Operating Systems (A)
(Honor Track)

Lecture 15: Synchronization (II)

Yao Guo (郭耀)

Peking University

Fall 2021

Acknowledgements: Prof. Xiangqun Chen & Tao Wang at PKU and Prof. Yuanyuan Zhou at UCSD

Review

Synchronization Basics

Why do we need synchronizations

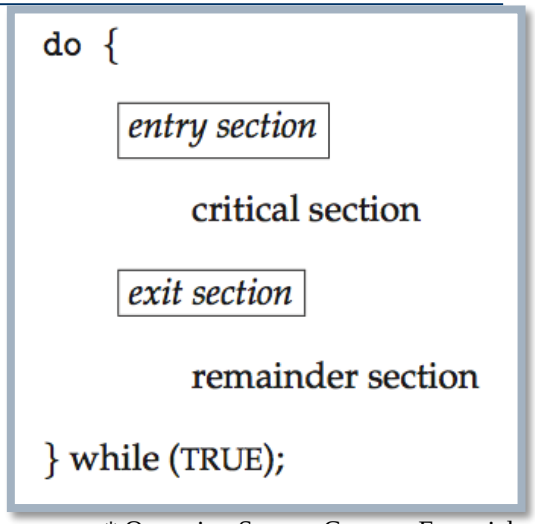
Critical sections

Mutual Exclusion

- We want to use mutual exclusion to synchronize access to shared resources
  - This allows us to have larger atomic blocks
- Code that uses mutual exclusion (and…) 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
- What requirements would you place on a critical section?

Critical Section Requirements

- Mutual exclusion (mutex)
  - If one thread is in the critical section, then no other is
- Progress
  - If some thread T is not in the critical section, then T cannot prevent some other thread S from entering the critical section
  - A thread in the critical section will eventually leave it
- Bounded waiting (no starvation)
  - If some thread T is waiting on the critical section, then T will eventually enter the critical section
- Performance
  - The overhead of entering and exiting the critical section is small with respect to the work being done within it



* Operating System Concept Essentials (Abrahan Silberschatz et al.), P220

How to Build Critical Sections

- Mechanisms for building critical sections
  - Atomic read/write
  - Locks
  - Semaphores
  - Monitors
  - Messages

Peterson's Algorithm

```
int turn = 1;
bool ready1 = false, ready2 = false;
```

```
while (true) {
    ready1 = true;
    turn = 2;
    while (turn == 2 && ready2) ;
    critical section
    ready1 = false;
    outside of critical section
}
```

```
while (true) {
    ready2 = true;
    turn = 1;
    while (turn == 1 && ready1) ;
    critical section
    ready2 = false;
    outside of critical section
}
```

- Does it work?
  - Mutual exclusion
  - Progress
  - Bounded waiting

Peterson's Algorithm (Figure 2-24)

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
    int other;                          /* number of the other process */

    other = 1 – process;                /* the opposite of process */
    interested[process] = TRUE;         /* show that you are interested */
    turn = process;                     /* set flag */
    while (turn == process && interested[other] == TRUE)  /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
    interested[process] = FALSE;        /* indicate departure from critical region */
}
```

This Lecture

Common Synchronization Mechanisms

Locks

Semaphores

Monitors

Buzz Words

- Lock
- Spinlock
- Semaphore
- Starvation
- Monitor
- Condition variable

How to Build Critical Sections

- Mechanisms for building critical sections
  - Atomic read/write
  - Locks
  - Semaphores
  - Monitors
  - Messages

Locks

- A lock is an object in memory providing two operations
  - acquire(): before entering the critical section
  - release(): after leaving a critical section
  - Some systems may refer this pair as Lock() / Unlock()
- 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? What if not?
- What happens when a third thread calls acquire?

Implementing Locks (1/3)

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

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

busy-wait (spin-wait) for lock to be released

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

Implementing Locks (2/3)

- 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 (3/3)

- 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
- How do we make them atomic?
- Need help from hardware
  - Atomic instructions (e.g., test-and-set)
  - Disable/enable interrupts (prevents context switches)

Atomic Instructions: Test-And-Set

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

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

- 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 work?
- Does it work on multiprocessors?

Other Similar Hardware Instruction

- Swap/XCHG

```
void Swap (char* x,* y);
{ // All done atomically
      char temp = *x;
      *x = *y;
      *y = temp
}
```

- How to implement Test-And-Set using Swap/XCHG?

