Overview

- Overview of semaphores
- Basic semaphore implementation
- Synchronization problems using semaphores
- Correct blocking semaphore implementation
- Comparison with locking and monitors
Overview of Semaphores

- Semaphores convert mutual exclusion and synchronization into a resource management problem
  - Threads acquire resources when they need them, waiting when resources are not available
  - Threads release resources when they are not needed, waking threads that are waiting on resources

- A semaphore is a variable that tracks number of available resources using two operations:
  - down: acquires a resource (why down?)
  - up: releases a resource (why up?)
Basic Semaphore Implementation

```c
semaphore s = INIT_NR_RESOURCES;

down(semaphore s) {
    while (s <= 0) { // wait until resource is available
        s = s - 1;    // acquire a resource
        // after down(), s >= 0
    }
}

up(semaphore s) {
    s = s + 1;    // make a resource available
}
```

- `down()` and `up()` share semaphore
  - Implementations require mutual exclusion
  - Correct implementation later
Variable Initialization Using Semaphores

Why does this code work?

```c
char *V = NULL;
semaphore init_sem = 0;

// called by Thread T1
Init() {
    V = malloc(...);
    // signal that V is initialized
    up(init_sem);
    ...
}

// called by Thread T2
Use() {
    // wait until V is initialized
    down(init_sem);
    assert(V);
    // read V
    ...
}
```

Sync
Producer-Consumer with Semaphores

Global variables:
buf[n], in, out;
sem full = 0;  // no full slots
sem empty = n;  // all slots are empty
lock l = 0;

void send(char elem) {
    down(empty);
    lock(l);
    buf[in] = elem;
    in = (in + 1) % n;
    unlock(l);
    up(full);
}

char receive() {
    down(full);
    lock(l);
    elem = buf[out];
    out = (out + 1) % n;
    unlock(l);
    up(empty);
    return elem;
}

• Why does the code not check the full and empty buffer conditions?
• Does code work for multiple producers/consumers?
• Can we switch down(), lock()?
Revisiting Basic Semaphore Implementation

- Basic semaphore implementation had two problems
  - down() operation used spinning when waiting for a resource
  - Both down() and up() required mutual exclusion

```c
semaphore s = INIT_NR_RESOURCES;

down(semaphore s) {
    while (s <= 0) { // wait until resource is available
    }
    s = s - 1;    // acquire a resource
                      // after down(), s >= 0
}

up(semaphore s) {
    s = s + 1; // make a resource available
}
```
Implementing Blocking Semaphores

- How can mutual exclusion be implemented?
  - Disable interrupts on uniprocessors
  - Use spinlocks on multi-processors

```
struct semaphore { int count; wait_queue *wq; ... }

down(semaphore *sem) {
    while (sem->count <= 0) {
        thread_sleep(sem->wq);
    }
    sem->count--;
}

up(semaphore *sem) {
    sem->count++;
    thread_wakeup(sem->wq);
}
```
This code works correctly, but why?
  - Why does `down()` disable interrupts before calling `thread_sleep()`?
  - Isn’t sleeping with interrupts disabled a problem?
Semaphores versus Locks

- When a semaphore $s$ is initialized to 1, then
  - $\text{down}(s)$ behaves like $\text{lock}(s)$
  - $\text{up}(s)$ behaves like $\text{unlock}(s)$

- Difference between semaphores and locks is in usage
  - Semaphores for synchronization, locks for mutex
  - Different threads call $\text{down}()$ and $\text{up}()$ for synchronization
  - $\text{up}()$ can be called before $\text{down}()$, to “bank” resources
Semaphores versus Monitors

<table>
<thead>
<tr>
<th>Init()</th>
<th>Use()</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock(l); V = malloc(...); signal(init_cv, l); unlock(l);</td>
<td>lock(l); if (V == NULL) { wait(init_cv, l); } assert(V); // read V unlock(l);</td>
</tr>
</tbody>
</table>

Variable initialization using monitor

<table>
<thead>
<tr>
<th>Init()</th>
<th>Use()</th>
</tr>
</thead>
<tbody>
<tr>
<td>V = malloc(...); up(init_sem);</td>
<td>down(init_sem); assert(V); // read V</td>
</tr>
</tbody>
</table>

