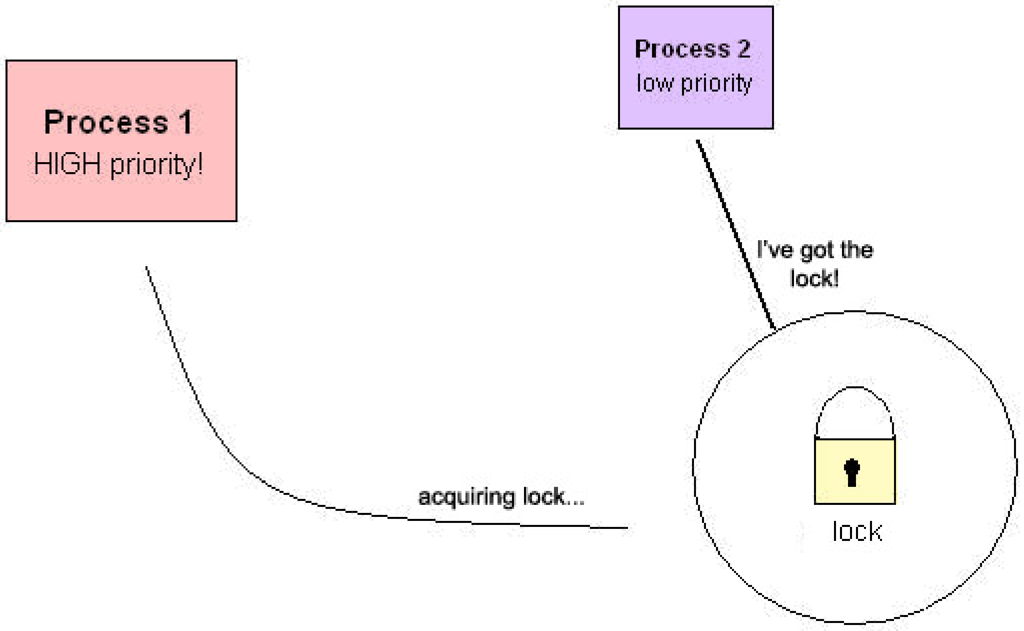
Now, we must deal with the following issues of waiting for the lock:

(1) Should the process "sleep" during acquire/lock?
(2) Who wakes up?
(3) Priority inversion problem

The answer to question (1) is yes, the process should "sleep" if it is trying to acquire a lock that is unavailable. This was previously determined because the process is wasting CPU resources if it continually polls the lock status! A "sleep" action is therefore implemented by putting the process on a wait queue.

To determine the next process to wake up (2), the answer depends on the implementation of the wait queue (FIFO, round-robin, shortest-job-first, and priority). In Lecture 5, we analyzed 4 different wait queues and their advantages with respect to CPU utilization, turn-around time, waiting time, throughput, and response time. Who will wake up first will be determined by the users of the system.

When a priority queue is used, there are the issues of how to schedule priority as well as starvation. With locks, there is also the priority inversion problem:



Suppose the next process in the wait queue is of very low priority. It acquires the lock, and it takes a very long time to execute due to the little CPU attention it gets. Meanwhile, a process of very high priority tries to grab the lock, and since the lock is taken, it must wait until the low-priority process releases the lock. This is called priority inversion.

Definition (priority inversion) = a high priority process is blocked on a resource held by a low priority process.

Problem solution: The low priority process has two attributes that cause this problem: it gets little CPU running time and it holds the lock. Therefore, a solution to solving this issue is to raise the priority of the low process such that it will release the lock sooner. How high do we raise the priority? The best way to do this is to match the highest priority of the processes:

if Process1 tries to acquire the lock but is taken by another process, Process2
then Process 2 priority = Max (Process1 priority, Process2 priority)
until release;