```
bool  test_and_set (bool *flag)
{
      bool X=True;
      Swap(&X, &flag);
      return X;
}
```

Problems with Spinlocks

- The problem with spinlocks is that they are wasteful
  - If a thread is spinning on a lock, it occupies the CPU, and the other thread holding the lock is not executing to get out of the critical section and release the lock
- Solution:
  - If a thread cannot get the lock, call thread_yield to give up the CPU!
- Solution 2: sleep and wakeup
  - When blocked, go to sleep
  - Wakeup when it is OK to retry entering the critical section

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
  - There is nothing saying you are locked or not; no lock->held
- Can two threads disable interrupts simultaneously in one processor core?
- Does it work on multi-processors?

On Disabling Interrupts

- Disabling interrupts blocks notification of external events that could trigger a context switch (e.g., timer)

- Only use disabling interrupts to implement higher-level synchronization primitives
  - Don’t want interrupts disabled between acquire and release

Issue with the current solutions

- 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**
- **Longer the CS, greater the chance for lock holder to be interrupted**

```
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)**

Do we have any solution?

Solution

- Implementing locks with interrupts enabled in critical sections

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

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

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

Interrupts Disabled

Interrupts Enabled

Interrupts Disabled

Another Issue of Disabling Interrupts

- Disabling interrupts is insufficient on a multiprocessor
- So we are back to atomic instructions

This Lecture

Common Synchronization Mechanisms

Locks

Semaphores

Monitors

Higher-Level Synchronization

- We looked at using locks to provide mutual exclusion
- Those locks work, but they have some drawbacks when critical sections are long
  - Spinlocks – inefficient
  - Disabling interrupts – can miss or delay important events, and do not work on multicores.
- Instead, we want synchronization mechanisms that
  - Block waiters
  - Leave interrupts enabled inside the critical section
- Look at some common high-level mechanism
  - Semaphores: binary (mutex) and counting

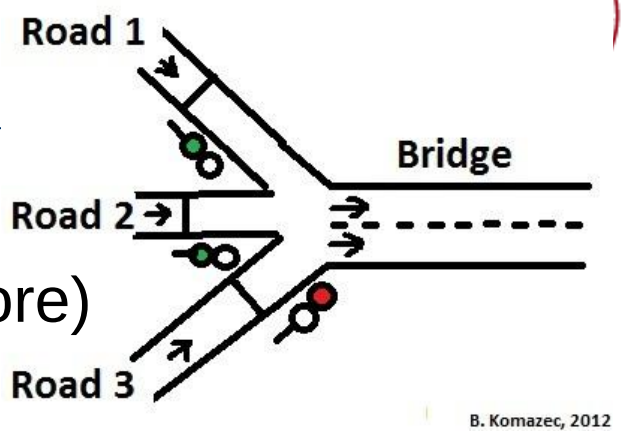
Semaphores

- Semaphores are an abstract data type that provides mutual exclusion to critical sections
- Semaphores are **integers** that support two operations:
  - P(semaphore): decrement, block until semaphore is open
    - Also Wait(), or Down()
  - V(semaphore): increment, allow another thread to enter
    - Also Signal(), or Up()
  - That's it! No other operations – not even just reading its value – exist
- Semaphore safety property: the semaphore value is always greater than or equal to 0
- Semaphores can also be used as atomic counters

Blocking in Semaphores

- Associated with each semaphore is a queue of waiting processes
- When P() is called by a thread:
  - If semaphore is open, thread continues
  - If semaphore is closed, thread blocks on queue
- Then V() 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, V() has “history”
    - This “history” is a counter

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)
  - Represents a resource with many units available, or a resource that allows certain kinds of unsynchronized concurrent access (e.g., reading)
  - Multiple threads can pass the semaphore
  - Number of threads determined by the semaphore “count”
    - mutex has count = 1, counting has count = N
- You can use one type to implement the other
- Does a counting semaphore implement a critical section?

