Higher-Level Synchronization

• We looked at using locks to provide mutual exclusion
• Locks work, but they have limited semantics
  ♦ Just provide mutual exclusion
• Instead, we 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 (mutex) and counting
  ♦ **Monitors**: mutexes 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 can also be used as atomic counters
  - More later
- Semaphores are “integers” that support two operations:
  - Semaphore::Wait(): decrement, block until semaphore is open
    - Also P(), after the Dutch word for “try to reduce” (also test, down)
  - Semaphore::Signal: increment, allow another thread to enter
    - Also V() after the Dutch word for increment, up
  - That's it! No other operations – not even just reading its value
- Semaphore safety property: the semaphore value is always greater than or equal to 0
Blocking in Semaphores

- Associated with each semaphore is a queue of waiting processes.
- When `wait()` is called by a thread:
  - If semaphore is `open`, thread continues.
  - If semaphore is `closed`, thread blocks on queue.
- Then `signal()` opens the semaphore:
  - If a thread is waiting on the queue, the thread is unblocked.
  - If no threads are waiting on the queue, the signal is remembered for the next thread.
    - In other words, `signal()` has “history” (c.f., condition vars later).
    - This “history” is a counter.
Semaphore Types

- 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
Using Semaphores

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

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

Threads

![Diagram showing the use of wait and signal with a critical section and undefined behavior after a signal](image)

It is undefined which thread runs after a signal
Semaphores in Nachos

\[
\begin{align*}
P() \{ & \text{ // wait} \\
& \textit{Disable interrupts}; \\
& \text{if (value == 0) \{}
\begin{align*}
& \text{add currentThread to waitQueue;} \\
& \text{KThread.sleep(); // currentThread}
\end{align*} \\
& \text{value = value – 1;}
\textit{Enable interrupts};
\}
\end{align*}
\]

\[
\begin{align*}
V() \{ & \text{ // signal} \\
& \textit{Disable interrupts}; \\
& \text{thread = get next on waitQueue;}
\text{thread.ready();}
\text{value = value + 1;}
\textit{Enable interrupts};
\}
\end{align*}
\]

- To reference current thread: KThread.currentThread()
- 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

Semaphore.P () { // wait
  Disable interrupts;
  if (value == 0) {
    add currentThread to waitQueue;
    KThread.sleep(); // currentThread
  }
  value = value – 1;
  Enable interrupts;
}

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

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

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

[Semaphore.P]
Disable interrupts;
if (value == 0) {
  add currentThread to waitQueue;
  KThread.sleep(); // currentThread
}
value = value – 1;
Enable interrupts;

[KThread.yield]
(Returns from runNextThread)
Enable interrupts;
Using Semaphores

• 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 more interesting examples
  ♦ 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: \( (\#r \geq 0) \land (0 \leq \#w \leq 1) \land ((\#r > 0) \Rightarrow (\#w = 0)) \)

• 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 {
    wait(w_or_r); // lock out readers
    Write;
    signal(w_or_r); // up for grabs
}

reader {
    wait(mutex);    // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(w_or_r); // synch w/ writers
    signal(mutex);  // unlock readcount
    Read;
    wait(mutex);    // lock readcount
    readcount -= 1; // one fewer reader
    if (readcount == 0)
        signal(w_or_r); // up for grabs
    signal(mutex);  // unlock readcount
}
Readers/Writers Notes

- `w_or_r` coordinates between readers and writers
  - writer wait/signal, reader wait/signal 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 the signal is above “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 (easier to think about)
  - The buffer set allows each to run without explicit handoff
- Safety:
  - Sequence of consumed values is prefix of sequence of produced values
  - If \( nc \) is number consumed, \( np \) number produced, and \( N \) the size of the buffer, then \( 0 \leq np - nc \leq N \)
Bounded Buffer (2)

- $0 \leq np - nc \leq N$ and $0 \leq (nc - np) + N \leq N$

- Use three semaphores:
  - empty – count of empty buffers
    » Counting semaphore
    » $\text{empty} = (nc - np) + N$
  - full – count of full buffers
    » Counting semaphore
    » $np - nc = \text{full}$
  - 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;
        wait(empty); // wait for empty buffer
        wait(mutex); // lock buffer list
        Add resource to an empty buffer;
        signal(mutex); // unlock buffer list
        signal(full);  // note a full buffer
    }
}

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

April 18, 2019  CSE 120 – Lecture 6 – Semaphores and Monitors
Bounded Buffer (4)

- Why need the mutex at all?
- Where are the critical sections?
- What happens if operations on mutex and full/empty are switched around?
  - The pattern of signal/wait on full/empty is a common construct often called an interlock
- Producer-Consumer and Bounded Buffer are classic examples of synchronization problems
Semaphore Questions

- Are there any problems that can be solved with counting semaphores that cannot be solved with mutex semaphores?
- Does it matter which thread is unblocked by a signal operation?
  - Hint: consider the following three processes sharing a semaphore `mutex` that is initially 1:

```c
while (1) {
    wait(mutex);
    // in critical section
    signal(mutex);
}
```

```c
while (1) {
    wait(mutex);
    // in critical section
    signal(mutex);
}
```

```c
while (1) {
    wait(mutex);
    // in critical section
    signal(mutex);
}
```
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 and 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
    » 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? (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(empty)” by reader in bounded buffer?

