CSE 120 Principles of Operating Systems

Fall 2026

Lecture 6: Semaphores and Monitors

Geoffrey M. Voelker

Administrivia

- Project 1 ongoing
- Homework 2 ongoing
- Lab hour calendar in autograder

Higher-Level Synchronization

- We looked at using locks to provide mutual exclusion
- Locks are necessary, they have limited semantics
  - Just provide mutual exclusion
- Also want synchronization mechanisms that
  - Block waiters, leave interrupts enabled in critical sections
  - Provide semantics beyond mutual exclusion
- Look at two common high-level mechanisms
  - Semaphores: binary and counting
  - Monitors: locks and condition variables
- Use them to solve common synchronization problems

Semaphores

- Semaphores are an abstract data type that provide mutual exclusion to critical sections
  - Described by Dijkstra in the "THE" system in 1968
- Semaphores are also used as atomic counters
  - For coordinating the execution of processes/threads
- Semaphores are "integers" that support two operations:
  - Semaphore.P(): decrement, block until semaphore is open
    - » P() after the Dutch word for "try to reduce" (also test, down)
  - Semaphore.V(): increment, allow another thread to enter
    - » V() after the Dutch word for increment, up
  - That's it! No other operations – not even just reading its value

sem_overview(7) — Linux manual page

NAME top

sem_overview - overview of POSIX semaphores

DESCRIPTION top

POSIX semaphores allow processes and threads to synchronize their actions.

A semaphore is an integer whose value is never allowed to fall below zero. Two operations can be performed on semaphores: increment the semaphore value by one (sem_post(3)); and decrement the semaphore value by one (sem_wait(3)). If the value of a semaphore is currently zero, then a sem_wait(3) operation will block until the value becomes greater than zero.

POSIX semaphores come in two forms: named semaphores and unnamed semaphores.

Named semaphores

A named semaphore is identified by a name of the form */somename*; that is, a null-terminated string of up to **NAME_MAX**-*4* (i.e., 251) characters consisting of an initial slash, followed by one or more characters, none of which are slashes. Two processes can operate on the same named semaphore by passing the same name to sem_open(3).

The sem_open(3) function creates a new named semaphore or

Filter by title

CreateSemaphoreA function
CreateSemaphoreExA function
CreateTapePartition function
CreateUmsCompletionList function
CreateUmsThreadContext function
DeleteFiber function
DeleteUmsCompletionList function
DeleteUmsThreadContext function
DequeueUmsCompletionListItems function
DestroyThreadpoolEnvironment function
EnterUmsSchedulingMode function
EraseTape function
ExecuteUmsThread function
GetActiveProcessorCount function
GetActiveProcessorGroupCount function
GetCurrentUmsThread function
GetEnvironmentVariable function
GetMaximumProcessorCount function
GetMaximumProcessorGroupCount function
GetNextUmsListItem function
GetNumaAvailableMemoryNode function
GetNumaAvailableMemoryNodeEx function
GetNumaNodeNumberFromHandle function
GetNumaNodeProcessorMask function
GetNumaProcessorNode function
GetNumaProcessorNodeEx function
GetNumaProximityNode function
GetProcessAffinityMask function

CreateSemaphoreA function (winbase.h)

07/26/2022

Creates or opens a named or unnamed semaphore object.

To specify an access mask for the object, use the CreateSemaphoreEx function.

Syntax

C++ Copy

```
HANDLE CreateSemaphoreA(
  [in, optional] LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
  [in]           LONG                  lInitialCount,
  [in]           LONG                  lMaximumCount,
  [in, optional] LPCSTR                lpName
);
```

Parameters

`[in, optional] lpSemaphoreAttributes`

A pointer to a SECURITY_ATTRIBUTES structure. If this parameter is **NULL**, the handle cannot be inherited by child processes.

The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new semaphore. If this parameter is **NULL**, the semaphore gets a default security descriptor. The ACLs in the default security descriptor for a semaphore come from the primary or impersonation token of the creator.

