CS 153
Design of Operating Systems

Fall 19

Lecture 7: Synchronization

Instructor: Chengyu Song

Process vs Thread

Cooperation between Threads

- What is the advantage of threads over process?
  - Faster creation
  - Easier share of resources, access shared data structures
    - Threads accessing a memory cache in a Web server
- Threads cooperate in multithreaded programs
- Why?
  - To coordinate their execution
    - One thread executes relative to another

Threads: Sharing Data

```
int count = 0; //shared variable since its global

void twiddledee() {
  int i=0; //for part b this will be global and shared
  for (i=0; i<2; i++) {
    count = count * count; //assume count read from memory once
}}

void twiddledum() {
  int i=0; // for part b, this will be global and shared
  for(i=0; i<2; i++) { count = count - 1;}
}

void main() {
  thread_fork(twiddledee);
  thread_fork(twiddledum);
  print count;
}
```

What are all the values that could be printed in main?

Threads: Cooperation

- Threads voluntarily give up the CPU with thread_yield

Ping Thread

```
while (1) {
    printf("ping\n");
    thread_yield();
}
```

Pong Thread

```
while (1) {
    printf("pong\n");
    thread_yield();
}
```

Synchronization

- For correctness, we need to control this cooperation
  - Threads interleave executions arbitrarily and at different rates
  - Scheduling is not under program control
- We control cooperation using synchronization
  - Synchronization enables us to restrict the possible inter-leavings of thread executions

What about processes?

- Does this apply to processes too?
  - Yes!
- What synchronization system call you have seen?
  - **`wait()`**
- Do I need to learn this if I don’t write multi-thread programs?
  - But share the OS structures and machine resources so we need to synchronize them too
  - Basically, the OS is a multi-threaded program

Shared Resources

We initially focus on coordinating access to shared resources

- Basic problem
  - If two concurrent threads are accessing a shared variable, and at least one thread modified/written the variable, then access to the variable must be controlled to avoid erroneous behavior
- Over the next couple of lectures, we will look at
  - Exactly what problems occur
  - How to build mechanisms to control access to shared resources
    - » Locks, mutexes, semaphores, monitors, condition variables, etc.
  - Patterns for coordinating accesses to shared resources
    - » Reader-writer, bounded buffer, producer-consumer, etc.

A First Example

- Suppose we have to implement a function to handle withdrawals from a bank account:

```
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance - amount;
    put_balance(account, balance);
    return balance;
}
```

- Now suppose that you and your father share a bank account with a balance of $1000
- Then you each go to separate ATM machines and simultaneously withdraw $100 from the account

Example Continued

- We'll represent the situation by creating a separate thread for each person to do the withdrawals
- These threads run on the same bank machine:

```
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

```
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

- What's the problem with this implementation?
  - Think about potential schedules of these two threads

Interleaved Schedules

- The problem is that the execution of the two threads can be interleaved:

Execution sequence seen by CPU

```
balance = get_balance(account);
balance = balance – amount;
```

```
balance = get_balance(account);
balance = balance – amount;
put_balance(account, balance);
```

```
put_balance(account, balance);
```

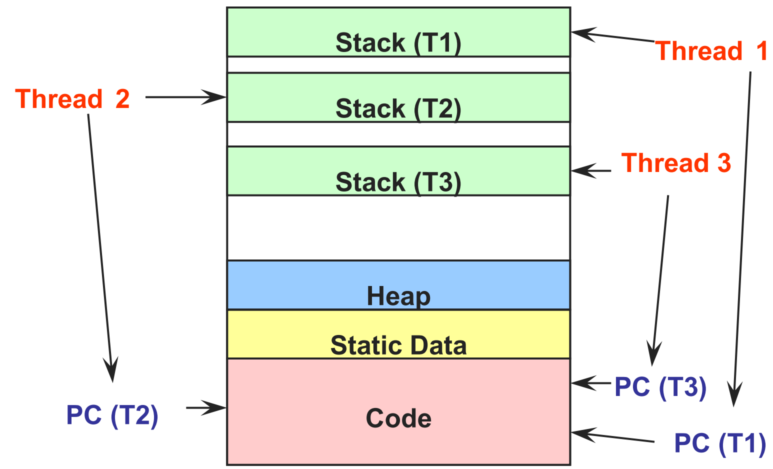
Context switch

- What is the balance of the account now?

