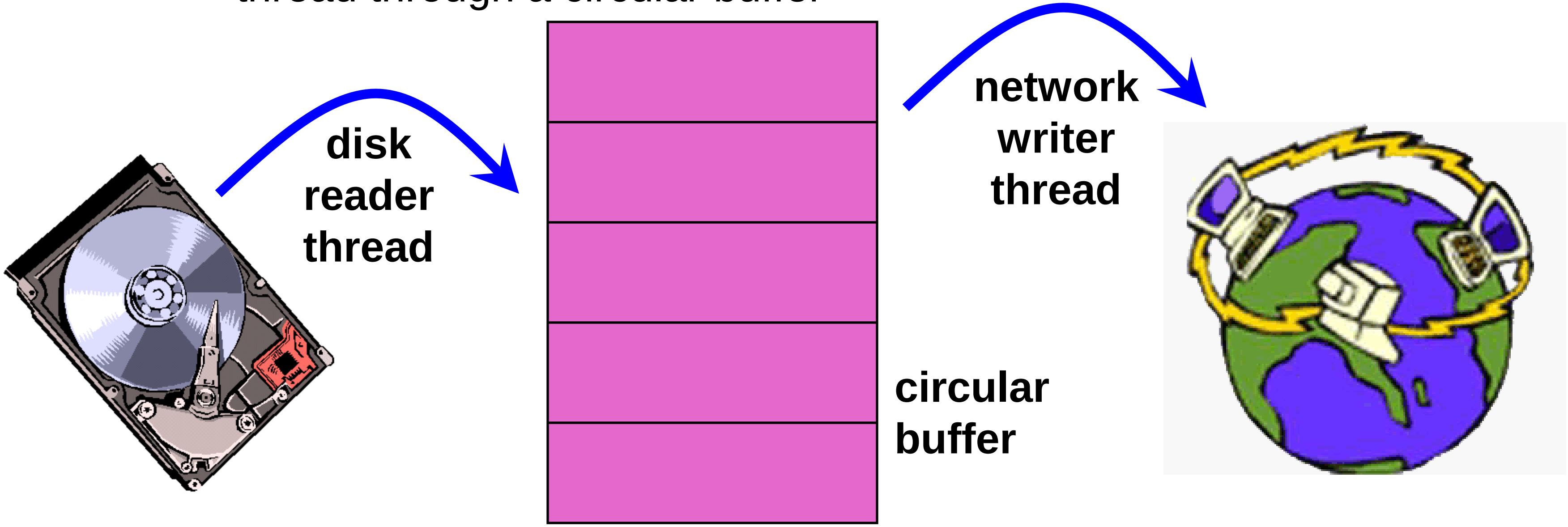
CSE 451: Operating Systems
Winter 2013

Synchronization

Gary Kimura

Synchronization

- Threads cooperate in multithreaded programs
  - to share resources, access shared data structures
    - e.g., threads accessing a memory cache in a web server
  - also, to coordinate their execution
    - e.g., a disk reader thread hands off blocks to a network writer thread through a circular buffer



disk reader thread

network writer thread

circular buffer

Synchronization

- For correctness, we have to control this cooperation
  - must assume threads interleave executions arbitrarily and at different rates
    - Modern OS's are preemptive
    - Most new machines are multicore
    - scheduling is not under application writers' control (except for real-time, but that's not of interest here).
- We control cooperation using synchronization
  - enables us to restrict the interleaving of executions
- Note: this also applies to processes, not just threads
  - (I'll almost never say "process" again!)
- It also applies across machines in a distributed system (Big Research Topic)

Shared resources

- We’ll focus on coordinating access to shared resources
  - basic problem:
    - two concurrent threads are accessing a shared variable
    - if the variable is read/modified/written by both threads, then access to the variable must be controlled
    - otherwise, unexpected results may occur
- Over the next several lectures, we’ll look at:
  - mechanisms to control access to shared resources
    - low level mechanisms like locks
    - higher level mechanisms like mutexes, semaphores, monitors, and condition variables
  - patterns for coordinating access to shared resources
    - bounded buffer, producer-consumer, …

The classic example

- Suppose we have to implement a function to withdraw money from a bank account:

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

- Now suppose that you and your S.O. share a bank account with a balance of $100.00
  - what happens if you both go to separate ATM machines, and simultaneously withdraw $10.00 from the account?

