Synchronization

Dr. Yingwu Zhu

Synchronization

- Threads cooperate in multithreaded programs
  - To share resources, access shared data structures
    - Threads accessing a memory cache in a Web server
  - To coordinate their execution
    - One thread executes relative to another (recall ping-pong)
- 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 interleavings of thread executions
- Discuss in terms of threads, also applies to processes

The Problem with Concurrent Execution

- Concurrent threads (& processes) often access shared data and resources
  - Need controlled access to the shared data; otherwise result in an inconsistent view of this data
- Maintaining data consistency must ensure orderly execution of cooperating processes
- We will look at
  - Mechanisms to control access to shared resources
    - Locks, mutexes, semaphores, monitors, condition variables, …
  - Patterns for coordinating accesses to shared resources
    - Bounded buffer, producer-consumer, etc.

Classic 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 significant other 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:

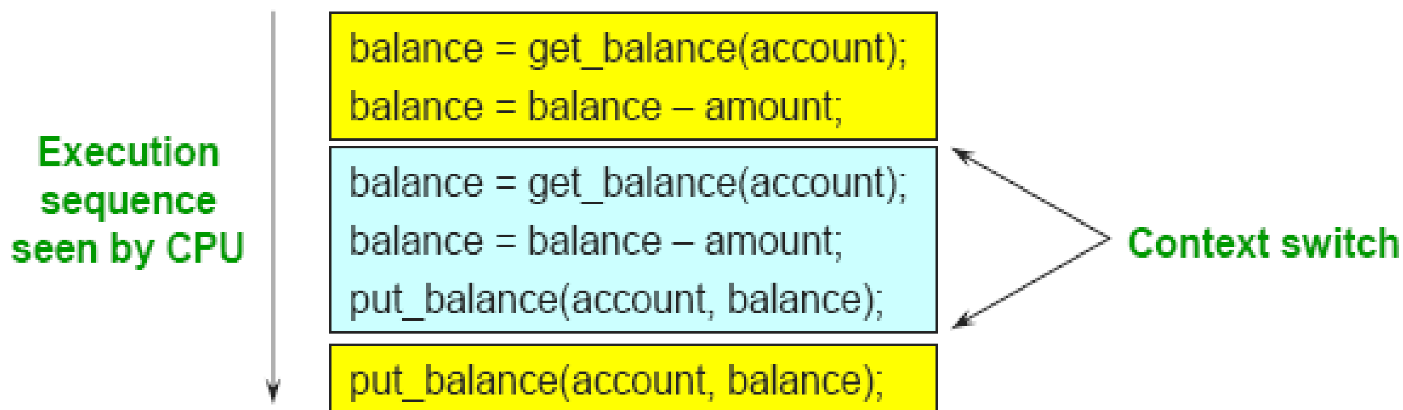
```
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

```
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:



- What is the balance of the account now?
- Is the bank happy with our implementation?

Shared Resources

- The problem is that two concurrent threads (or processes) accessed a shared resource (account) without any synchronization
  - Known as a race condition (memorize this buzzword)
- We need mechanisms to control access to these shared resources in the face of concurrency
  - 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.

When Are Resources Shared?

- Local variables are not shared (private)
  - Refer to data on the stack
  - Each thread has its own stack
  - Never pass/share/store a pointer to a local variable on another thread's stack!
- Global variables and static objects are shared
  - Stored in the static data segment, accessible by any thread
- Dynamic objects and other heap objects are shared
  - Allocated from heap with malloc/free or new/delete

Race Condition

- Multiple processes manipulate same data concurrently
- The outcome of execution depends on the particular order in which the data access takes place

Race Condition: Example

- count++ could be implemented as

  register1 = count
  register1 = register1 + 1
  count = register1

- count-- could be implemented as

  register2 = count
  register2 = register2 - 1
  count = register2

- Consider this execution interleaving with "count = 5" initially:

  S0: producer execute register1 = count {register1 = 5}
  S1: producer execute register1 = register1 + 1 {register1 = 6}
  S2: consumer execute register2 = count {register2 = 5}
  S3: consumer execute register2 = register2 - 1 {register2 = 4}
  S4: producer execute count = register1 {count = 6 }
  S5: consumer execute count = register2 {count = 4}

Mutual Exclusion & Critical-Section

- We want to use mutual exclusion to synchronize access to shared resources
- Code that uses mutual exclusion to synchronize its execution is 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

Critical-Section Problem

Each process looks like:

do {

CRITICAL SECTION

REMAINDER SECTION

} while (TRUE);

Solution to Critical-Section Problem

MUST satisfy the following three/four requirements:

1. Mutual Exclusion -- If one thread is in the critical section, then no other is
2. Progress - 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 - 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

Locks

- While one thread executes “withdraw”, we want some way to prevent other threads from executing in it
- Locks are one way to do this
- 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?
- Locks can spin (a spinlock) or block (a mutex)

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);
```

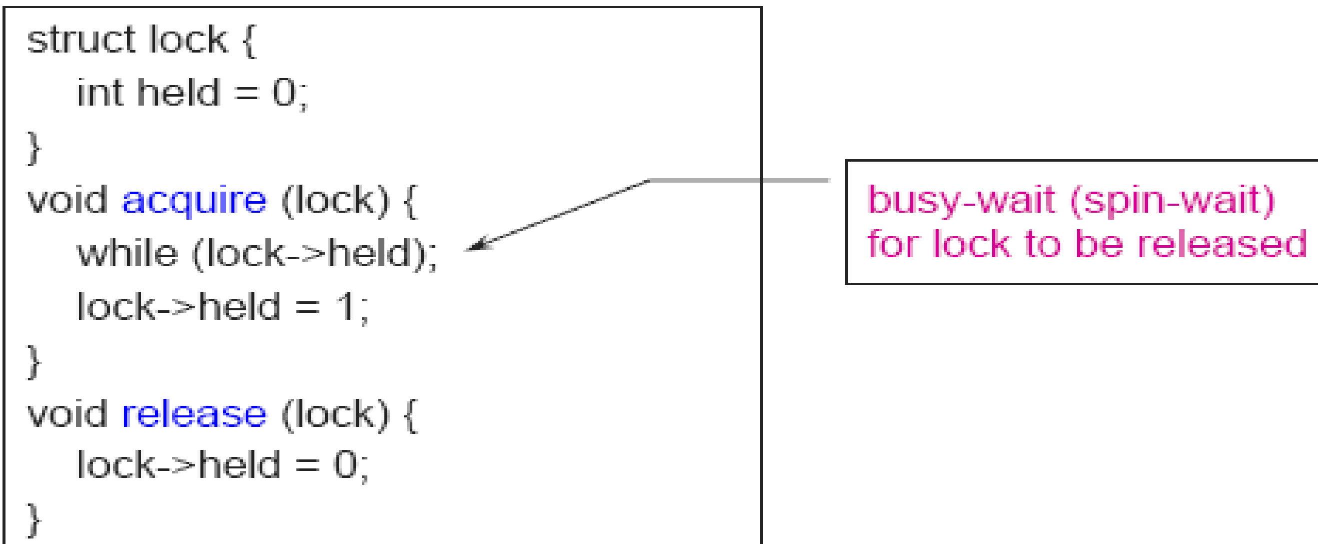
```
put_balance(account, balance);
release(lock);
```

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

- What happens when blue tries to acquire the lock?
- Why is the “return” outside the critical section? Is this ok?

Implementing Locks

- How do we implement locks? Here is one attempt:



- This is called a spinlock because a thread spins waiting for the lock to be released
- Does this work?

Implementing Locks

- No. Two independent threads may both notice that a lock has been released and thereby acquire it.