Semaphore Semantics

- NOT the semaphore's implementation
  - They should be atomic
  - Any better idea than busy waiting?

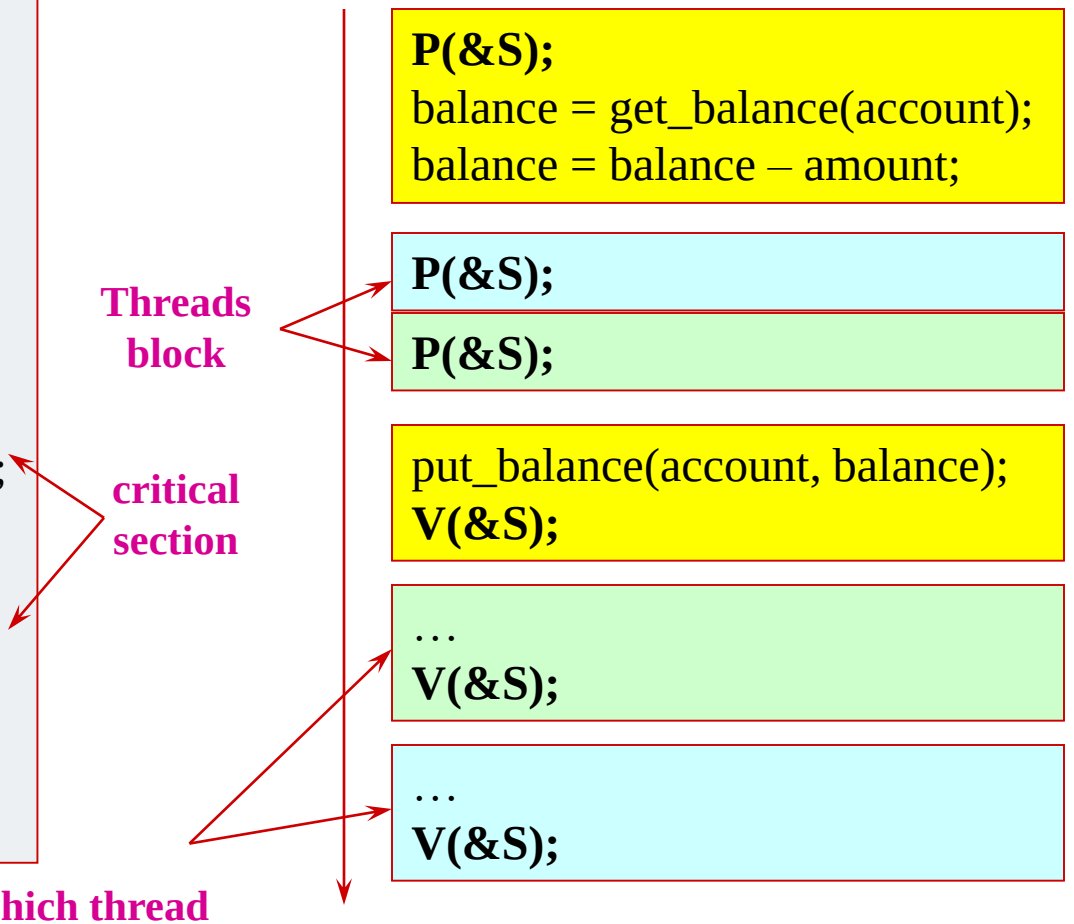
```
void P (int *semaphore) {
    while (*semaphore<=0);
    *semaphore--;
}
```

```
void V (int *semaphore) {
    *semaphore++;
}
```

Using Semaphores

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

```
struct Semaphore {
    int value;
    Queue q;
} S;
withdraw (account, amount) {
    …
    P(&S);
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    V(&S);
    return balance;
}
```



It is **undefined** which thread runs after the V

Possible Deadlocks with Semaphores

Example:

P0 **P1**

share two mutex semaphores S and Q

S:= 1; Q:=1;

P(S);	**P(Q);**
P(Q);	**P(S);**
……	**……**
V(S);	**V(Q);**
V(Q);	**V(S);**

Deadlock?