`[in] lInitialCount`

The initial count for the semaphore object. This value must be greater than or equal to zero and less than or equal to

std::counting_semaphore, std::binary_semaphore

Defined in header <semaphore>

```
template< std::ptrdiff_t LeastMaxValue = /* implementation-defined */ >
class counting_semaphore;
```
(1) (since C++20)

```
using binary_semaphore = std::counting_semaphore<1>;
```
(2) (since C++20)

1) A `counting_semaphore` is a lightweight synchronization primitive that can control access to a shared resource. Unlike a `std::mutex`, a `counting_semaphore` allows more than one concurrent access to the same resource, for at least `LeastMaxValue` concurrent accessors. The program is ill-formed if `LeastMaxValue` is negative.

2) `binary_semaphore` is an alias for specialization of `std::counting_semaphore` with `LeastMaxValue` being `1`. Implementations may implement `binary_semaphore` more efficiently than the default implementation of `std::counting_semaphore`.

A `counting_semaphore` contains an internal counter initialized by the constructor. This counter is decremented by calls to `acquire()` and related methods, and is incremented by calls to `release()`. When the counter is zero, `acquire()` blocks until the counter is incremented, but `try_acquire()` does not block; `try_acquire_for()` and `try_acquire_until()` block until the counter is incremented or a timeout is reached.

Similar to `std::condition_variable::wait()`, `counting_semaphore`'s `try_acquire()` can spuriously fail.

Specializations of `std::counting_semaphore` are not *DefaultConstructible*, *CopyConstructible*, *MoveConstructible*, *CopyAssignable*, or *MoveAssignable*.

Data Members

Member name	Definition
counter (private)	The internal counter of type `std::ptrdiff_t`. (exposition-only member object*)

Member functions

(constructor)	constructs a `counting_semaphore` (public member function)
(destructor)	destructs the `counting_semaphore` (public member function)

Blocking in Semaphores

- When decrement() is called by a thread
  - If semaphore is open, thread continues
  - If semaphore is closed, thread blocks on a wait queue
- Then increment() opens the semaphore
  - If a thread is waiting on the queue, the thread is unblocked
  - If no threads are waiting on the queue, the increment is remembered for the next thread
    - » Increment() has “history” (c.f., condition vars later)
    - » This “history” is a counter

Semaphore Types

- Semaphores come in two types
- Binary semaphore (or mutex 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"
    - » binary/mutex has count = 1, counting has count = N

Using Semaphores

- Use is similar to locks when used for mutual exclusion

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

Critical section

Threads block

decrement(S);
balance = get_balance(account);
balance = balance – amount;

decrement(S);

decrement(S);

put_balance(account, balance);
increment(S);

…
increment(S);

…
increment(S);

It is undefined which thread runs after increment

Semaphores in Nachos

```
P () {  // decrement
  Disable interrupts;
  if (value == 0) {
    add currentThread to waitQueue;
    KThread.sleep(); // currentThread
  } else {
    value = value – 1;
  }
  Restore interrupts;
}
```

```
V () {  // increment
  Disable interrupts;
  thread = get next on waitQueue;
  if (thread) {
    thread.ready();
  } else {
    value = value + 1;
  }
  Restore interrupts;
}
```

- Like locks, the semaphore methods are critical sections
- KThread.sleep() assumes interrupts are disabled
  - Note that interrupts are disabled only to enter/leave critical section
  - How can it sleep with interrupts disabled?