```
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (lock->held);
    lock->held = 1;
}
void release (lock) {
    lock->held = 0;
}
```

A context switch can occur here, causing a race condition

Implementing Locks

- The problem is that the implementation of locks has critical sections, too
- How do we stop the recursion?
- The implementation of acquire/release must be **atomic**
  - An atomic operation is one which executes as though it could not be interrupted
  - Code that executes "**all or nothing**"
- How do we make them atomic?
- Need help from hardware
  - Atomic instructions (e.g., test-and-set)
  - Disable/enable interrupts (prevents context switches)

Hardware Solution – Test and Set

- Test and modify the content of a word atomically

```
boolean TestAndSet (boolean
    *flag)
{
        boolean old = *flag;
        *flag = true;
        return old;
}
```

Implement locks using test and set

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

Problems with Spinlocks

- The problem with spinlocks is that they are wasteful
  - If a thread is spinning on a lock, then the thread holding the lock cannot make progress
- How did the lock holder give up the CPU in the first place?
  - Lock holder calls yield or sleep
  - Involuntary context switch
- Only want to use spinlocks as primitives to build higher-level synchronization constructs

Hardware Solution – Disable Interrupts

- Correct solution for uni-processor machine
  - Atomic instructions
- During critical section, multiprogramming is not utilized → perf. penalty
- Too inefficient for multi-processor machines, interrupt disabling message passing to all processors, delay entrance to critical section

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

On Disabling Interrupts

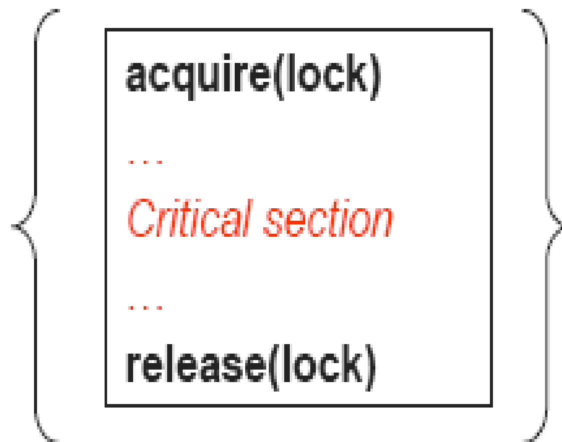
- Disabling interrupts blocks notification of external events that could trigger a context switch (e.g., timer)
  - This is what Nachos uses as its primitive
- In a "real" system, this is only available to the kernel
  - Why?
  - What could user-level programs use instead?
- Disabling interrupts is insufficient 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



Disabling Interrupts:

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

PART 2: High-Level Synchronization

High-Level Synchronization

- Spinlocks and disabling interrupts are useful only for very short and simple critical sections
  - Wasteful otherwise
  - These primitives are "primitive" – don't do anything besides mutual exclusion

High-Level Synchronization

- We looked at using locks to provide mutual exclusion
- Locks work, but they have some drawbacks when critical sections are long
  - Spinlocks – inefficient
  - Disabling interrupts – can miss or delay important events
- Instead, we want synchronization mechanisms that
  - Block waiters
  - Leave interrupts enabled inside the critical section
- Look at two common high-level mechanisms
  - Semaphores: binary (mutex) and counting
  - Monitors: mutexes and condition variables
- Use them to solve common synchronization problems

Semaphores

- Semaphores are another data structure that provides mutual exclusion to critical sections
  - Block waiters, interrupts enabled within CS
  - Described by Dijkstra in THE system in 1968
- Semaphores can also be used as atomic counters
  - More later
- Semaphores 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()

Blocking in Semaphores

- Associated with each semaphore is a queue of waiting processes
- When wait() is called by a thread:
  - If semaphore is open, thread continues
  - If semaphore is closed, thread blocks on queue
- Then signal() opens the semaphore:
  - If 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
    - In other words, signal() has “history” (c.f. condition vars later)
    - This “history” is a counter

Semaphore Types

- Counting semaphore – integer value can range over an unrestricted domain
  - Used to control access to a given resource consisting of a finite number of instances
  - The semaphore is initialized to the number of resources available
  - Multiple threads can pass the semaphore
- Binary semaphore – integer value can range only between 0 and 1;
  - Also known as mutex locks
  - Represents single access to a resource
  - Guarantees mutual exclusion to a critical section

Semaphore Implementation

- Implementation of wait:

