CSE 120
Principles of Operating Systems

Spring 2017

Condition Variables and 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 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 a monitor?

Account Example

```
Monitor account {
  double balance;

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

Threads block waiting to get into monitor

withdraw(amount)
balance = balance – amount;

withdraw(amount)

withdraw(amount)

return balance (and exit)

balance = balance – amount
return balance;

balance = balance – amount;
return balance;

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?

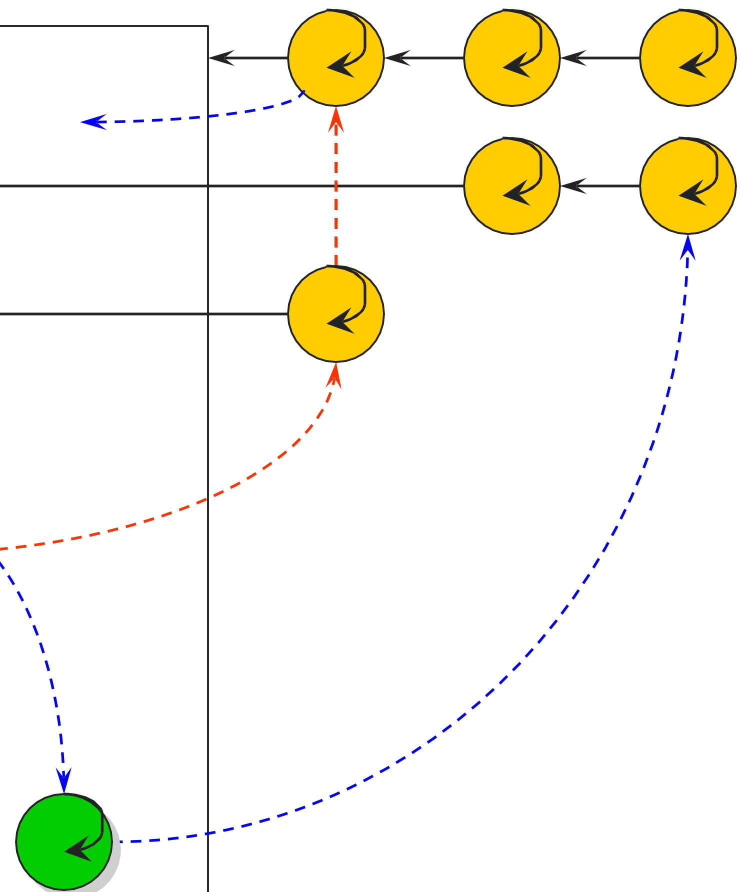
Monitors, Monitor Invariants and Condition Variables

- A monitor invariant is a safety property associated with the monitor, expressed over the monitored variables. It holds whenever a thread enters or exits the monitor.
- A condition variable is associated with a condition needed for a thread to make progress once it is in the monitor.

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

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

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 () {
    …
  }
}
```



Waiting to enter

Waiting on condition variables

Executing inside the monitor

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

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) {
    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 Readers and Writers

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

$$(nr \geq 0) \land (0 \leq nw \leq 1) \land ((nr > 0) \Rightarrow (nw = 0))$$

- Two conditions:
  - canRead: $nw = 0$
  - canWrite: $(nr = 0) \land (nw = 0)$

Monitor Readers and Writers

- Write with just wait()
  - Will be safe, maybe not live – why?

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

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

  void EndRead () {
    nr--;
  }
```

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

  void EndWrite () {
    nw--;
  }
} // end monitor
```

Monitor Readers and Writers

- add signal() and broadcast()

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

  void StartRead () {
    while (nw != 0) do wait(canRead);
    nr++;
  }          can we put a signal here?