Interrupts Disabled During Context Switch

```
KThread.yield () {
  Disable interrupts;
  currentThread.ready(); // add to Q
  runNextThread(); // context switch
  Restore interrupts;
}
```

```
Semaphore.P () {  // decrement
  Disable interrupts;
  if (value == 0) {
    add currentThread to waitQueue;
    KThread.sleep(); // currentThread
  } else {
    value = value – 1;
  }
  Restore interrupts;
}
```

[KThread.yield]
Disable interrupts;
currentThread.ready();
runNextThread();

[KThread.yield]
(Returns from runNextThread)
Restore interrupts;

[Semaphore.P] (decrement)
Disable interrupts;
if (value == 0) {
  add currentThread to waitQueue;
  Kthread.sleep();

[KThread.yield]
(Returns from runNextThread)
Restore interrupts;

Using Semaphores

- We looked at a simple example for synchronization
  - Mutual exclusion while accessing a bank account
- Now we're going to use semaphores to look at examples for coordinating execution
  - Readers/Writers
  - Bounded Buffers

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: $(\#r \geq 0) \wedge (0 \leq \#w \leq 1) \wedge ((\#r > 0) \Rightarrow (\#w = 0))$

      (0 or more readers) ∧ (0 or 1 writer) ∧ (no simul. readers & writers)
- How can we use semaphores to control access to the object to implement this protocol?
- 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;
// coordinate writer or readers
Semaphore w_or_r = 1;

writer {
    w_or_r.decr(); // lock out readers
    Write;
    w_or_r.incr();  // up for grabs
}
```

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

Readers/Writers Notes

- w_or_r coordinates between readers and writers
  - writer decr/incr, reader decr/incr when readcount goes from 0 to 1 or from 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?
- Why don't writers use mutex?
- What if “incr(mutex)” is before “if (readcount == 1)”?

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
  - No serialization of one behind the other
  - Tasks are independent, decoupled (easier to think about)
  - The buffer set allows each to run without explicit handoff
- Safety
  - Seq. of consumed values is prefix of seq. of produced values

Bounded Buffer (2)

- Invariants
  - $0 \leq \#p - \#c \leq N$ – Produced never more than N
  - $0 \leq (\#c - \#p) + N \leq N$ – Consumed never more than produced
- Use three semaphores:
  - empty – count of empty buffers
    - » Counting semaphore
    - » empty = $(\#c - \#p) + N$
  - full – count of full buffers
    - » Counting semaphore
    - » full = $\#p - \#c$
  - mutex – mutual exclusion to shared set of buffers
    - » Binary semaphore

Bounded Buffer (3)

```
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;
    empty.decr(); // wait for empty buffer
    mutex.decr(); // lock buffer list
    Add resource to an empty buffer;
    mutex.incr();  // unlock buffer list
    full.incr();       // note a full buffer
  }
}
```

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

Bounded Buffer (4)

- Why need the mutex at all?
- Where are the critical sections?
- The pattern of incr/decr on full/empty is a common construct often called an interlock
- What happens if operations on mutex and full/empty are switched around?

- Producer-Consumer and Bounded Buffer are classic examples of synchronization problems

Semaphore Questions

- Any problems that can be solved with counting semaphores that cannot be solved with mutexes?

- Can we predict which thread is unblocked by an increment operation?
  - Hint: consider the following three processes sharing a semaphore mutex that is initially 1:

```
while (1) {
    mutex.decr();
    // in critical section
    mutex.incr();
}
```

```
while (1) {
    mutex.decr();
    // in critical section
    mutex.incr();
}
```