Shared Resources

- Problem: two threads accessed a shared resource
  - Known as a race condition (remember this buzzword!)
- Need mechanisms to control this access
  - So we can reason about how the program will operate
- Our example was updating a shared bank account
- Also necessary for synchronizing access to any shared data structure
  - Buffers, queues, lists, hash tables, etc.

What Resources Are Shared?



- Local variables?
  - Not shared: refer to data on the stack
  - Each thread has its own stack
  - Don't pass/share/store a pointer to a local variable on the stack for thread T1 to another thread T2
- Global variables and static objects?
  - Shared: in static data segment, accessible by all threads
- Dynamic objects and other heap objects?
  - Shared: Allocated from heap with malloc/free or new/delete

How Interleaved Can It Get?

How contorted can the interleavings be?

- We'll assume that the only atomic operations are reads and writes of individual memory locations
  - Some architectures don't even give you that!
- We'll assume that a context switch can occur at any time
- We'll assume that you can delay a thread as long as you like as long as it's not delayed forever

.... get_balance(account);
balance = get_balance(account);
balance = ...
balance = balance – amount;
balance = balance – amount;
put_balance(account, balance);
put_balance(account, balance);

What do we do about it?

- Does this problem matter in practice?
- Are there other concurrency problems?
- And, if so, how do we solve it?
    - Really difficult because behavior can be different every time
- How do we handle concurrency in real life?

Mutual Exclusion

- Mutual exclusion to synchronize access to shared resources
  - This allows us to have larger “atomic” blocks
- Code that uses mutual called a critical section
  - Only one thread at a time can execute in the critical section
  - All other threads are forced to wait on entry
  - When a thread leaves a critical section, another can enter
  - Example: sharing an ATM with others
- What requirements would you place on a critical section?

Critical Section Requirements

Critical sections have the following requirements:

1) Mutual exclusion (mutex)
   - If one thread is in the critical section, then no other is

2) Progress
   - A thread in the critical section will eventually leave the critical section
   - If some thread T is not in the critical section, then T cannot prevent some other thread S from entering the critical section

3) Bounded waiting (no starvation)
   - If some thread T is waiting on the critical section, then T will eventually enter the critical section

4) Performance
   - The overhead of entering and exiting the critical section is small with respect to the work being done within it

About Requirements

There are three kinds of requirements that we'll use

- Safety property: nothing bad happens
  - Mutex
- Liveness property: something good happens
  - Progress, Bounded Waiting
- Performance requirement
  - Performance
- Properties hold for each run, while performance depends on all the runs
  - Rule of thumb: When designing a concurrent algorithm, worry about safety first, but don't forget liveness!

Mechanisms For Building Critical Sections

- Locks
  - Primitive, minimal semantics, used to build others
- Architecture help
  - Atomic test-and-set
- Semaphores
  - Basic, easy to get the hang of, but hard to program with
- Monitors
  - High-level, requires language support, operations implicit

Locks

- A lock is an object in memory providing two operations
  - acquire(): before entering the critical section
  - release(): after leaving a critical section
- Threads pair calls to acquire() and release()
  - Between acquire()/release(), the thread holds the lock
  - acquire() does not return until any previous holder releases
  - What can happen if the calls are not paired?

Using Locks

```
withdraw (account, amount) {
    acquire(lock);
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    release(lock);
    return balance;
}
```

Critical Section

```
acquire(lock);
balance = get_balance(account);
balance = balance – amount;

acquire(lock);

put_balance(account, balance);
release(lock);

balance = get_balance(account);
balance = balance – amount;
put_balance(account, balance);
release(lock);
```

- Why is the "return" outside the critical section? Is this ok?
- What happens when a third thread calls acquire?

How do we implement a lock? First try

```
pthread_trylock(mutex) {
    if (mutex==0) {
     mutex= 1;
     return 1;
    } else return 0;
}
```

```
Thread 0, 1, …

…//time to access critical region
while(!pthread_trylock(mutex); // wait
<critical region>
pthread_unlock(mutex)
```

- Does this work? Assume reads/writes are **atomic**
- The lock itself is a critical region!
  - Chicken and egg
- Computer scientist struggled with how to create software locks

Second try

```
int turn = 1;
```

```
while (true) {
    while (turn != 1) ;
    critical section
    turn = 2;
    outside of critical section
}
```

```
while (true) {
    while (turn != 2) ;
    critical section
    turn = 1;
    outside of critical section
}
```

This is called alternation

It satisfies mutex:

- If blue is in the critical section, then turn == 1 and if yellow is in the critical section then turn == 2
- (turn == 1) ≡ (turn != 2)

Is there anything wrong with this solution?