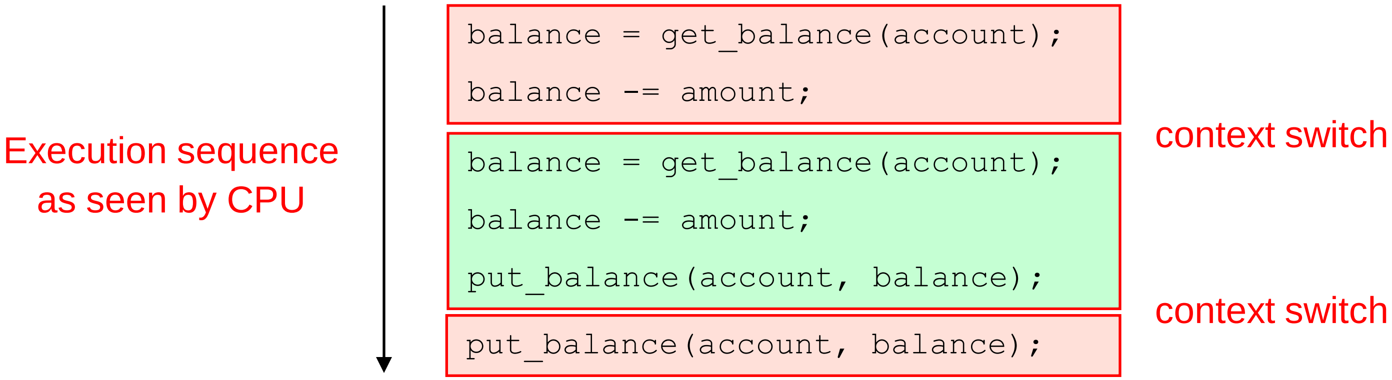
- Represent the situation by creating a separate thread for each person to do the withdrawals
  - have both threads run on the same bank mainframe:

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

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

Interleaved schedules

- The problem is that the execution of the two threads can be interleaved, assuming preemptive scheduling:



Execution sequence as seen by CPU

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

context switch

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

context switch

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

- What’s the account balance after this sequence?
  - who’s happy, the bank or you?
- How often is this unfortunate sequence likely to occur?

Other Execution Orders

- Which interleavings are ok? Which are not?

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

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

How About Now?

```
int xfer(from, to, amt) {
  int bal = withdraw(from, amt);
  deposit( to, amt );
  return bal;
}
```

```
int xfer(from, to, amt) {
  int bal = withdraw(from, amt);
  deposit( to, amt );
  return bal;
}
```

And This?

```
i++;
```

```
i++;
```

The crux of the matter

- The problem is that two concurrent threads (or processes) access a shared resource (account) without any synchronization
  - creates a **race condition**
    - output is non-deterministic, depends on timing
- We need mechanisms for controlling access to shared resources in the face of concurrency
  - so we can reason about the operation of programs
    - essentially, re-introducing determinism
- Synchronization is necessary for any shared data structure
  - buffers, queues, lists, hash tables, scalars, …

What resources are shared?

- Local variables are *not* shared
  - 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 are shared
  - stored in the static data segment, accessible by any thread
- Dynamic objects are shared
  - stored in the heap, shared if you can name it
    - in C, can conjure up the pointer
      - e.g., `void *x = (void *) 0xDEADBEEF`
    - in Java/C#, strong typing prevents this
      - must pass references explicitly

Mutual exclusion

- We want to use mutual exclusion to synchronize access to shared resources
- Mutual exclusion makes reasoning about program behavior easier
  - making reasoning easier leads to fewer bugs
- 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 requirements

- Critical sections have the following requirements
  - mutual exclusion
    - at most one thread is in the critical section
  - progress
    - if thread T is outside the critical section, then T cannot prevent thread S from entering the critical section
  - bounded waiting (no starvation)
    - if thread T is waiting on the critical section, then T will eventually enter the critical section
      - assumes threads eventually leave critical sections
    - vs. fairness?
  - performance
    - the overhead of entering and exiting the critical section is small with respect to the work being done within it

Mechanisms for building critical sections

- Locks
  - very primitive, minimal semantics; used to build others
- Semaphores
  - basic, easy to get the hang of, hard to program with
- Monitors
  - high level, requires language support, implicit operations
  - easy to program with; Java "`synchronized()`" as an example
- Messages
  - simple model of communication and synchronization based on (atomic) transfer of data across a channel
  - direct application to distributed systems (SOAP, RPC)