```
while (1) {
    mutex.decr();
    // in critical section
    mutex.incr();
}
```

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 → prone to bugs
  - Another approach: Use programming language support

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

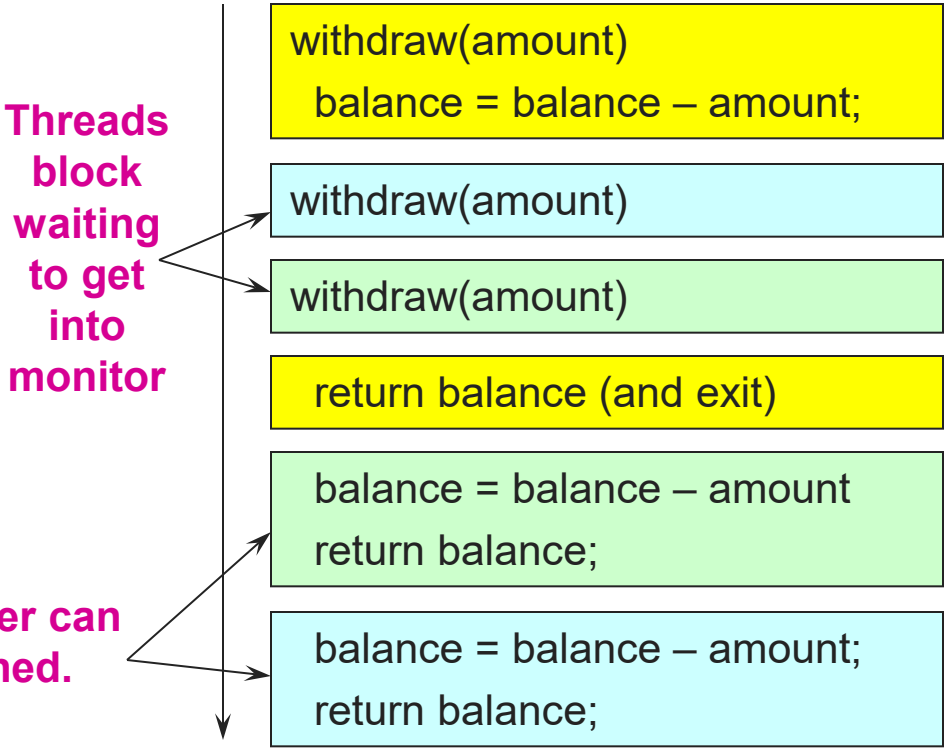
Monitor Semantics

- 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
    - » The monitor has a lock with a wait queue
  - If a thread within a monitor blocks, another one can enter
  - If you've used `synchronized` methods in Java, you've used monitors!
- What are the implications in terms of parallelism in a monitor? (How many threads can be executing monitor methods at a time?)

Account Example

```
Monitor Account {
  double balance;

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



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

Condition Variables

- Condition variables support three operations:
  - Wait – release monitor lock, sleep, reacquire lock when woken
    - » So condition variables have wait queues, too
    - » Also wait (Java, C++, Rust), sleep (Nachos, C#)
  - Signal – wakeup a waiting thread
    - » Also wake (Nachos, C#), notify (Java), notify_one (C++, Rust)
  - Broadcast – wakeup all waiting threads
    - » Also wakeAll (Nachos, C#), notifyAll (Java), notify_all (C++, Rust)
- 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 BoundedBuffer {
  Resource buffer[N];
  // Variables for indexing buffer
  // monitor invariant involves these vars
  Condition notFull;      // space in buffer
  Condition notEmpty;  // value in buffer

  void putResource (Resource R) {
    while (buffer array is full)
        notFull.wait();
    Add R to buffer array;
    notEmpty.signal();
  }
```

```
  Resource getResource() {
    while (buffer array is empty)
        notEmpty.wait();
    Get resource R from buffer array;
    notFull.signal();
    return R;
  }
} // end monitor
```

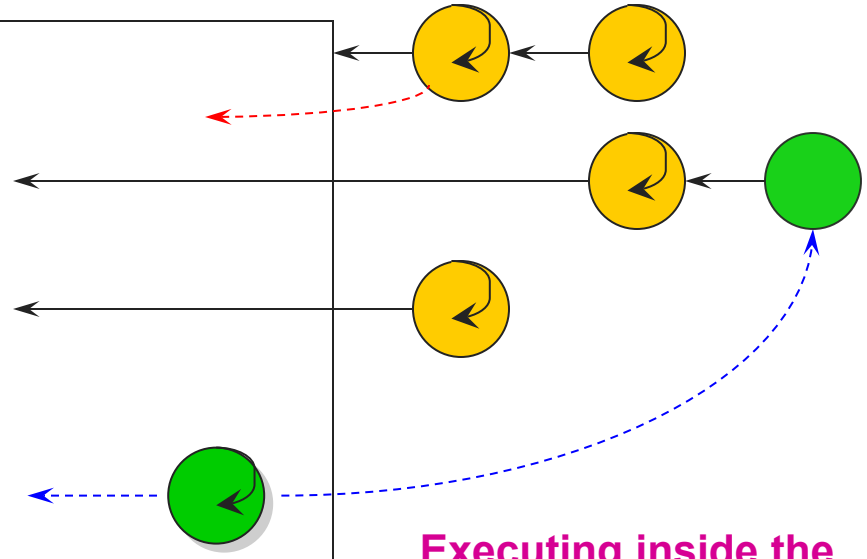
- What happens if no threads are waiting when signal is called?

Monitor Queues (Wait)