Variable initialization using semaphore
Summary

- Threads use synchronization to order their operations
  - A thread waits on a condition
  - When another thread makes the condition occur, it wakes up waiting thread

- Semaphores provide a systematic method for synchronization
  - Semaphores track number of available resources
  - Provide semaphore variables with down and up operations

- Solutions for variable initialization, producer-consumer using semaphores

- Blocking semaphore implementation

- Next lecture, we solve fun synchronization problems
Think Time: Semaphores

- Why are locks not sufficient for solving synchronization? How are locks different from semaphores?
- What are the P and V operations on a semaphore?
Think Time Answers: Semaphores

- Why are locks not sufficient for solving synchronization? How are locks different from semaphores?
  - (lock, unlock) is used together by each thread, and in that order, to ensure mutual exclusion. Synchronization problems require a more general primitive: conditional sleep and wakeup. One thread performs the sleep, the other the wakeup. For example, down and up operations are issued by different threads, and can occur in either order.

- What are the P and V operations on a semaphore?
  - P and V operations: down(s) is also called P(s), up(s) is also called V(s). Read book to find out why the down/up operations were originally called P and V.
Think Time: Blocking Semaphores

- How would you solve the producer-consumer problem using interrupt disabling on a single CPU?
- How would you implement a blocking semaphore on a multi-processor?
Think Time Answers: Blocking Semaphores

- How would you solve the producer-consumer problem using interrupt disabling on a single CPU?
  - You would, in essence, do the work of down/up or wait/signal in the producer consumer code. try it.
How would you implement a blocking semaphore on a multi-processor?

- A multi-processor implementation is more complicated. In addition to disabling interrupts (so there is no concurrency with the local CPU), a spinlock on a wait_queue associated with the sem semaphore needs to be acquired (to avoid a lost wakeup). This spinlock must be released in the scheduler because threads can’t sleep while holding the spinlock (or a deadlock will happen). To ensure mutual exclusion and avoid a wakeup being lost, the scheduler’s thread_sleep function must first acquire a ready_queue spinlock (this could be per-processor), add the current thread to the wait queue, and only then release the wait_queue spin lock before calling thread_switch. When the thread returns from thread_switch, thread_sleep should reacquire the wait_queue spin lock, and then release the ready_queue spin lock. (Continued)
Think Time Answers: Blocking Semaphores

- How would you implement a blocking semaphore on a multi-processor?
  - (Continued) Similarly, thread_wakeup must be called after acquiring the wait_queue spin_lock, and it should acquire the ready_queue spinlock when the thread being woken up is added to the ready queue. These steps will ensure that there are no deadlocks and there is no race between the up and down operations (i.e., a wakeup is not lost). You will need to think why.
Think Time: Monitors or Semaphores

- How are monitors, semaphores different?
- Why do semaphores require initialization, but condition variables don’t require initialization?
- What are the differences between wait() and down()? 
- What are the differences between signal() and up()?
- Why might you prefer monitors or semaphores?
Think Time Answers: Monitors or Semaphores

- How are monitors, semaphores different?
  - Monitors require locks, semaphores don’t. Their initialization, the semantics of down and wait, and up and signal are quite different, see next set of questions.

- Why do semaphores require initialization, but condition variables don’t require initialization?
  - Semaphores track resources, and so the programmer needs to initialize a semaphore with the initial number of resources. condition variables do not hold any state, such as the number of resources, and so they do not require initialization.
Think Time Answers: Monitors or Semaphores

- What are the differences between `wait()` and `down()`?
  - Wait doesn’t track any resources, it always waits. `down` has a notion of available resources, and will only wait if resources are not available. `wait` releases a lock, waits, and then reacquires a lock. `down` doesn’t have any notion of an associated lock.

- What are the differences between `signal()` and `up()`?
  - A signal can be lost if no one is waiting. hence the need to use locks with condition variables, so that there is no race with `wait()`. an `up()` will always increase the resource available, so a future `down` can acquire the resource.
Think Time Answers: Monitors or Semaphores

Why might you prefer monitors or semaphores?

- Some resource counting problems, including the variable initialization (one resource) and the producer-consumer problem (n empty slots, n full slots) are easily solved using semaphores. For other problems, it is easier to use monitors because they provide mutual exclusion, and the programmer can write arbitrary conditions for waiting.