```
wait (S){
        value--;
        if (value < 0) {
                add this thread T to waiting queue
                block(P);
        }
}
```

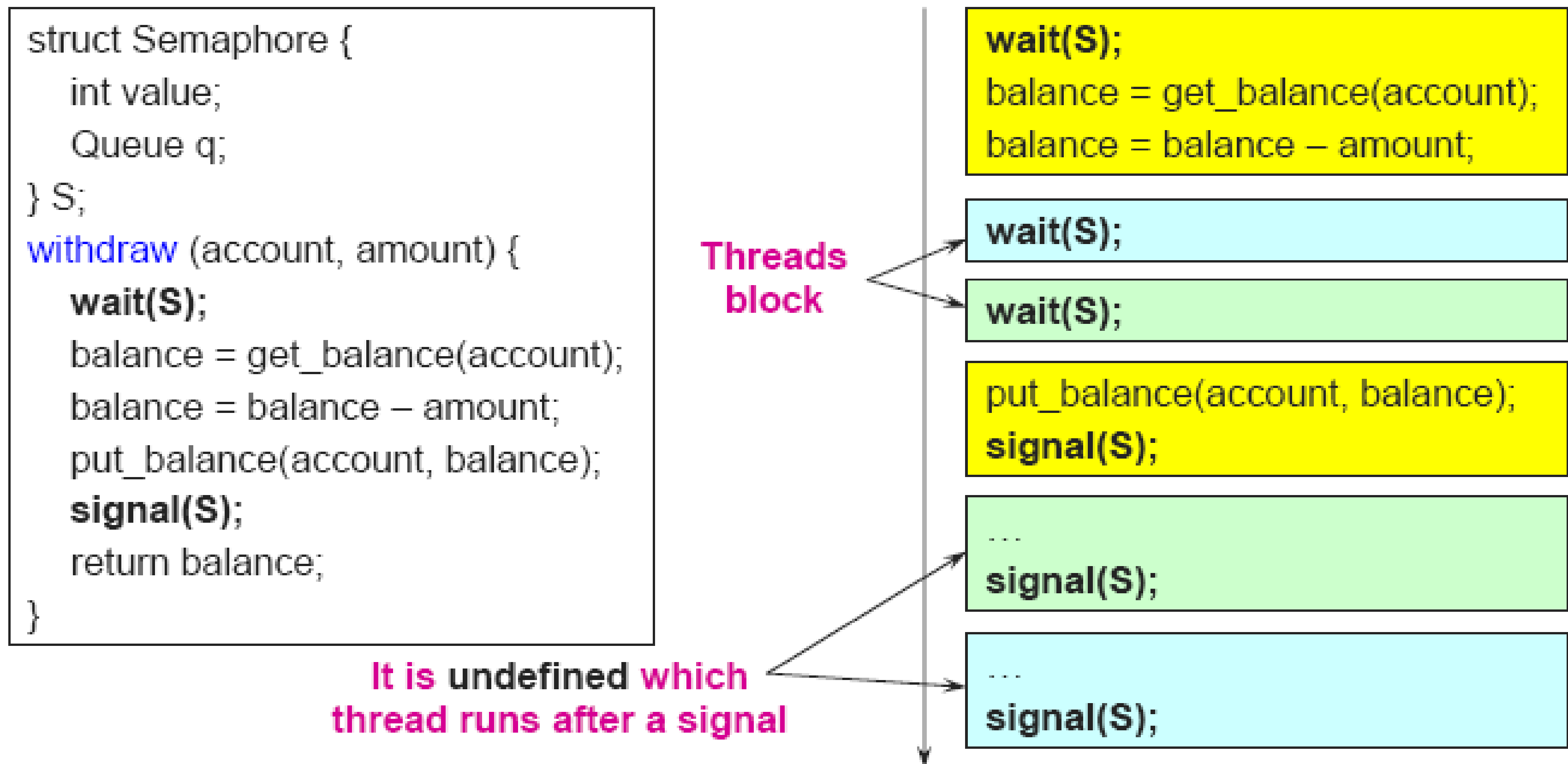
```
Struct Semaphore {
        int value;
        Queue q;
} S;
```

- Implementation of signal:

```
signal (S){
          value++;
          if (value <= 0) {
                remove a thread T from the waiting queue
                wakeup(P);
          }
}
```

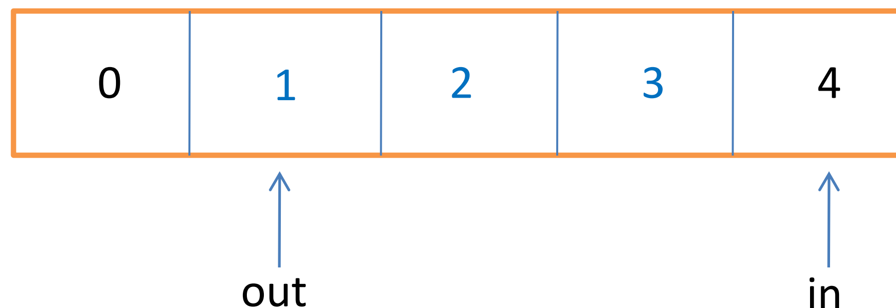
Using Semaphores

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



Producer-Consumer: Bounded Buffer

- Problem: There is a set of resource buffers shared by producer and consumer threads
- Producer inserts resources into the buffer set
  - Output, disk blocks, memory pages, processes, etc.
- Consumer removes resources from the buffer set
  - Whatever is generated by the producer
- Producer and consumer execute at different rates
- Cyclic buffer:



Using Semaphores

- Use three semaphores:
- mutex – mutual exclusion to shared set of buffers
  - Binary semaphore
- empty – count of empty buffers
  - Counting semaphore
- full – count of full buffers
  - Counting semaphore

Producer-Consumer: bounded buffer

Initialization: semaphores: mutex = 1, full = 0; empty = N;
integers: int = 0, out = 0;

```
void append(int d) {
  buffer[in] = d;
  in = (in + 1) % N;
}


int take() {
  int x = out;
  out = (ouit+1) %N;
  return buffer[x];
}
```

Producer:

```
While (1) {
  produce x;
  wait(empty);
  wait(mutex);
  append(x);
  signal(full);
  signal(mutex);
}
```

Consumer:

```
While (1) {
  wait(full);
  wait(mutex);
  x = take();
  signal(empty);
  signal(mutex);
  consume x;
}
```

What happens if operations on mutex and full/empty are switched around?

Readers-Writers Problem

- A data set is shared among a number of concurrent threads
  - Readers – only read the data set; they do not perform any updates
  - Writers – can both read and write.
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time (if on writer access it, no readers or other writers are allowed to access the data).
- Shared Data
  - Data set
- Using semaphores
  - What semaphores do we need?

Readers-Writers Problem

- Semaphore mutex initialized to 1, control access to readcount.
- Semaphore wrt initialized to 1.
- Integer readcount initialized to 0.

Readers-Writers Problem (Cont.)

- Exercise: Implement readers and writers

Readers-Writers Problem

Initialization: semaphores: mutex = 1, wrt = 1;
integers: readcount = 0;

```
Reader:
{
    wait(mutex);
    readcount++;
    If (readcount==1)
       wait(wrt);
    signal(mutex);
    read;
    wait(mutex);
    readcount--;
    If(readcount==0)
       signal(wrt);
    signal(mutex);
}
```