```
Monitor BoundedBuffer {

  Condition notFull;
  …other variables…
  Condition notEmpty;

  void putResource () {
    …notFull.wait()…
    …notEmpty.signal()…
    return;
  }
  Resource getResource () {
    …
  }
}
```



Waiting to enter on monitor lock queue

Waiting on condition variable queues

Executing inside the monitor:

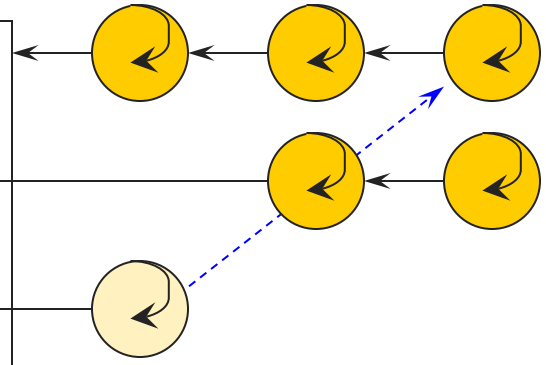
Calling wait adds the thread to a CV queue and releases the monitor lock (which then wakes up a waiting thread…)

Monitor Queues (Signal)

```
Monitor BoundedBuffer {

  Condition notFull;
  …other variables…
  Condition notEmpty;

  void putResource () {
    …notFull.wait()…
    …notEmpty.signal()…
    return;
  }
  Resource getResource () {
    …
  }
}
```



Waiting to enter on monitor lock queue

Waiting on condition variable queues

Executing inside the monitor:

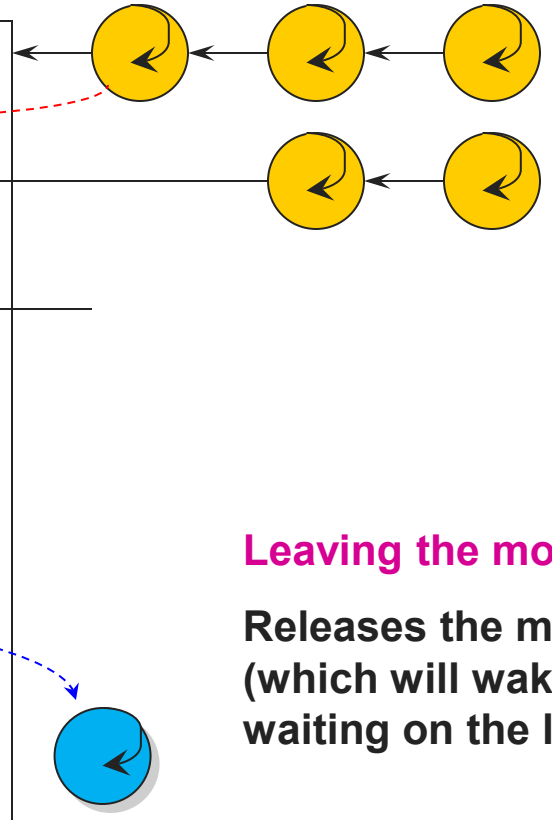
Calling signal removes a thread from the CV queue (lock is still held) and wakes it up

Monitor Queues (Return)