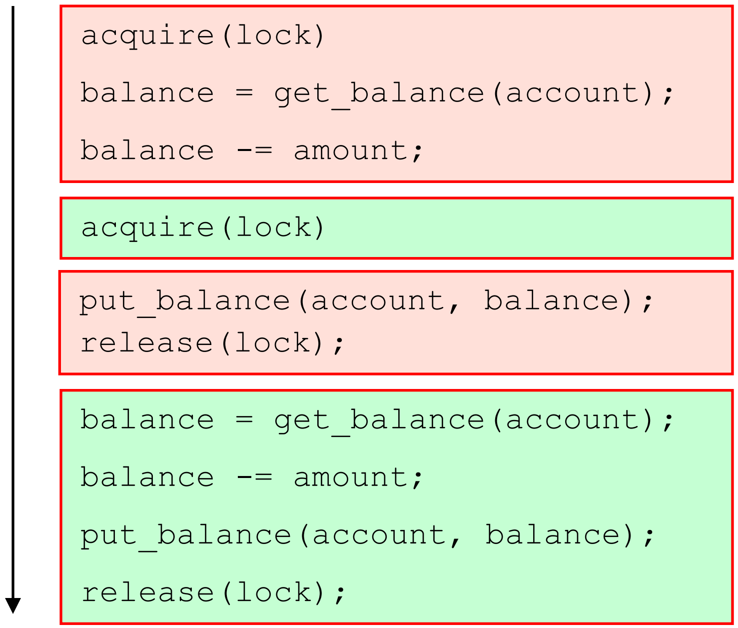
Locks

- A lock is a object (in memory) that provides the following two operations:
  - `acquire()`: a thread calls this before entering a critical section
  - `release()`: a thread calls this after leaving a critical section
- Threads pair up calls to `acquire()` and `release()`
  - between `acquire()` and `release()`, the thread holds the lock
  - `acquire()` does not return until the caller holds the lock
    - at most one thread can hold a lock at a time (usually)
  - so: what can happen if the calls aren't paired?
- Two basic flavors of locks
  - spinlock
  - blocking (a.k.a. "mutex")

Using locks

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

critical section



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

```
acquire(lock)
```

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

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

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

Spinlocks

- How do we implement locks? Here’s one attempt:

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

the caller “busy-waits”,
or spins, for lock to be
released ⇒ hence spinlock

- Why doesn’t this work?
  - where is the race condition?

Implementing locks (cont.)

- Problem is that implementation of locks has critical sections, too!
  - the acquire/release must be **atomic**
    - atomic == executes as though it could not be interrupted
    - code that executes “all or nothing”
- Need help from the hardware
  - disable/enable interrupts
    - to prevent context switches
  - atomic instructions
    - test-and-set, compare-and-swap, …
  - multiple processors?

Spinlocks redux: Test-and-Set

- CPU provides the following as one atomic instruction:

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

- Remember, this is a single instruction…

Spinlocks redux: Test-and-Set

- So, to fix our broken spinlocks, do:

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

  - mutual exclusion?
  - progress?
  - bounded waiting?
  - performance?

Real World Example

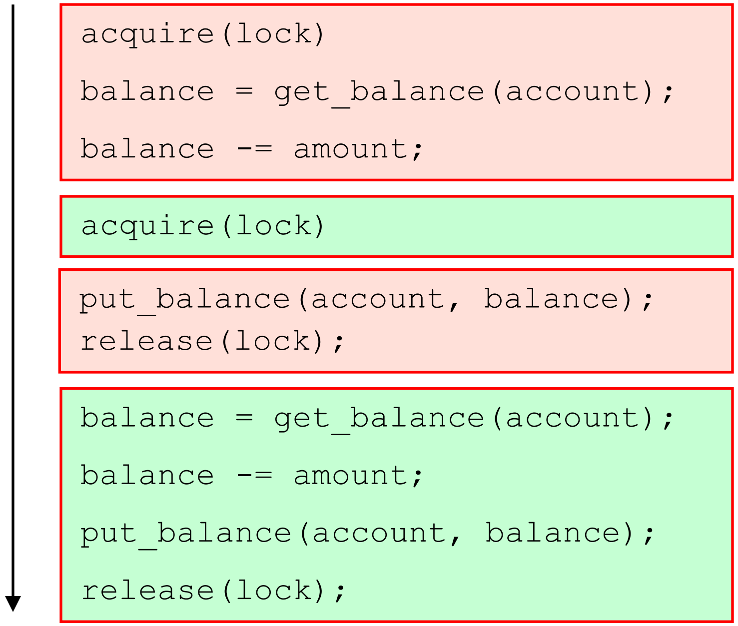
- Windows XP AcquireSpinlock

```
AcquireSpinlock:
;
; Attempt to assert the lock
;
        lock bts dword ptr [LockAddress], 0
        jc      SpinLabel  ; spinlock owned
        ret
SpinLabel:
;
; Was spinlock cleared?
;
        test    dword ptr [LockAddress], 1
        jz      AcquireSpinlock
        YIELD
        jmp     Spinlabel
; ...
```

Reminder of use …

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

critical section



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

```
acquire(lock)
```

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

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

- How does a thread blocked on an "acquire" (that is, stuck in a test-and-set loop) yield the CPU?
  - calls yield( ) *(spin-then-block)*
  - there's an involuntary context switch