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

```
  void StartWrite () {
    while (nr != 0 || nw != 0) do wait(canWrite);
    nw++;
  }          can we put a signal here?

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

Monitor Readers and Writers

- Is there any priority between readers and writers?
- What if you wanted to ensure that a waiting writer would have priority over new readers?

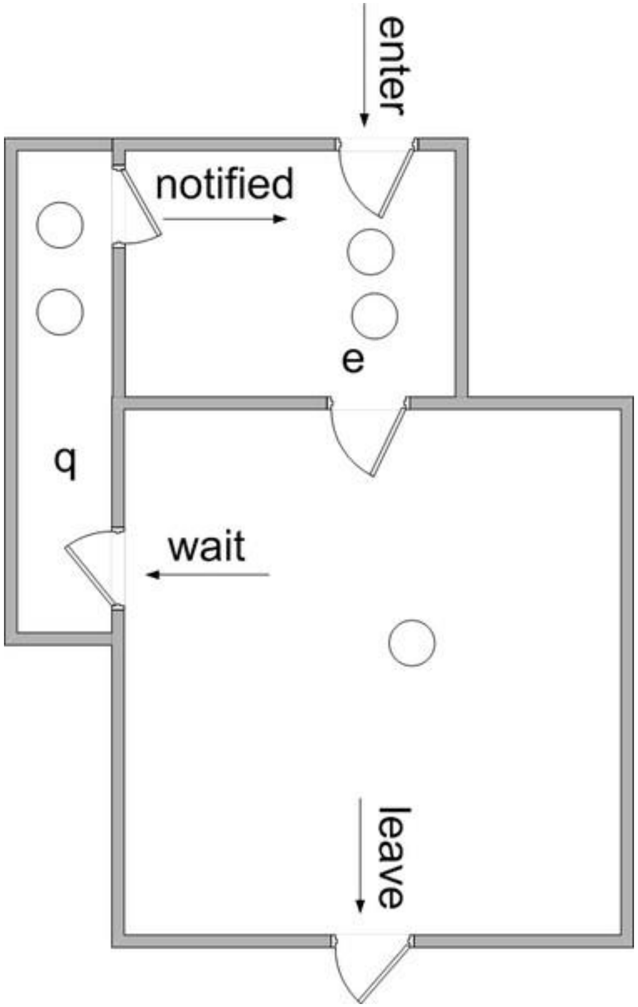
Monitors and Java

- 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
  - The compiler generates code to acquire the object's lock at the start of the method and release it just before returning
    - » The lock itself is implicit, programmers do not worry about it

Monitors and Java

- Every object can be treated as a condition variable
  - Half of Object's methods are for synchronization!
- Take a look at the Java Object class:
  - Object::wait(*) is Condition::wait()
  - Object::notify() is Condition::signal()
  - Object::notifyAll() is Condition::broadcast()

Monitors and Java



- https://commons.wikimedia.org/wiki/File:Monitor_(synchronization)-Java.png

Condition Vars & Locks

- Condition variables are also used without monitors in conjunction with blocking locks
  - This is what you are implementing in Project 1
- 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

Condition Variables

- 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 more clear

Condition Vars != Semaphores

- Condition variables != semaphores
  - Although their operations have the same names, they have entirely different semantics (such is life, worse yet to come)
  - 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

Using Cond Vars & 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

CV Implementation – Data Struct.

```
struct condition {
    proc next; /* doubly linked list implementation of */
    proc prev; /* queue for blocked threads */
    mutex listLock; /*protects queue */
};
```

CV – Wait Implementation

```
void wait (condition *cv, mutex *mx)
{
    mutex_acquire(&cv->listLock); /* protect the queue */
    enqueue(&cv->next, &cv->prev, thr_self()); /* enqueue */
    mutex_release (&cv->listLock); /* we're done with the list */
    /* The suspend and mutex_release operation must be atomic */
   mutex_release(mx);
   thr_suspend (self); /* Sleep 'til someone wakes us */
   mutex_acquire(mx); /* Woke up – our turn, get resource lock */
   return;
}
```

CV – Signal Implementation

```
void signal (condition *cv)
{
    thread_id tid;
    mutex_acquire(cv->listlock); /* protect the queue */
    tid = dequeue(&cv->next, &c->prev);
    mutex_release(listLock);
    if (tid>0)
        thr_continue (tid);
    return;
}
/* Note: This did not release mx */
```

CV Implementation - Broadcast

```
void broadcast (condition *cv)
{
    thread_id tid;
    mutex_acquire(c->listLock); /* protect the queue */

    while (&cv->next) /* queue is not empty */
    {
        tid = dequeue(&c->next, &c->prev); /* wake one */
        thr_continue (tid); /* Make it runnable */
     }
        mutex_release (c->listLock); /* done with the queue */
}
/* Note: This did not release mx */
```

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