```
Monitor BoundedBuffer {

  Condition notFull;
  …other variables…
  Condition notEmpty;

  void putResource () {
    …notFull.wait()…
    …notEmpty.signal()…
    return;
  }
  Resource getResource () {
    …
  }
}
```



Waiting to enter on monitor lock queue

Waiting on condition variable queues

Leaving the monitor:

Releases the monitor lock (which will wake up a thread waiting on the lock…)

Condition Vars != Semaphores

- Although their operations can have similar names, they have very 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.decrement 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.increment increases the semaphore count, allowing future entry even if no thread is waiting
    - » Condition variables have no history

Signal Semantics

- signal() places a waiter on the ready queue, but signaler continues inside monitor
  - Known as “Mesa” semantics after an early operating system developed at Xerox PARC
- Conditional not necessarily true when waiter runs again
  - Returning from wait() is only a hint that something changed
  - Must recheck conditional case

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 (liveness) once it is in the monitor.

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

  void enterMonitor (...) {
    while (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 Readers and Writers

- Will have four methods: acquireRead, releaseRead, acquireWrite, and releaseWrite
- Monitored data: nr (number of readers) and nw (number of writers) with the monitor invariant

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

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

Monitor Readers and Writers

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

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

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

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

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

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

Monitor Readers and Writers

- add signal() and broadcast()

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

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

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

can we put a signal here?

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

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

can we put a signal here?

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?

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

Using Cond Vars & Locks

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

```
Lock lock;
Condition cond;

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

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

Bounded Buffer w/ Locks

```
class BoundedBuffer {
  Resource buffer[N];
  // Variables for indexing buffer
  // monitor invariant involves these vars
  Condition notFull;      // space in buffer
  Condition notEmpty;  // value in buffer

  void put_resource (Resource R) {
    lock.acquire();  // explicit lock
    while (buffer array is full)
        notFull.wait(lock);
    Add R to buffer array;
    notEmpty.signal(lock);
    lock.release();  // explicit lock
  }
```

```
  Resource get_resource() {
    lock.acquire();  // explicit lock
    while (buffer array is empty)
        notEmpty.wait(lock);
    Get resource R from buffer array;
    notFull.signal(lock);
    lock.release();  // explicit lock
    return R;
  }
} // end monitor
```

Monitors and Java

- A lock and condition variable are in every Java object
  - Later added explicit classes for locks / 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 method
      - 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 wait (Condition.sleep in Nachos)
  - Object.notify() is signal (Condition.wake)
  - Object.notifyAll() is broadcast (Condition.wakeAll)

OVERVIEW CLASS USE TREE PREVIEW NEW DEPRECATED INDEX SEARCH HELP Java SE 25 & JDK 25

java.base > java.lang > Object

Search documentation (type /)

Contents Filter contents

Description
Constructor Summary
Method Summary
Constructor Details
Object()
Method Details
getClass()
hashCode()

final void	notify()	Wakes up a single thread that is waiting on this object's monitor.
final void	notifyAll()	Wakes up all threads that are waiting on this object's monitor.
String	toString()	Returns a string representation of the object.
final void	wait()	Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*.
final void	wait(long timeoutMillis)	Causes the current thread to wait until it is awakened, typically by being *notified* or

Modern Languages

- Modern languages provide locks and condition variables for synchronization and coordination
  - C, C++, C#, Java, Go, Rust, …
  - Most common form of synchronization you will encounter
  - If curious, see link to supplementary material on course page
- Typically locks are explicit
  - Programmers have to use acquire and release explicitly
    - » C++ and Rust have “release on return” language semantics
    - » A half-way monitor implementation…
  - Even Java eventually added separate classes (Lock, Condition) for flexibility

Summary

- Semaphores
  - Increment/decrement implement atomic counters
  - Threads block until the semaphore counter is positive
  - A single mechanism for mutual exclusion and coordination
- Monitors
  - Provide mutual exclusion across all procedures/methods
    - » Only one thread executes in the monitor at a time
  - Condition variables enable threads to block within a monitor
  - Language support can make lock management automatic
  - Common in modern programming languages

Next time…

- Read Chapters 7, 8, 32