Third try – two variables

```
bool flag[2] = {0, 0};
```

```
while (flag[1] != 0);
flag[0] = 1;
```
critical section
```
flag[0]=0;
```
outside of critical section

```
while (flag[0] != 0);
flag[1] = 1;
```
critical section
```
flag[1]=0;
```
outside of critical section

We added two variables to try to break the race for the same variable

Is there anything wrong with this solution?

Fourth try – set before you check

```
bool flag[2] = {0, 0};
```

```
flag[0] = 1;
while (flag[1] != 0);
critical section
flag[0]=0;
outside of critical section
```

```
flag[1] = 1;
while (flag[0] != 0);
critical section
flag[1]=0;
outside of critical section
```

Is there anything wrong with this solution?

Fifth try – double check and back off

```
bool flag[2] = {0, 0};
```

```
flag[0] = 1;
while (flag[1] != 0) {
        flag[0] = 0;
        wait a short time;
        flag[0] = 1;
}
critical section
flag[0]=0;
outside of critical section
```

```
flag[1] = 1;
while (flag[0] != 0) {
        flag[1] = 0;
        wait a short time;
        flag[1] = 1;
}
critical section
flag[1]=0;
outside of critical section
```

Six try – Dekker's Algorithm

```
bool flag[2] = {0, 0};
int turn = 1;
```

```
flag[0] = 1;
while (flag[1] != 0) {
        if (turn == 2) {
        flag[0] = 0;
        while (turn == 2);
        flag[0] = 1;
        } //if
} //while
```
critical section
```
flag[0]=0;
turn=2;
```
outside of critical section

```
flag[1] = 1;
while (flag[0] != 0) {
        if (turn == 1) {
        flag[1] = 0;
        while (turn == 1);
        flag[1] = 1;
        } //if
} //while
```
critical section
```
flag[1]=0;
turn=1;
```
outside of critical section

Peterson's Algorithm

```
int turn = 1;
bool try1 = false, try2 = false;
```

```
while (true) {
    try1 = true;
    turn = 2;
    while (try2 && turn != 1) ;
    critical section
    try1 = false;
    outside of critical section
}
```

```
while (true) {
    try2 = true;
    turn = 1;
    while (try1 && turn != 2) ;
    critical section
    try2 = false;
    outside of critical section
}
```

- This satisfies all the requirements
- Here's why...

Peterson's Algorithm: analysis

```
int turn = 1;
bool try1 = false, try2 = false;
```

```
while (true) {
   {¬ try1 ∧ (turn == 1 ∨ turn == 2) }
1  try1 = true;
   { try1 ∧ (turn == 1 ∨ turn == 2) }
2  turn = 2;
   { try1 ∧ (turn == 1 ∨ turn == 2) }
3  while (try2 && turn != 1) ;
  {  try1 ∧ (turn == 1 ∨ ¬ try2 ∨
       (try2 ∧ (yellow at 6 or at 7)) }
   critical section
4  try1 = false;
   {¬ try1 ∧ (turn == 1 ∨ turn == 2) }
   outside of critical section
}
```

```
while (true) {
   {¬ try2 ∧ (turn == 1 ∨ turn == 2) }
5  try2 = true;
   { try2 ∧ (turn == 1 ∨ turn == 2) }
6  turn = 1;
   { try2 ∧ (turn == 1 ∨ turn == 2) }
7  while (try1 && turn != 2) ;
   {  try2 ∧ (turn == 2 ∨ ¬ try1 ∨
       (try1 ∧ (blue at 2 or at 3)) }
   critical section
8  try2 = false;
   {¬ try2 ∧ (turn == 1 ∨ turn == 2) }
   outside of critical section
}
```

(blue at 4) ∧ try1 ∧ (turn == 1 ∨ ¬ try2 ∨ (try2 ∧ (yellow at 6 or at 7))
∧ (yellow at 8) ∧ try2 ∧ (turn == 2 ∨ ¬ try1 ∨ (try1 ∧ (blue at 2 or at 3))
... ⇒ (turn == 1 ∧ turn == 2)