Problems with spinlocks

- Spinlocks work, but are horribly wasteful!
  - if a thread is spinning on a lock, the thread holding the lock cannot make progress
  - And neither can anyone else! Why?
- Only want spinlocks as primitives to build higher-level synchronization constructs
  - Why is this okay?
- *When might the above points be misleading?*

Another approach: Disabling interrupts

```
struct lock {
}
void acquire(lock) {
   cli();   // disable interrupts
}
void release(lock) {
  sti();    // reenable interrupts
}
```

Problems with disabling interrupts

- Only available to the kernel
  - Can’t allow user-level to disable interrupts!
- Insufficient on a multiprocessor
  - Each processor has its own interrupt mechanism
- “Long” periods with interrupts disabled can wreak havoc with devices

- Just as with spinlocks, you only want to use disabling of interrupts to build higher-level synchronization constructs

Simple Locks

- Locks are the lowest-level mechanism
  - very primitive in terms of semantics – error-prone
  - implemented by spin-waiting (crude) or by disabling interrupts (also crude, and can only be done in the kernel)
- What else is there
  - semaphores are a slightly higher level abstraction
    - less crude implementation too
  - monitors are significantly higher level
    - utilize programming language support to reduce errors

Semaphores

- Semaphore = a synchronization primitive
  - higher level of abstraction than locks
  - invented by Dijkstra in 1968, as part of the THE operating system
- A semaphore is:
  - a variable that is manipulated through two operations, P and V (Dutch for “test” and “increment”)
    - **P(sem)** (wait/down)
      - block until sem > 0, then subtract 1 from sem and proceed
    - **V(sem)** (signal/up)
      - add 1 to sem
- Do these operations *atomically*

Blocking in semaphores

- Each semaphore has an associated queue of threads
  - when P(sem) is called by a thread,
    - if sem was “available” (>0), decrement sem and let thread continue
    - if sem was “unavailable” (<=0), place thread on associated queue; dispatch some other runnable thread
  - when V(sem) is called by a thread
    - if thread(s) are waiting on the associated queue, unblock one
      - place it on the ready queue
      - might as well let the “V-ing” thread continue execution
      - or not, depending on priority
    - otherwise (when no threads are waiting on the sem), increment sem
      - the signal is “remembered” for next time P(sem) is called
- Semaphores thus have history

Abstract implementation

- P/wait/down(sem)
  - acquire “real” mutual exclusion
    - if sem is “available” (>0), decrement sem; release “real” mutual exclusion; let thread continue
    - otherwise, place thread on associated queue; release “real” mutual exclusion; run some other thread
- V/signal/up(sem)
  - acquire “real” mutual exclusion
    - if thread(s) are waiting on the associated queue, unblock one (place it on the ready queue)
    - if no threads are on the queue, sem is incremented
      - » the signal is “remembered” for next time P(sem) is called
  - release “real” mutual exclusion
  - [the “V-ing” thread continues execution or is preempted]

Two types of semaphores

- Binary semaphore (aka mutex semaphore)
  - sem is initialized to 1
  - guarantees mutually exclusive access to resource (e.g., a critical section of code)
  - only one thread/process allowed entry at a time

- Counting semaphore
  - sem is initialized to N
    - N = number of units available
  - represents resources with many (identical) units available
  - allows threads to enter as long as more units are available

Usage

- From the programmer’s perspective, P and V on a binary semaphore are just like Acquire and Release on a lock

```
P(sem)
    .
    .
    .
    do whatever stuff requires mutual exclusion; could conceivably
    be a lot of code
    .
    .
    .
V(sem)
```

  - same lack of programming language support for correct usage

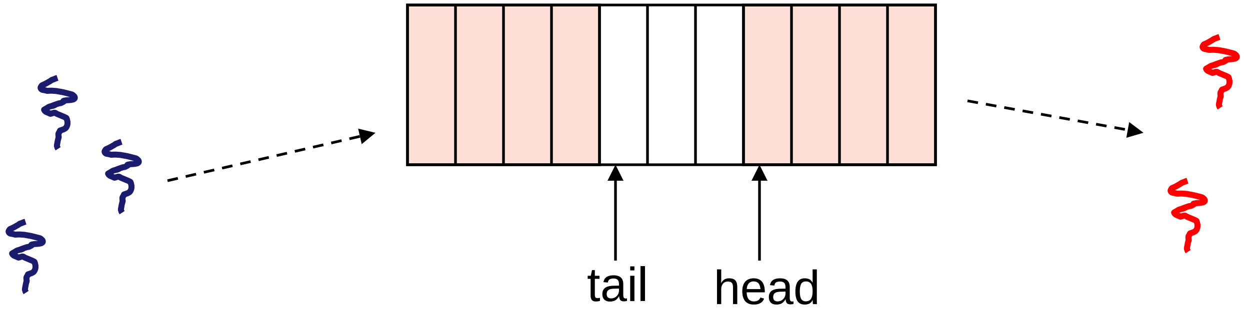
- Important differences in the underlying implementation, however

Pressing questions

- How do you acquire “real” mutual exclusion?
- Why is this any better than using a spinlock (test-and-set) or disabling interrupts (assuming you’re in the kernel) in lieu of a semaphore?
- What if some bozo issues an extra V?
- What if some bozo forgets to P?