Mutex (mutual exclusion) is a synchronization object which helps to protect shared data structures from being concurrently accessed and modified. Threads that want to access a resource protected by a mutex must wait until the currently active thread is finished and unlocks the mutex. A semaphore is another synchronization object that allows zero to any number of threads access simultaneously. Other objects include p_thread_mutex_t and synchronized (object) {}.

```
lock_acquire(l) {
   acquire global mutex;
   if (l already locked) {   // again:
      add to wait queue;
      release global mutex;
      yield();
      goto again;
   }
   lock l;
   release global mutex;
   }
```

```
lock_release(l) {
   acquire global mutex;
   unlock l;
   wake up processes on wait queues;
   release global mutex;
}
```

The above code are the functions for acquiring and releasing the lock using mutex. There are critical sections over the access of the lock and the operations on the wait queue. Recall that without the mutex, multiple processes may test at the same time that l is unlocked. Furthermore, note the problems that arise when multiple processes are enqueued and dequeued at the same time.

The lock_acquire function is now revised such that the overhead of context switching is reduced.

```
lock_acquire(l)
   if (l locked)
      sleep;
   acquire mutex
   if (l locked)
      release;
      sleep;
}
```

Condition Variables
Purpose: be able to go to sleep and yield if some complex condition that we need is not met. This allows other processes to do useful work while we are waiting for the right conditions.

```
//This function has a regular spin lock but in addition it has a condition_wait function,
//that waiting of a specific condition to be satisfied. condition_signal will signal
//that a condition has been met.
//The two functions work in the following way
//wait(v,l);
//      unlock l
//      wait until signal v
//      lock l
//
//signal(l)
//      wakes up any waiting threads
void
queue_enq(queue_t *q, queuedata_t data)
{
    queueelem_t *qe = (queueelem_t *) malloc(sizeof(queueelem_t));
    qe->data = data;

    //grab a lock
    spin_lock(&q->lock);
    //here we can see that the condition being checked is that of the queue not being
full.
    while (q->size == q->capacity)
        //Release the lock and wait until a nonfull signal is sent. Then it will double
        //check that the queue is not actually full. If the queue is still full it will
        //wait again for a signal. Otherwise it will lock and add something to the queue
        condition_wait(&q->nonfull, &q->lock);

    qe->data = data;
    qe->next = NULL;
    if (q->tail)
        q->tail->next = qe;
    else
        q->head = qe;
    q->tail = qe;
    q->size++;

    spin_unlock(&q->lock);
    //The second needed function is one to actually signal that a condition has been met.
    //This one will wake up the processes trying to take something off the queue
informing
    //them that something has been placed on the queue so they can proceed
    condition_signal(&q->nonempty);
}

queuedata_t
queue_deq(queue_t *q)
{
    queueelem_t *qe = NULL;
    queuedata_t data;

    spin_lock(&q->lock);
    //similar to the above waiting for the queue to be nonempty
    while (q->size == 0)
        condition_wait(&q->nonempty, &q->lock);

    qe = q->head;
    q->head = qe->next;
    if (q->head == NULL)
        q->tail = NULL;
    q->size--;

    spin_unlock(&q->lock);
    data = qe->data;
    free((void *) qe);
    //notifying everyone that the queue is no longer full.
    condition_signal(&q->nonfull);
    return data;
}


These two functions are used to ensure that no process tries to put something
on the queue if the queue is full, or tries to take something off a queue that
has nothing on it.
```

Limiting Concurrency

Create a wait free data structure that never needs to lock. Such a structure will be faster since there is no locking overhead and by its design it avoids nasty concurrency bugs. A downside however is that it is more complex than structures that use locking.

Before we has a queue whose properties could be modified by any number of processes so we needed to lock the queue to prevent processes from stepping on each others toes. We design a structure with the assumption that there will only be one process doing an enqueue and one doing a dequeue. However we can still have an enq and a deq at the same time. This can be solved by noting that the enq function works mostly with the tail pointer and the deq with the head pointer. We replace the linked list with a circular array and have the head and tail point to the slots for enq and deq. Essentially we split the size value into two parts one that is dealt with by enq and other by deq allowing both operations to run at the same time. Note: look at the very nice example Dr. Kohler provided.