Some observations

- This stuff (software locks) is hard
  - Hard to get right
  - Hard to prove right
- It also is inefficient
  - A spin lock – waiting by checking the condition repeatedly
- Even better, software locks don't really work
  - Compiler and hardware reorder memory references from different threads
    - Something called memory consistency model
    - Well beyond the scope of this class ☺
- So, we need to find a different way
  - Hardware help; more in a second

Hardware to the rescue

- Crux of the problem:
  - We get interrupted between checking the lock and setting it to 1
  - Software locks reordered by compiler/hardware
- Possible solutions?
  - Atomic instructions: create a new assembly language instruction that checks and sets a variable atomically
    - » Cannot be interrupted!
    - » How do we use them?
  - Disable interrupts altogether (no one else can interrupt us)

Atomic Instruction: Test-and-Set

- The semantics of test-and-set are:
  - Record the old value
  - Set the value to indicate available
  - Return the old value
- Hardware executes it atomically!

```
bool test_and_set (bool *flag) {
    bool old = *flag;
    *flag = True;
    return old;
}
```

One read → while (lock->held);

One write → lock->held = 1;

- When executing test-and-set on "flag"
  - What is value of flag afterwards if it was initially False? True?
  - What is the return result if flag was initially False? True?

Using Test-and-Set

- Here is our lock implementation with test-and-set:

```
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (test-and-set(&lock->held));
}
void release (lock) {
    lock->held = 0;
}
```

- When will the while return? What is the value of held?
- Does it satisfy critical region requirements? (mutex, progress, bounded wait, performance?)

Still a Spinlocks

- The problem with spinlocks is that they are wasteful
  - Although still useful in some cases; lets discuss advantages and disadvantages
- If a thread is spinning on a lock, then the scheduler thinks that this thread needs CPU and puts it on the ready queue
- If N threads are contending for the lock, the thread which holds the lock gets only 1/N'th of the CPU

Disabling Interrupts

- Another implementation of acquire/release is to disable interrupts:

```
struct lock {
}
void acquire (lock) {
    disable interrupts;
}
void release (lock) {
    enable interrupts;
}
```

- Note that there is no state associated with the lock
- Can two threads disable interrupts simultaneously?

On Disabling Interrupts

- Disabling interrupts blocks notification of external events that could trigger a context switch (e.g., timer)
- In a "real" system, this is only available to the kernel
  - Why?
- Disabling interrupts is insufficient on a multiprocessor
  - Back to atomic instructions
- Like spinlocks, only want to disable interrupts to implement higher-level synchronization primitives
  - Don't want interrupts disabled between acquire and release

Summarize Where We Are

- Goal: Use mutual exclusion to protect critical sections of code that access shared resources
- Method: Use locks (spinlocks or disable interrupts)
- Problem: Critical sections can be long

Spinlocks:

- Threads waiting to acquire lock spin in test-and-set loop
- Wastes CPU cycles
- Longer the CS, the longer the spin
- Greater the chance for lock holder to be interrupted
- Memory consistency model causes problems (out of scope of this class)

```
acquire(lock)
…
Critical section
…
release(lock)
```

Disabling Interrupts:

- Should not disable interrupts for long periods of time
- Can miss or delay important events (e.g., timer, I/O)

Higher-Level Synchronization

- Spinlocks and disabling interrupts are useful for short and simple critical sections
  - Can be wasteful otherwise
  - These primitives are “primitive” – don’t do anything besides mutual exclusion
- Need higher-level synchronization primitives that:
  - Block waiters
  - Leave interrupts enabled within the critical section
- All synchronization requires atomicity
- So we’ll use our atomic locks as primitives to implement them

Implementing a Blocking Lock

- Block waiters, interrupts enabled in critical sections

```
struct lock {
    int held = 0;
    queue Q;
}
void acquire (lock) {
    Disable interrupts;
    if (lock->held) {
        put current thread on lock Q;
        block current thread;
    }
    lock->held = 1;
    Enable interrupts;
}
```

```
void release (lock) {
    Disable interrupts;
    if (Q)
        remove and unblock a waiting thread;
    else
        lock->held = 0;
    Enable interrupts;
}
```

acquire(lock)	**Interrupts Disabled**
… *Critical section* …	**Interrupts Enabled**
release(lock)	**Interrupts Disabled**