```
Writer:
{
    wait(wrt);
    write;
    signal(wrt);
}
```

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)
  - No control or guarantee of proper usage
- Sometimes hard to use and prone to bugs
  - Another approach: Use programming language support

Monitor

- A monitor is a programming language construct that controls access to shared data
  - Synchronization code added by compiler, enforced at runtime
  - Why is this an advantage?
- A monitor is a module that encapsulates
  - Shared data structures
  - Procedures that operate on the shared data structures
  - Synchronization between concurrent threads that invoke the procedures
- A monitor protects its data from unstructured access
- It guarantees that threads accessing its data through its procedures interact only in legitimate ways

Monitor Semantics

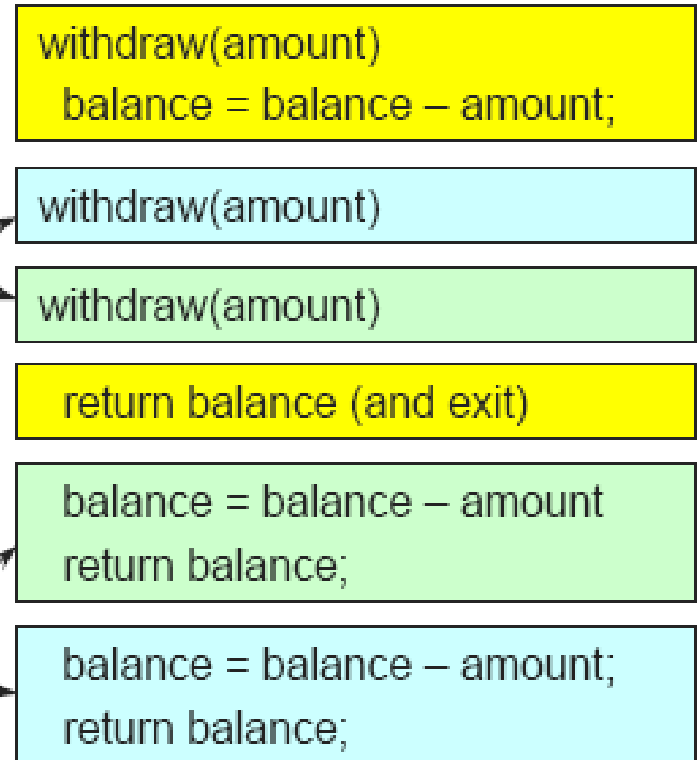
- A monitor guarantees mutual exclusion
  - Only one thread can execute any monitor procedure at any time (the thread is "in the monitor")
  - If a second thread invokes a monitor procedure when a first thread is already executing one, it blocks
    - So the monitor has to have a wait queue…
  - If a thread within a monitor blocks, another one can enter
- What are the implications in terms of parallelism in monitor?

Account Example Again

```
Monitor account {
  double balance;

