Two types of synchronization

Mutual exclusion

- Ensure that only 1 thread (or more generally, fewer than N threads) is in a critical section at once
- Lock/unlock

Ordering constraints

- Used when thread should wait for some event (not just another thread leaving a critical section)
- Used to enforce before-after relationships
- E.g. dequeuer wants to wait for enqueuer to add something to the queue

Monitors

Note that this differs from Tanenbaum's treatment

Monitors use separate mechanisms for the types of synchronization

- Use **locks** for mutual exclusion
- Use **condition variables** for ordering constraints

A monitor = a lock + the condition variable associated with the lock

Condition variables

Main idea: make it possible for thread to sleep inside a critical section by **atomically**

- Release the lock
- Put the thread on a wait queue and go to sleep

Each condition variable has a queue of waiting threads (i.e. threads that are sleeping, waiting for a certain condition)

Each condition variable is associated with one lock

Operations on condition variables

- **Wait**: atomically release lock, put thread on condition wait queue, go to sleep (i.e. start to wait for wakeup)
  When wait returns it automatically re-acquires the lock.
- **Signal**: wake up a thread waiting on this condition variable (if any)
- **Broadcast**: wake up all threads waiting on this condition variable (if any)

Note that thread must be holding lock when it calls wait

Should thread re-establish the invariant before calling wait?

Thread-safe queue with monitors

```
enqueue () {
  lock (queueLock)
  find tail of queue
  add new element to tail of queue




  unlock (queueLock)
}

dequeue () {
  lock (queueLock)









  remove item from queue
  unlock (queueLock)
  return removed item
}
```

Mesa vs. Hoare monitors

So far we have described Mesa monitors

- When waiter is woken, it must contend for the lock with other threads
- Hence, it must re-check the condition

What would be required to ensure that the condition is met when the waiter returns from wait and starts running again?

Hoare monitors give special priority to the woken-up waiter

- Signaling thread gives up lock 9hence signaler must re-establish invariant before calling signal)
- Woken-up waiter acquires lock
- Signaling thread re-acquires lock after waiter unlocks

We'll stick to Mesa monitors (as most operating systems do)

Tips for programming with monitors

List the shared data needed to solve the problem

Decide which locks (and how many) will protect which data

- More locks (protecting finer-grained data) allows different data to be accessed simultaneously, but is more complex
- One lock will *usually* enough in this class

Put lock … unlock calls around code the uses shared data

List before-after conditions

- One condition variable per condition
- Condition variable's lock should be the lock that protects the shared data that is used to evaluate the condition

Call wait() when thread needs to wait for a condition to be true; use a while loop to re-check condition after wait returns (aka "loop before you leap")

Call signal when a condition changes that another thread might be interested in

Make sure invariants are established whenever a lock is not held (i.e. before you call unlock and before you call wait)

Producer-consumer (bounded buffer)

Problem: producer puts things into a shared buffer, consumer takes them out. Need synchronization for coordinating producer and consumer.

- E.g. Unix pipeline (gcc calls cpp | cc1 | cc2 | as)
- Buffer between producer and consumer allows them to operate somewhat independently. Otherwise must operate in lockstep (producer puts one thing in buffer, then consumer takes it out, then producer adds another, then consumer takes it out, etc.)

E.g. soda machine

- Delivery person (producer) fills machine with sodas
- Students (consumer) buy sodas and drink them
- Soda machine has infinite space

Producer-consumer using monitors

Variables

- Shared data for the soda machine (assume machine can hold "max" cans)
- numSodas (number of cans in the machine)

One lock (sodaLock) to protect this shared data

- fewer locks make the programming simpler, but allow less concurrency

Ordering constraints

- consumer must wait for producer to fill buffer if all buffers are empty (ordering constraint)
- producer must wait for consumer to empty buffer if buffers are completely full (ordering constraint)

What if we wanted to have producer continuously loop? Can we put the loop inside the lock … unlock region?

Can we use only 1 condition variable?

Can we always use broadcast() instead of signal()?

Reader/writer locks using monitors

With standard locks, threads acquire the lock in order to read shared data. This prevents any other threads from accessing the data. Can we allow more concurrency without risking the viewing of unstable data?

Problem definition

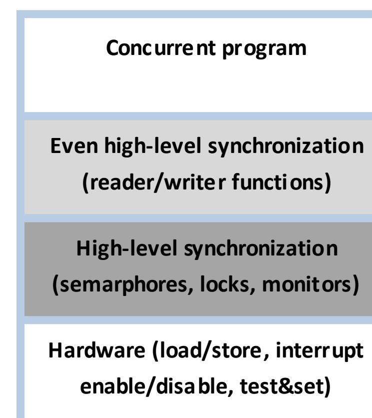
- shared data that will be read and written by multiple threads
- allow multiple readers to access shared data when no threads are writing data
- a thread can write shared data only when no other thread is reading or writing the shared data

Interface: two types of functions to allow threads different types of access

- readerStart ()
- readerFinish ()
- writerStart ()
- writerFinish ()

- many threads can be in between a readerStart and readerFinish (only if there are no threads who are between a writerStart and writerFinish)
- only 1 thread can be between writerStart and writerFinish

Implement reader/writer locks using monitors. Note the increased layering of synchronization operations



Monitor data (this is not the application data. Rather, it's the data needed to implement readerStart, readerFinish, writerStart, and writerFinish)

- what shared data is needed to implement reader/writer functions?