Implementing a Blocking Lock

- Can use a spinlock instead of disabling interrupts

```
struct lock {
    int held = 0;
    queue Q;
}
void acquire (lock) {
    spinlock->acquire();
    if (lock->held) {
        put current thread on lock Q;
        block current thread;
    }
    lock->held = 1;
    spinlock->release();
}
```

```
void release (lock) {
    spinlock->acquire();
    if (Q)
        remove and unblock a waiting thread;
    else
        lock->held = 0;
    spinlock->release();
}
```

acquire(lock)	Spinning
… *Critical section* …	Running or Blocked
release(lock)	Spinning

Using Locks

```
withdraw (account, amount) {
    acquire(lock);
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    release(lock);
    return balance;
}
```

Critical Section (balance = get_balance(account); balance = balance – amount; put_balance(account, balance);)

```
acquire(lock);
balance = get_balance(account);
balance = balance – amount;
```

```
acquire(lock);
```

```
put_balance(account, balance);
release(lock);
```

```
balance = get_balance(account);
balance = balance – amount;
put_balance(account, balance);
release(lock);
```

- Remember to release the lock!

Mechanisms For Building Critical Sections

- Locks
  - Primitive, minimal semantics, used to build others
- Architecture help
  - Atomic test-and-set
- Semaphores
  - Basic, easy to get the hang of, but hard to program with
- Monitors
  - High-level, requires language support, operations implicit

Semaphores

- Semaphores are an abstract data type that provide mutual exclusion to critical sections
  - Block waiters, interrupts enabled within critical section
  - Described by Dijkstra in THE system in 1968
- Semaphores are **integers** that support two operations:
  - wait(semaphore): decrement, block until semaphore is open
    - Also P(), after the Dutch word for test, or down()
  - signal(semaphore): increment, allow another thread to enter
    - Also V() after the Dutch word for increment, or up()
  - That's it! No other operations – not even just reading its value

Blocking in Semaphores

- Associated with each semaphore is a queue of waiting threads/processes
- When wait() is called by a thread:
  - If semaphore is open (>= 0), and thread continues
  - If semaphore is closed (< 0), thread blocks on queue
- Then signal() opens the semaphore:
  - If semaphore is closed before increase, a thread is waiting on the queue, the thread is unblocked
  - If no threads are waiting on the queue, the signal is remembered for the next thread, but not exceeding the max value

Semaphore Types

- Semaphores come in two types
- Mutex semaphore (or binary semaphore)
  - Represents single access to a resource
  - Guarantees mutual exclusion to a critical section
- Counting semaphore (or general semaphore)
  - Multiple threads pass the semaphore determined by count
    - mutex has count = 1, counting has count = N
  - Represents a resource with many units available
  - or a resource allowing some unsynchronized concurrent access (e.g., reading)

Using Semaphores

- Use is similar to our locks, but semantics are different

```
struct Semaphore {
    int value;
    Queue q;
} S;
withdraw (account, amount) {
    wait(S);
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    signal(S);
    return balance;
}
```

critical section

```
wait(S);
balance = get_balance(account);
balance = balance – amount;
```

Threads block

```
wait(S);
```

```
wait(S);
```

```
put_balance(account, balance);
signal(S);
```

```
…
signal(S);
```

```
…
signal(S);
```

It is undefined which thread runs after a signal

Beyond Mutual Exclusion

- We've looked at a simple example for using synchronization
  - Mutual exclusion while accessing a bank account
- We're going to use semaphores to look at more interesting examples
  - Counting critical region
  - Ordering threads
  - Readers/Writers
  - Producer consumer with bounded buffers
  - More general examples

Readers/Writers Problem

- Readers/Writers Problem:
  - An object is shared among several threads
  - Some threads only read the object, others only write it
  - We can allow multiple readers but only one writer
    - » Let #r be the number of readers, #w be the number of writers
    - » Safety: $(\#r \geq 0) \wedge (0 \leq \#w \leq 1) \wedge ((\#r > 0) \Rightarrow (\#w = 0))$
- Use three variables
  - int readcount – number of threads reading object
  - Semaphore mutex – control access to readcount
  - Semaphore w_or_r – exclusive writing or reading