Example: Bounded buffer problem

- *AKA* “producer/consumer” problem
  - there is a buffer in memory with N entries
  - producer threads insert entries into it (one at a time)
  - consumer threads remove entries from it (one at a time)
- Threads are concurrent
  - so, we must use synchronization constructs to control access to shared variables describing buffer state



Bounded buffer using semaphores (both binary and counting)

```
var mutex: semaphore = 1     ;mutual exclusion to shared data
    empty: semaphore = n     ;count of empty buffers (all empty to start)
    full: semaphore = 0      ;count of full buffers (none full to start)
```

```
producer:
    P(empty)                 ; one fewer buffer, block if none available
    P(mutex)                 ; get access to pointers
      <add item to buffer>
    V(mutex)                 ; done with pointers
    V(full)                  ; note one more full buffer
```

```
consumer:
    P(full)                  ;wait until there's a full buffer
    P(mutex)                  ;get access to pointers
      <remove item from buffer>
    V(mutex)                 ; done with pointers
    V(empty)                 ; note there's an empty buffer
      <use the item>
```

Note 1:

I have elided all the code concerning which is the first full buffer, which is the last full buffer, etc.

Note 2:

Try to figure out how to do this without using counting semaphores!

Example: Readers/Writers

- Description:
  - A single object is shared among several threads/processes
  - Sometimes a thread just reads the object
  - Sometimes a thread updates (writes) the object

  - **We can allow multiple readers at a time**
    - why?

  - **We can only allow one writer at a time**
    - why?

Readers/Writers using semaphores

```
var mutex: semaphore = 1         ; controls access to readcount
    wrt: semaphore = 1           ; control entry for a writer or first reader
    readcount: integer = 0       ; number of active readers
```

```
writer:
        P(wrt)                   ; any writers or readers?
                <perform write operation>
        V(wrt)                   ; allow others
```

```
reader:
        P(mutex)                                  ; ensure exclusion
          readcount++                             ; one more reader
          if readcount == 1 then P(wrt)           ; if we’re the first, synch with writers
        V(mutex)
                <perform read operation>
        P(mutex)                                  ; ensure exclusion
          readcount--                             ; one fewer reader
          if readcount == 0 then V(wrt)           ; no more readers, allow a writer
        V(mutex)
```

Readers/Writers notes

- Notes:
  - the first reader blocks on P(wrt) if there is a writer
    - any other readers will then block on P(mutex)
  - if a waiting writer exists, the last reader to exit signals the waiting writer
    - can new readers get in while a writer is waiting?
  - when writer exits, if there is both a reader and writer waiting, which one goes next?

Semaphores vs. Locks

- Threads that are blocked at the level of program logic are placed on queues, rather than busy-waiting

- Busy-waiting may be used for the “real” mutual exclusion required to implement P and V
    - but these are very short critical sections – totally independent of program logic

- In the not-very-interesting case of a thread package implemented in an address space “powered by” only a single kernel thread, it’s even easier that this

Problems with semaphores (and locks)