Be Careful When Using Semaphores


```
// Violation of Mutual Exclusion
V(mutex);
critical section
P(mutex);

// Deadlock Situation
P(mutex);
critical section
P(mutex);

// Violation of Mutual Exclusion
critical section
V(mutex);
```

A Classic Synchronization Problem

- We've looked at a simple example for using synchronization
  - Mutual exclusion while accessing a bank account
- Now we're going to use semaphores to look at a more interesting example
  - Readers/Writers

Readers/Writers Problem

- http://en.wikipedia.org/wiki/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

- How can we use semaphores to control access to the object to implement this protocol?

First Attempt

```
Semaphore w_or_r=1;

Reader{
     P(w_or_r); // lock out writers
     read;
     V(w_or_r); // up for grabs
}

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

Does it work?

Why?

Second Attempt

```
Semaphore w_or_r=1;
int readcount; //record #readers

Reader{
    readcount++;
    if (readcount == 1){
          P(w_or_r); // lock out writers
    }
   read;
   readcount--;
    if (readcount == 0){
        V(w_or_r); // up for grabs
     }
}

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

Does it work?

Why?

Is readcount a shared data?
Who protects it?

Readers/Writers: Real Solution

- Use three variables
  - int readcount – number of threads reading object
  - Semaphore mutex – control access to readcount
  - Semaphore w_or_r – exclusive writing or reading

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

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

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

Readers/Writers Notes

- w_or_r provides mutex between readers and writers, and also between multiple writers
- Why do readers use mutex (binary semaphore)?

- What if V() is above "if (readcount == 1)"?
- Why do we need "if(readcount==1)"?
- Why do we need "if(readcount==0)"?

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

But It Still Has a Problem

- What if a writer is waiting, but readers keep coming?

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

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

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

Problem: Starvation

- What if a writer is waiting, but readers keep coming, the writer is starved!

- How to write a solution without starvation?

Thoughts on Semaphores

- Semaphores can be used to solve 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

This Lecture

Common Synchronization Mechanisms

Locks

Semaphores

Monitors

Monitors

- 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 unsynchronized access
- It guarantees that threads accessing its data through its procedures interact only in legitimate ways

```
Monitor account {
   double balance;

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

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’s the difference with critical sections we’ve learnt?
- What are the implications in terms of parallelism in monitor?

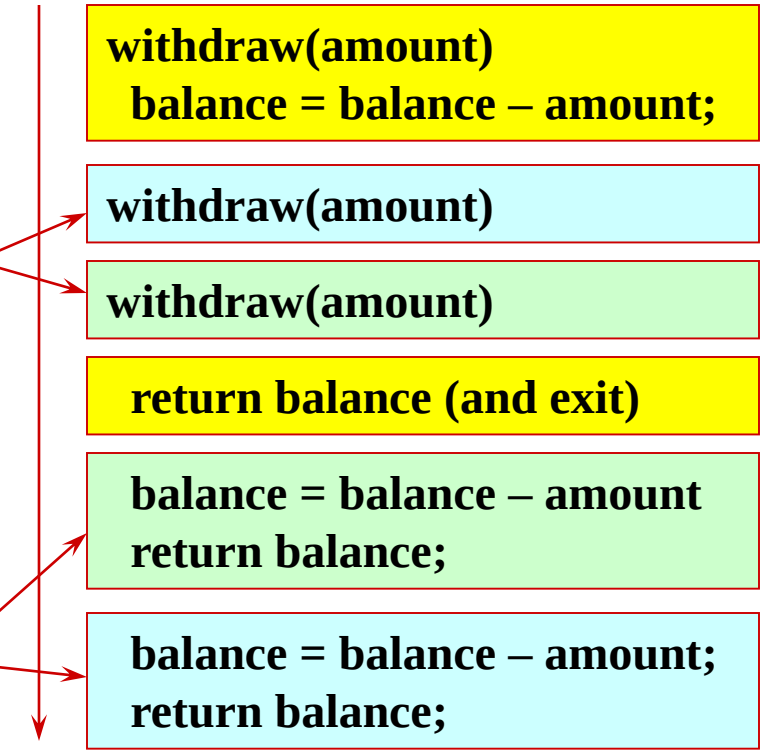
Account Example

```
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?