  double withdraw(amount) {
    balance = balance – amount;
    return balance;
  }
}
```

Threads block waiting to get into monitor



When first thread exits, another can enter. Which one is undefined.

- Hey, that was easy
- But what if a thread wants to wait inside the monitor?
  - Such as “mutex(empty)” by reader in bounded buffer?

Condition Variables

- Condition variables provide a mechanism to wait for events (a “rendezvous point”)
  - Resource available, no more writers, etc.
- Condition variables support three operations:
  - Wait – release monitor lock, wait for C/V to be signaled
    - So condition variables have wait queues, too
  - Signal – wakeup one waiting thread
  - Broadcast – wakeup all waiting threads
- Note: Condition variables are not boolean objects
  - “if (condition_variable) then” … does not make sense
  - “if (num_resources == 0) then wait(resources_available)” does
  - An example will make this more clear

Monitor Bounded Buffer

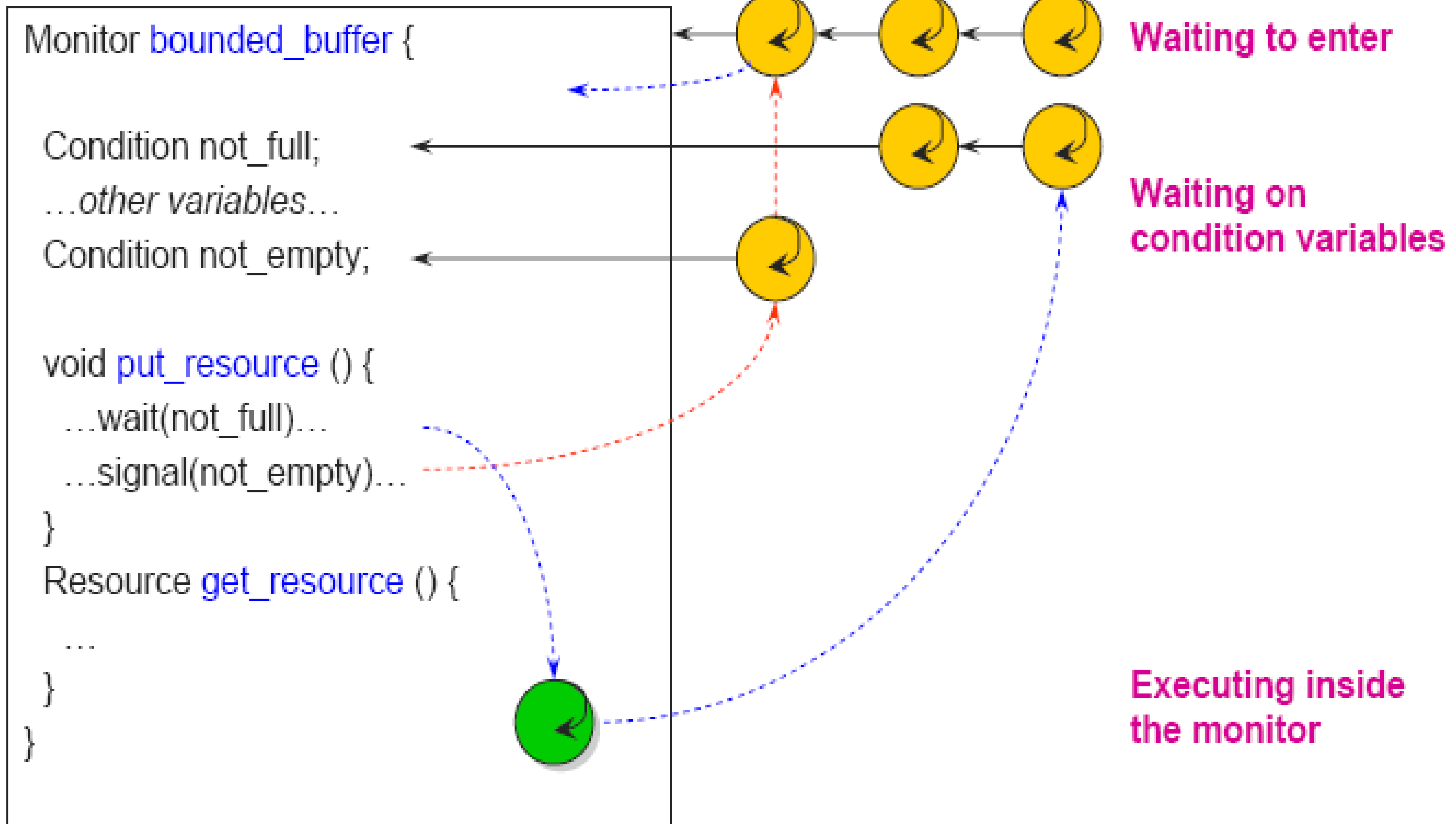
```
Monitor bounded_buffer {
  Resource buffer[N];
  // Variables for indexing buffer
  Condition not_full, not_empty;

  void put_resource (Resource R) {
    while (buffer array is full)
        wait(not_full);
    Add R to buffer array;
    signal(not_empty);
  }
```