When first thread exits, another can enter. Which one is undefined.
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 enterMonitor (...) {
    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
  }
}
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
    » Also called wait (Java, C++), sleep (Nachos, C#)
  ♦ **Signal** – wakeup one waiting thread
    » Also called wake (Nachos, C#), notify (Java), notify_one (C++)
  ♦ **Broadcast** – wakeup all waiting threads
    » Also called wakeAll (Nachos, C#), notifyAll (Java), notify_all (C++)

• 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** {  
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);  
}  
} // end monitor

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

What happens if no threads are waiting when signal is called?
Monitor Queues (Wait)

Monitor bounded_buffer {

Condition not_full;
…other variables…
Condition not_empty;

void put_resource () {
…wait(not_full)…
…signal(not_empty)…
return;
}

Resource get_resource () {
…
}

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
Monitor Queues (Signal)

Monitor `bounded_buffer` {

Condition `not_full`;
...`other variables`...
Condition `not_empty`;

void `put_resource` () {
  ...`wait(not_full)`...
  ...`signal(not_empty)`...
  return;
}
Resource `get_resource` () {
  ...
}

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)
Monitor bounded_buffer {

  Condition not_full;
  ...other variables...
  Condition not_empty;

  void put_resource () {
    ...wait(not_full)...
    ...signal(not_empty)...
    return;
  }

  Resource get_resource () {
    ...
  }
}

Waiting to enter on monitor lock queue
Waiting on condition variable queues
Leaving the monitor:
Releases the monitor lock
Condition Vars != Semaphores

- Condition variables != semaphores
  - Although their operations can have the same names, they have entirely different semantics
  - However, they each can be used to implement the other

- Access to the monitor is controlled by a lock
  - `wait()` blocks the calling thread, and gives up the lock
    - To call wait, the thread has to be in the monitor (hence has lock)
    - Semaphore::wait just blocks the thread on the queue
  - `signal()` causes a waiting thread to wake up
    - If there is no waiting thread, the signal is lost
    - Semaphore::signal increases the semaphore count, allowing future entry even if no thread is waiting
    - Condition variables have no history
**Signal Semantics**

- `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
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++;
  }

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

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

  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?
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:

```c
Lock lock;
Condition cond;

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

- Must acquire lock before you can wait (similar to needing interrupts disabled to call sleep() in Nachos)
- Wait atomically releases lock and blocks until signal()
- Wait atomically acquires lock before it returns
Monitors and Java

- A lock and condition variable are in every Java object
  - No explicit classes for locks or condition variables
- Every object is/has a monitor
  - At most one thread can be inside an object’s monitor
  - A thread enters an object’s monitor by
    - Executing a method declared synchronized
      - Can mix synchronized/unsynchronized methods in same class
    - Executing the body of a synchronized statement
      - Supports finer-grained locking than an entire 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)
Modern Languages

• Modern languages provide some form of locks and condition variables for synchronization and coordination
  ♦ C, C++, C#, Java, Go, Rust, …
  ♦ Most common form of synchronization you will encounter

• Typically locks are explicit
  ♦ Programmers have to use acquire and release explicitly
  ♦ Even Java eventually added separate classes (Lock, Condition) for flexibility
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
Next time...

- Read Chapters 7, 8, 32
Race Conditions

What is the range of possible values for x? Why?

```c
int x = 0;
int i, j;

void AddToX() {
    for (i = 0; i < 100; i++) x++;
}

void SubFromX() {
    for (j = 0; j < 100; j++) x--;
}
```
Instance Variables

- Using explicit locks and CVs (common in languages)
- Bug: race condition on seq (instance variables shared)
  - What is a sequence of dangerous context switches?
Local Variables

- Local variables are private (not shared) across multiple threads

```java
Class SequenceNum {
    Lock lock;
    int seq;

    double next() {
        int result;
        lock.acquire();
        seq = seq + 1;
        result = seq;
        lock.release();
        return result;
    }
}
```

Assign in critical section
**Common Pitfall (actual bug in Linux device driver)**

```c
void mptctl_simplified(unsigned long arg) {
    mpt_ioctl_header khdr, __user *uhdr = (void __user *) arg;
    MPT_ADAPTER *iopc = NULL;

    // first fetch
    if (copy_from_user(&khdr, uhdr, sizeof(khdr)))
        return -EFAULT;

    // dependency lookup
    if (mpt_verify_adapter(khdr.ioctnum, &iopc) < 0 || iopc == NULL)
        return -EFAULT;

    // dependency usage
    mutex_lock(&iopc->ioctl_cmds.mutex);
    struct mpt_fw_xfer kfwdl, __user *ufwdl = (void __user *) arg;
    if (copy_from_user(&kfwdl, ufwdl, sizeof(struct mpt_fw_xfer)))
        return -EFAULT;
    mptctl_do_fw_download(kfwdl.ioctnum, ....);
    mutex_unlock(&iopc->ioctl_cmds.mutex);
}
```

Fig. 1: A dependency lookup double-fetch bug, adapted from __mptctl_ioctl in file drivers/message/fusion/mptctl.c
Signal

• Does the order of setting the flag and calling signal change the correctness?

lock.acquire();
...
while (flag != 1) {
    cv.wait();
}
...
lock.release();

lock.acquire();
...
flag = 1;
cv.signal();
...
lock.release();

lock.acquire();
...
cv.signal();
flag = 1;
...
lock.release();
Synchronization Practice

- Event synchronization (e.g., Win32)
- Event::Wait blocks if and only if Event is unsignaled
- Event::Signal makes Event signaled, wakes up blocked threads
- Once signalled, an Event remains signaled until deleted
- Use locks and condition variables (e.g., as in Nachos)