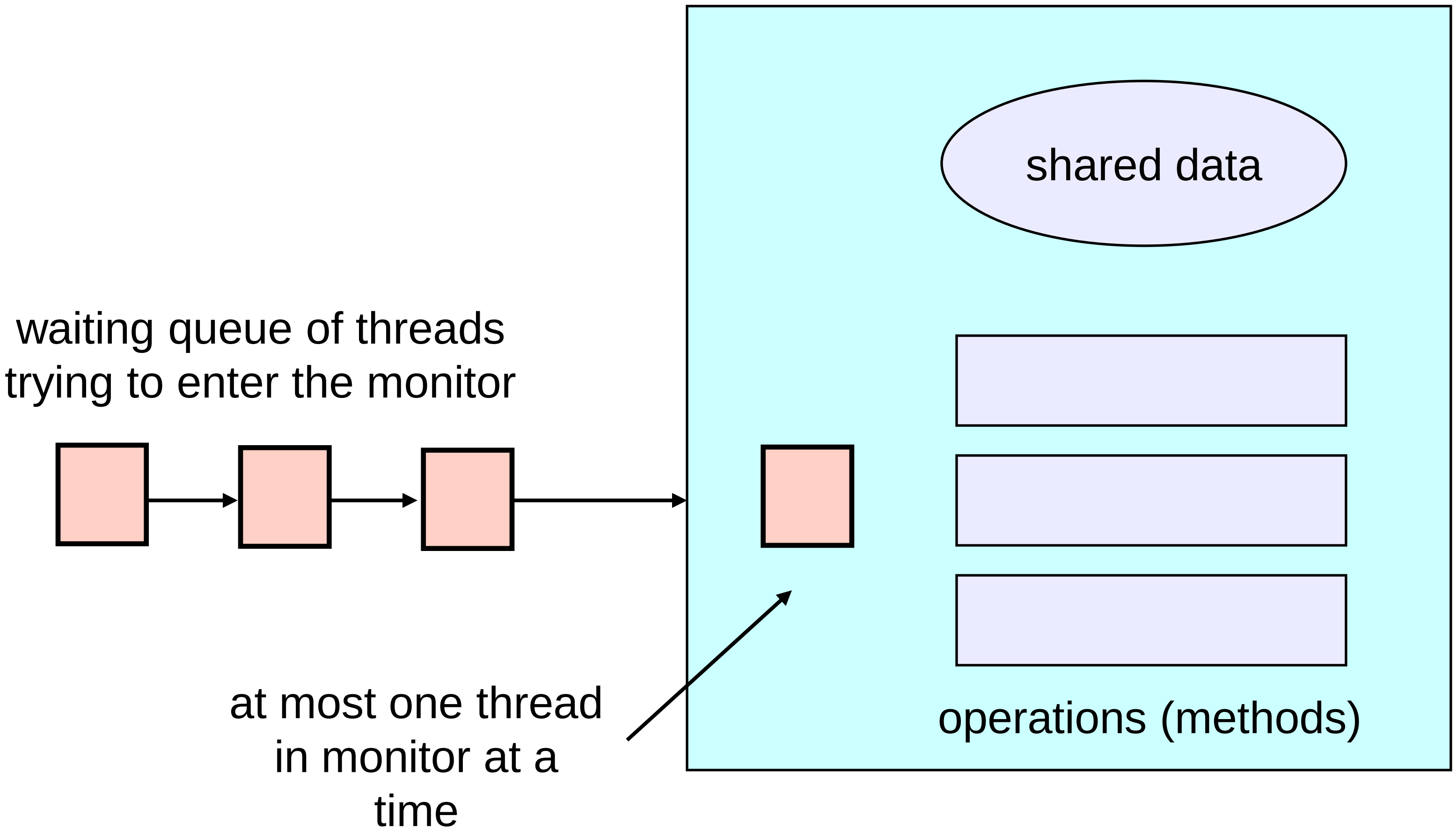
- They can be used to solve any of the traditional synchronization problems, but:
  - semaphores are essentially shared global variables
    - can be accessed from anywhere (bad software engineering)
  - there is no connection between the semaphore and the data being controlled by it
  - used for both critical sections (mutual exclusion) and for coordination (scheduling)
  - no control over their use, no guarantee of proper usage

- Thus, they are prone to bugs
  - another (better?) approach: use programming language support

One More Approach: Monitors

- A *monitor* is a programming language construct that supports controlled access to shared data
  - synchronization code is added by the compiler
    - why does this help?
- A monitor encapsulates:
  - shared data structures
  - procedures that operate on the shared data
  - synchronization between concurrent threads that invoke those procedures
- Data can only be accessed from within the monitor, using the provided procedures
  - protects the data from unstructured access
- Addresses the key usability issues that arise with semaphores