Readers/Writers

```
// number of readers
int readcount = 0;
// mutual exclusion to readcount
Semaphore mutex = 1;
// exclusive writer or reader
Semaphore w_or_r = 1;

writer {
   wait(w_or_r); // lock out readers
   Write;
  signal(w_or_r); // up for grabs
}
```

```
reader {
   wait(mutex);       // lock readcount
   readcount += 1; // one more reader
   if (readcount == 1)
       wait(w_or_r); // synch w/ writers
   signal(mutex);   // unlock readcount
   Read;
   wait(mutex);      // lock readcount
   readcount -= 1; // one less reader
   if (readcount == 0)
     signal(w_or_r); // up for grabs
   signal(mutex);   // unlock readcount
}
```

Readers/Writers Notes

- w_or_r provides mutex between readers and writers
  - Readers wait/signal when readcount goes from 0 to 1 or 1 to 0
- If a writer is writing, where will readers be waiting?
- Once a writer exits, all readers can fall through
  - Which reader gets to go first?
  - Is it guaranteed that all readers will fall through?
- If readers and writers are waiting, and a writer exits, who goes first?
- Why do readers use mutex?
- What if the signal is above “if (readcount == 1)”?
- If read in progress when writer arrives, when can writer get access?

Avoid Starvation

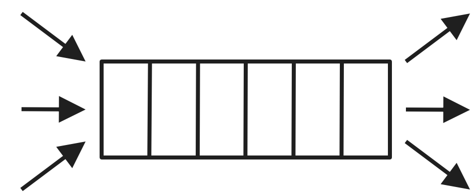
```
// number of readers
int readcount = 0;
// mutual exclusion to readcount
Semaphore mutex = 1;
// exclusive writer or reader
Semaphore w_or_r = 1;
// turnstile for everyone
Semaphore turnstile = 1;

writer {
    wait(turnstile); // get in the queue
    wait(w_or_r); // lock out readers
    Write;
    signal(w_or_r); // up for grabs
    signal(turnstile); // next
}
```

```
reader {
    wait(turnstile);  // get in the queue
    signal(turnstile); // next
    wait(mutex);       // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(w_or_r); // synch w/ writers
    signal(mutex);   // unlock readcount
    Read;
    wait(mutex);      // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(w_or_r); // up for grabs
    signal(mutex);   // unlock readcount
}
```

Bounded Buffer*

- Problem: Set of buffers shared by producer and consumer threads
  - Producer inserts jobs into the buffer set
  - Consumer removes jobs from the buffer set



- Producer and consumer execute at different rates
  - No serialization of one behind the other
  - Tasks are independent (easier to think about)
  - The buffer set allows each to run without explicit handoff
- Data structure should not be corrupted
  - Due to race conditions
  - Or producer writing when full
  - Or consumer deleting when empty

Bounded Buffer (2)*

```
Semaphore mutex = 1;   // mutual exclusion to shared set of buffers
Semaphore empty = N;  // count of empty buffers (all empty to start)
Semaphore full = 0;       // count of full buffers (none full to start)
```

```
producer {
  while (1) {
    Produce new resource;
    wait(empty); // wait for empty buffer
    wait(mutex); // lock buffer list
    Add resource to an empty buffer;
    signal(mutex); // unlock buffer list
    signal(full);      // note a full buffer
  }
}
```

```
consumer {
  while (1) {
    wait(full);          // wait for a full buffer
    wait(mutex);      // lock buffer list
    Remove resource from a full buffer;
    signal(mutex); // unlock buffer list
    signal(empty); // note an empty buffer
    Consume resource;
  }
}
```

Bounded Buffer (3)*

- Why need the mutex at all?
- The pattern of signal/wait on full/empty is a common construct often called an interlock
- Producer-Consumer and Bounded Buffer are classic examples of synchronization problems
  - We will see and practice others

Semaphore Summary

- Semaphores can be used to solve any of the traditional synchronization problems
- However, they have some drawbacks
  - They are essentially shared global variables
    - Can potentially be accessed anywhere in program
  - No connection between the semaphore and the data being controlled by the semaphore
  - Used both for critical sections (mutual exclusion) and coordination (scheduling)
    - Note that I had to use comments in the code to distinguish
  - No control or guarantee of proper usage
- Sometimes hard to use and prone to bugs
  - Another approach: Use programming language support