Condition Variables

- wait(condition, lock), signal(condition)
- Condition variables are NOT conditions
  - So don't EVER do this:
    - if (condition variable){

      …

      }
- Instead, it is a way for one thread to wait (if some resource is not available), and some other thread to wake it up once the resource becomes available
  - Sleep
  - Wake (also called as "signal")
  - Wakeall (or "signalAll")

Condition Variable & Lock

- Condition variable doesn't replace lock, instead it complements lock
- wait(condition, lock)
  - First release the lock, put the thread into the queue of the condition, if waking up, re-acquiring the lock
  - Once sleep returns, it is awaken by some other thread, and it also holds the lock
- signal(condition)
  - Wake up a thread waiting on the condition (queue)
- broadcast(condition)
  - Wake all the thread waiting on the condition (queue)

Condition Variables inside Monitors (1/2)

- A condition variable inside a monitor is associated with a condition needed for a thread to make progress once it is in the monitor.

```
Monitor M {
    ... monitored variables
    Condition c, d;

    void enter_mon (...) {
        if (extra property C not true) wait(c);  // waits outside of the monitor's mutex
        do what you have to do
        if (extra property true D) signal(d);   // brings in one thread waiting on d
    }
```

Condition Variables inside Monitors (2/2)

- 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

- 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 clearer

Bounded Buffer Problem

- 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
  - No serialization of one behind the other
  - Tasks are independent (easier to think about)
  - The buffer set allows each to run without explicit handoff
- Safety:
  - If *nc* is number consumed, *np* number produced, and N the size of the buffer, then $0 \leq np - nc \leq N$

Monitor Bounded Buffer

```
Monitor bounded_buffer {
    Resource buffer[N];
    // Variables for indexing buffer
    // monitor invariant involves these vars
    Condition not_full; // space in buffer
    Condition not_empty; // value in buffer

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

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

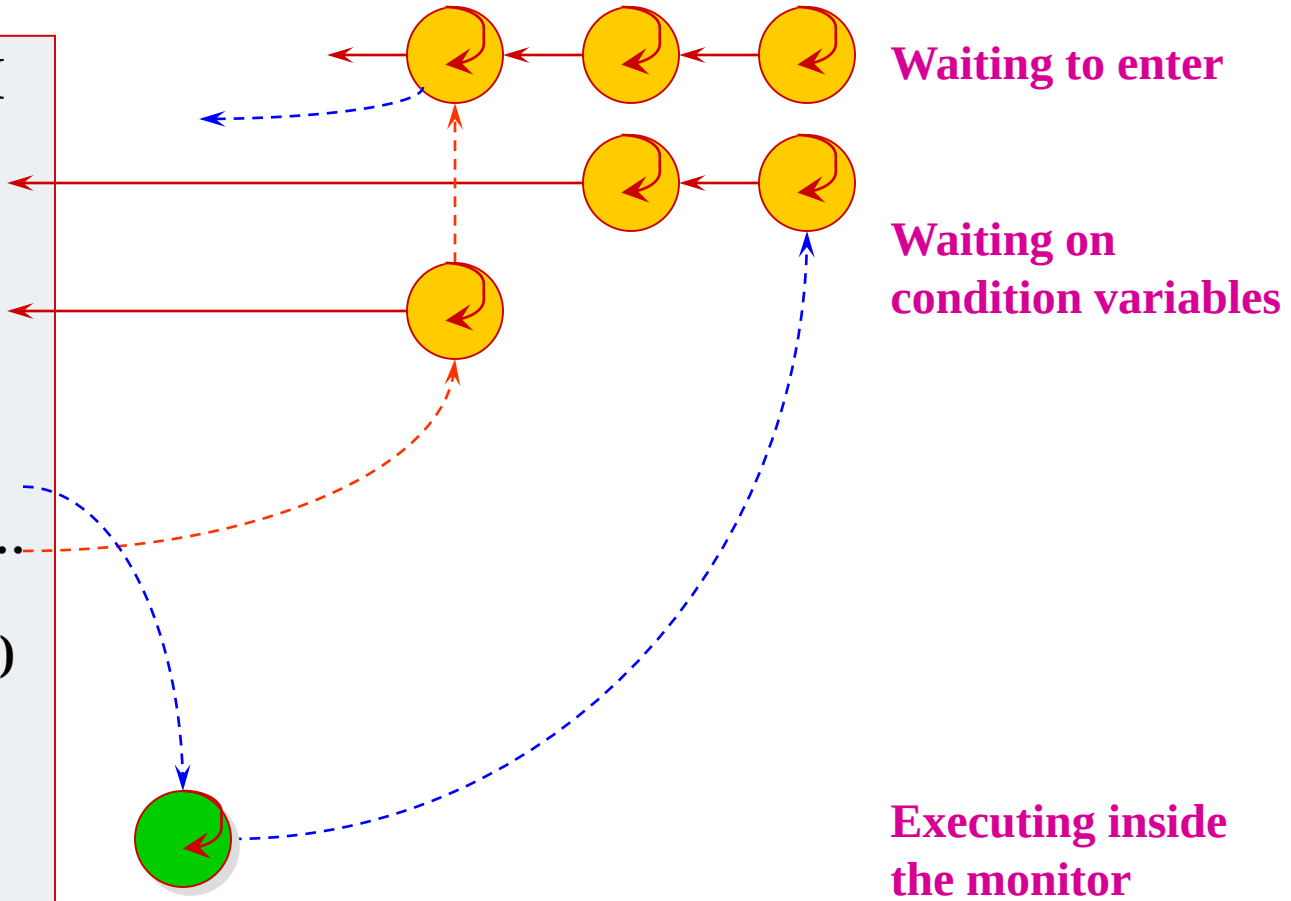
- What happens if no threads are waiting when signal is called?
  - Signal is lost

Monitor Queues