A monitor

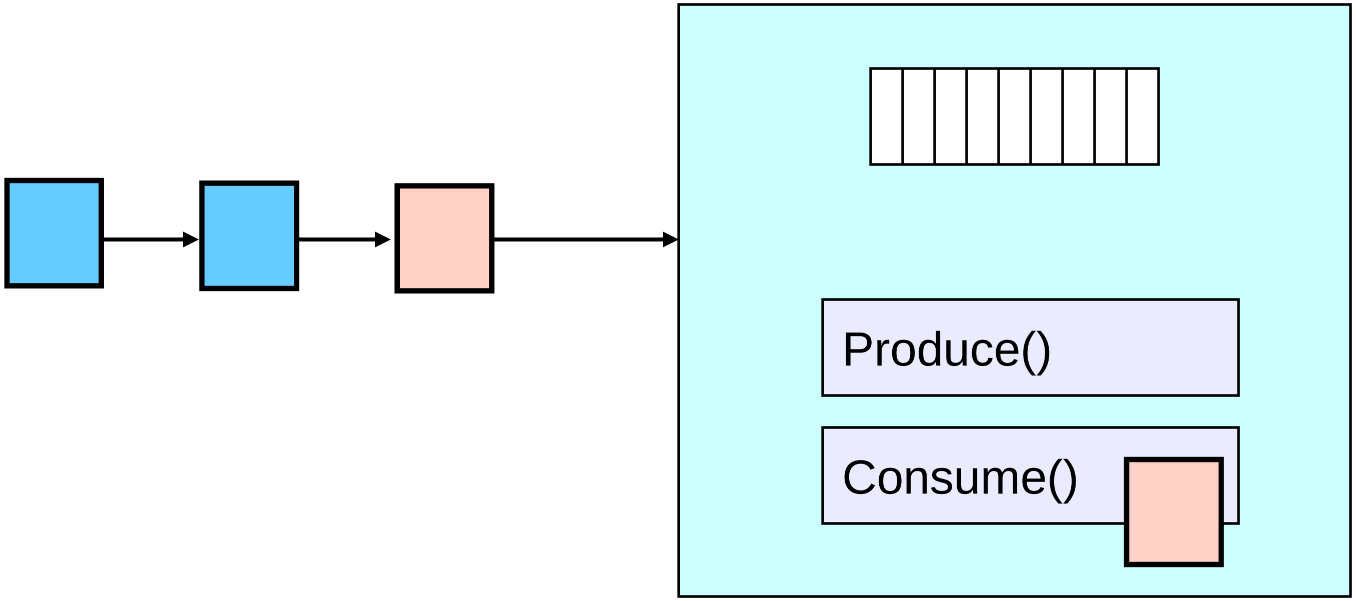


shared data
waiting queue of threads
trying to enter the monitor
at most one thread
in monitor at a
time
operations (methods)

Monitor facilities

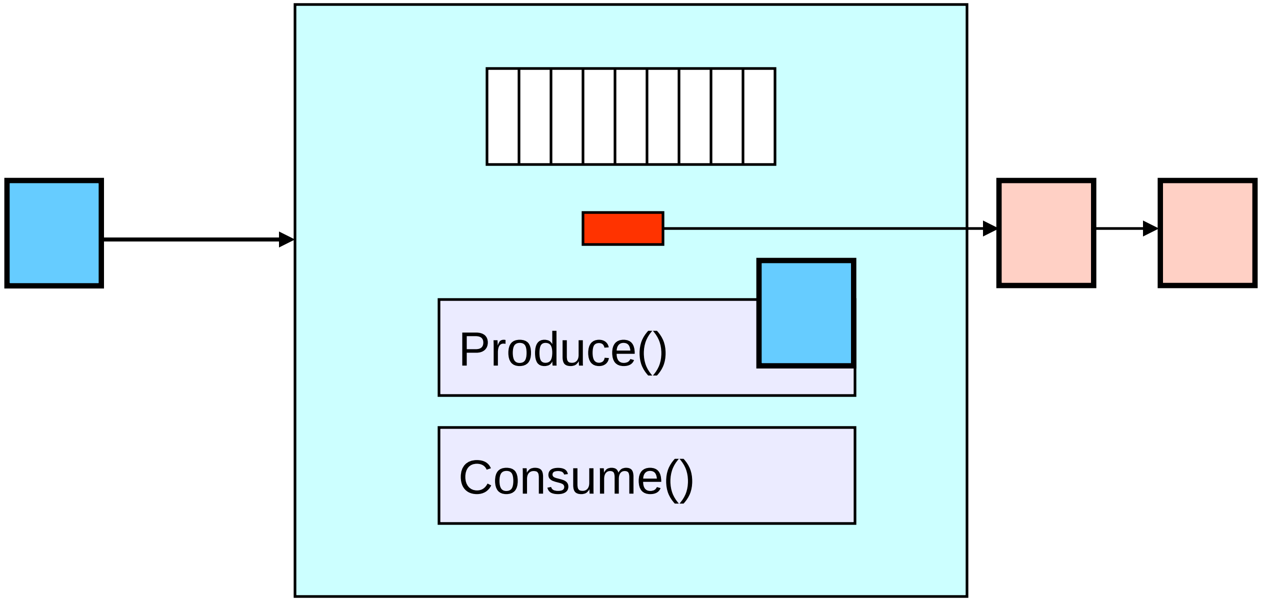
- “Automatic” mutual exclusion
  - only one thread can be executing inside at any time
    - thus, synchronization is implicitly associated with the monitor – it “comes for free”
  - if a second thread tries to execute a monitor procedure, it blocks until the first has left the monitor
    - more restrictive than semaphores
    - but easier to use (most of the time)
- But, there’s a problem…

Example: Bounded Buffer Scenario



- Buffer is empty
- Now what?