```
  Resource get_resource() {
    while (buffer array is empty)
        wait(not_empty);
    Get resource R from buffer array;
    signal(not_full);
    return R;
  }
} // end monitor
```

- What happens if no threads are waiting when signal is called?

Monitor Queues



Condition Variables != Semaphores

- Condition variables != semaphores
  - Although their operations have the same names, they have entirely different semantics (
  - However, they each can be used to implement the other
- Access to the monitor is controlled by a lock
  - wait() blocks the calling thread, and gives up the lock
    - To call wait, the thread has to be in the monitor (hence has lock)
    - Semaphore::wait just blocks the thread on the queue
  - signal() causes a waiting thread to wake up
    - If there is no waiting thread, the signal is lost
    - Semaphore::signal increases the semaphore count, allowing future entry even if no thread is waiting
    - Condition variables have no history

Signal Semantics

- There are two flavors of monitors that differ in the scheduling semantics of signal()
  - Hoare monitors (original)
    - signal() immediately switches from the caller to a waiting thread
    - The condition that the waiter was anticipating is guaranteed to hold when waiter executes
    - Signaler must restore monitor invariants before signaling
  - Mesa monitors (Mesa, Java)
    - signal() places a waiter on the ready queue, but signaler continues inside monitor
    - Condition is not necessarily true when waiter runs again
      - Returning from wait() is only a hint that something changed
      - Must recheck conditional case

Hoare vs. Mesa Monitors

- Hoare

```
if (empty)
    wait(condition);
```

- Mesa

```
while (empty)
    wait(condition);
```

- Tradeoffs
  - Mesa monitors easier to use, more efficient
    - Fewer context switches, easy to support broadcast
  - Hoare monitors leave less to chance
    - Easier to reason about the program

Condition Variables vs. Locks

- Condition variables are also used without monitors in conjunction with blocking locks
- A monitor is “just like” a module whose state includes a condition variable and a lock
  - Difference is syntactic; with monitors, compiler adds the code
- It is “just as if” each procedure in the module calls acquire() on entry and release() on exit
  - But can be done anywhere in procedure, at finer granularity
- With condition variables, the module methods may wait and signal on independent conditions

Using Condition Variables & Locks

- Alternation of two threads (ping-pong)
- Each executes the following:

```
Lock lock;
Condition cond;

void ping_pong () {
  acquire(lock);
  while (1) {
     printf("ping or pong\n");
     signal(cond, lock);
     wait(cond, lock);
  }
  release(lock);
}
```

Must acquire lock before you can wait (similar to needing interrupts disabled to call Sleep in Nachos)

Wait atomically releases lock and blocks until signal()

Wait atomically acquires lock before it returns

Monitors and Java

- A lock and condition variable are in every Java object
  - No explicit classes for locks or condition variables
- Every object is/has a monitor
  - At most one thread can be inside an object's monitor
  - A thread enters an object's monitor by
    - Executing a method declared "synchronized"
    - Can mix synchronized/unsynchronized methods in same class
  - Executing the body of a "synchronized" statement
    - Supports finer-grained locking than an entire procedure
    - Identical to the Modula-2 "LOCK (m) DO" construct
- Every object can be treated as a condition variable
  - Object::notify() has similar semantics as Condition::signal()

Summary

- Semaphores
  - wait()/signal() implement blocking mutual exclusion
  - Also used as atomic counters (counting semaphores)
  - Can be inconvenient to use
- Monitors
  - Synchronizes execution within procedures that manipulate encapsulated data shared among procedures
  - Only one thread can execute within a monitor at a time
  - Relies upon high-level language support
- Condition variables
  - Used by threads as a synchronization point to wait for events
  - Inside monitors, or outside with locks