```
Monitor bounded_buffer {

    Condition not_full;
    …other variables…
    Condition not_empty;

    void put_resource () {
        …wait(not_full)…
        …signal(not_empty)…
    }
    Resource get_resource ()
{
        …wait(not_empty)…
        …signal(not_full)…
    }
}
```



Condition Vars != Semaphores

- Monitor with condition variables != semaphores
  - But they can implement each 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::P() 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::V() 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
  - 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

Monitor: Readers and Writers (1/2)

Using Mesa monitor semantics.

- Will have four methods: StartRead, StartWrite, EndRead and EndWrite
- Monitored data: nr (number of readers) and nw (number of writers) with the monitor invariant
- Two conditions:
  - canRead: nw = 0
  - canWrite: (nr = 0) ∧ (nw = 0)

Monitor: Readers and Writers (2/2)

```
Monitor RW {
    int nr = 0, nw = 0;
    Condition canRead, canWrite;

    void StartRead () {
        while (nw != 0) do wait(canRead);
        nr++;
    }

    void EndRead () {
        nr--;
        if (nr==0) signal(canWrite)
    }

    void StartWrite {
        while (nr != 0 || nw != 0) do wait(canWrite);
        nw++;
    }

    void EndWrite () {
        nw--;
        signal(canWrite);
        broadcast(canRead);
    }
} // end monitor
```

- Who protects accesses to nr and nw?
- Is there any priority between readers and writers? Any issue?
  - Starvation
- Why do we need “broadcast” in EndWrite?

Homework

1. Investigation Reports
   - Survey how locks are implemented in Linux (both for uniprocessor and multiprocessors)
   - Survey how semaphores are implemented in Linux (both for uniprocessor and multiprocessors)
2. Exercise 2-57
3. Exercise 2-60
4. Exercise 2-62

- Present them in class
  - Student sign-up in WeChat group.
  - One student each question.

Summary

- Locks
- Semaphores
- Monitors and condition variables

- Next lecture: Discussions on Synchronization