Example: Bounded Buffer Scenario



- Buffer is empty
- Now what?

Condition variables

- A place to wait; sometimes called a rendezvous point
- “Required” for monitors
  - So useful they’re often provided even when monitors aren’t available
- Three operations on condition variables
  - wait(c)
    - release monitor lock, so somebody else can get in
    - wait for somebody else to signal condition
    - thus, condition variables have associated wait queues
  - signal(c)
    - wake up at most one waiting thread
    - if no waiting threads, signal is lost
      - this is different than semaphores: no history!
  - broadcast(c)
    - wake up all waiting threads

Bounded buffer using (Hoare) monitors

```
Monitor bounded_buffer {
  buffer resources[N];
  condition not_full, not_empty;

produce(resource x) {
    if (array "resources" is full, determined maybe by a count)
          wait(not_full);
    insert "x" in array "resources"
    signal(not_empty);
  }

 consume(resource *x) {
    if (array "resources" is empty, determined maybe by a count)
          wait(not_empty);
    *x = get resource from array "resources"
    signal(not_full);
  }
```

Runtime system calls for (Hoare) monitors

- EnterMonitor(m) {guarantee mutual exclusion}
- ExitMonitor(m) {hit the road, letting someone else run}
- Wait(c) {step out until condition satisfied}
- Signal(c) {if someone’s waiting, step out and let him run}

Bounded buffer using (Hoare) monitors

```
Monitor bounded_buffer {
  buffer resources[N];
  condition not_full, not_empty;

  procedure add_entry(resource x) {                    EnterMonitor
    if (array "resources" is full, determined maybe by a count)
      wait(not_full);
    insert "x" in array "resources"
    signal(not_empty);                                 ExitMonitor
  }
  procedure get_entry(resource *x) {                   EnterMonitor
    if (array "resources" is empty, determined maybe by a count)
      wait(not_empty);
    *x = get resource from array "resources"
    signal(not_full);                                  ExitMonitor
  }
```

There is a subtle issue with that code…

- Who runs when the signal() is done and there is a thread waiting on the condition variable?

- **Hoare monitors**: signal(c) means
  - run waiter immediately
  - signaller blocks immediately
    - condition guaranteed to hold when waiter runs
    - but, signaller must restore monitor invariants before signalling!
      - cannot leave a mess for the waiter, who will run immediately!

- **Mesa monitors**: signal(c) means
  - waiter is made ready, but the signaller continues
    - waiter runs when signaller leaves monitor (or waits)
  - signaller need not restore invariant until it leaves the monitor
  - being woken up is only a hint that something has changed
    - signalled condition may no longer hold
    - must recheck conditional case

Hoare vs. Mesa Monitors

- Hoare monitors: `if (notReady) wait(c)`
- Mesa monitors: `while (notReady) wait(c)`
- Mesa monitors easier to use
  - more efficient: fewer context switches
  - directly supports broadcast
- Hoare monitors leave less to chance
  - when wake up, condition guaranteed to be what you expect

Runtime system calls for Hoare monitors

- EnterMonitor(m) {guarantee mutual exclusion}
  - if m occupied, insert caller into queue m
  - else mark as occupied, insert caller into ready queue
  - choose somebody to run
- ExitMonitor(m) {hit the road, letting someone else run}
  - if queue m is empty, then mark m as unoccupied
  - else move a thread from queue m to the ready queue
  - insert caller in ready queue
  - choose someone to run

Runtime system calls for Hoare monitors (cont’d)

- Wait(c) {step out until condition satisfied}
  - if queue m is empty, then mark m as unoccupied
  - else move a thread from queue m to the ready queue
  - put the caller on queue c
  - choose someone to run
- Signal(c) {if someone’s waiting, step out and let him run}
  - if queue c is empty then put the caller on the ready queue
  - else move a thread from queue c to the ready queue, and put the caller into queue m
  - choose someone to run

Runtime system calls for Mesa monitors

- EnterMonitor(m) {guarantee mutual exclusion}
  - …
- ExitMonitor(m) {hit the road, letting someone else run}
  - …
- Wait(c) {step out until condition satisfied}
  - …
- Signal(c) {if someone’s waiting, give him a shot after I’m done}
  - if queue c is occupied, move one thread from queue c to queue m
  - return to caller

- Broadcast(c) {food fight!}
  - move all threads on queue c onto queue m
  - return to caller

Monitor Summary

- Language supports monitors
- Compiler understands them
  - compiler inserts calls to runtime routines for
    - monitor entry
    - monitor exit
    - signal
    - Wait
  - Language/object encapsulation ensures correctness
    - Sometimes! With conditions you STILL need to think about synchronization
- Runtime system implements these routines
  - moves threads on and off queues
  - *ensures mutual exclusion!*