- Use one lock (RWlock)
- Condition variables?

In readerFinish(), could I switch the order of “numReaders—“ and “broadcast”?

If a writer finishes and there are several waiting readers and writers, who will win (i.e. will writerStart return, or will 1 readerStart, or will multiple readerStarts)?

How long will a writer wait?

How to give priority to a waiting writer?

Why use broadcast?

Note that all waiting readers and writers are woken up each time any thread leaves. How can we decrease the number of spurious wakeups?

Reader-writer functions are very similar to standard locks

- Call readerStart before you read the data (like calling lock())
- Call readerFinish after you are done reading the data (like calling unlock())
- Call writerStart before you write the data (like calling lock ())
- Call writerFinish after you are done writing the data (like calling unlock())

These functions are known as "reader-writer locks".

- Thread that is between readerStart and readerFinish is said to "hold a read lock"
- Thread that is between writerStart and writerFinish is said to "hold a write lock"

Compare reader-writer locks with standard locks

Semaphores

Semaphores are like a generalized lock

A semaphore has a non-negative integer value (>= 0) and supports the following operations

- **Down**: wait for semaphore to become positive, then decrement semaphore by 1 (originally called "P" for the Dutch "proberen")

- **Up**: increment semaphore by 1 (originally called "V" for the Dutch "verhogen"). This wakes up a thread waiting in down(), if there are any.
- Can also set the initial value of the semaphore

The key parts in down() and up() are atomic

- Two down() calls at the same time can't decrement the value below 0

Binary semaphore

- Value is either 0 or 1
- Down() waits for value to become 1, then sets it to 0
- Up() sets value to 1, waking up waiting down (of any)

Can use semaphores for both types of synchronization

Mutual exclusion

- Initial value of semaphore is 1 (or more generally N)

```
cown()
<critical section>
up()
```

- Like lock/unlock, but more general
- Implement lock as a binary semaphore, initialized to 1

Ordering constraints

- Usually (not always) initial value is 0
- E.g. thread A wants to wait for thread B to finish before continuing

```
semaphore initialized to 0

A                           B
  down()                      do task
  continue execution          up()
```

Solving producer-consumer with semaphores

Semaphore assignments

- mutex: ensures mutual exclusion around code that manipulates buffer queue (initialized to 1)
- fullBuffers: counts the number of full buffers (initialized to 0)
- emptyBuffers: counts the number of empty buffers (initialized to N)

Why do we need different semaphores for fullBuffers and emptyBuffers?

Does the order of the down() calls matter in the consumer (or the producer)?

Does the order of the up() calls matter in the consumer (or the producer)?

What (if anything) must change to allow multiple producers and/or multiple producers and/or multiple consumers?

What if there's 1 full buffer, and multiple consumers call down(fullBuffers) at the same time?

Comparing monitors and semaphores

Semaphores used for both mutual exclusion and ordering constraints

- elegant (one mechanism for both purposes)
- code can be hard to reason about and get right

Monitor lock is just like a binary semaphore that is initialized to 1

- lock() = down()
- unlock() = up()

Condition variables vs. semaphores

Condition variables	Semaphores
`while(cond) {wait();}`	`down();`
Conditional code in user program	Conditional code in semaphore definition
user writes customized condition	Condition specified by semaphore definition (wait if value == 0)
User provides shared variables, protect with lock	Semaphore provides shared variable (integer) and thread-safe operations on the integer
No memory of past signals	"remembers" past up() calls

Condition variables are more flexible than using semaphores for ordering constraints

- condition variables: can use arbitrary conditions to wait
- semaphores: wait if semaphore value equals 0

Semaphores work best if the shared integer and waiting condition (==0) maps naturally to the problem domain

Implementing threads on a uni-processor

So far, we've been assuming that we have enough physical processors to run each thread on its own processor

- but threads are useful also for running on a uni-processor (see web server example)
- how to give the illusion of infinite physical processors on a single processor?

Play analogy

Ready threads

What to do with a thread while it's not running

- must save its private state somewhere
- what constitutes private data for a thread?

This information is called the thread "context" and is stoned in a "thread control block" when the thread isn't running

- to save space, share code among all threads
- to save space, don't copy stack to the thread control block. Rather, use multiple stacks in the same address space, and just copy the stack pointer to the thread control block.

Keep thread control blocks of threads that aren't running on a queue of **ready** (but not running) threads

- thread state can now be running (the thread that's currently using the CPU), ready (ready to run, but waiting for the CPU), or blocked (waiting for a signal() or up() or unlock() from another thread)

Switching threads

Steps needed to switch to another thread

- thread returns control to the OS
- choose new thread to run
- save state of current thread (into its thread control block)
- load the context of the next thread (from its thread control block)
- run thread

Returning control to the OS

How does a thread return control back to the OS (so system can save the state of the current thread and run a new thread)?

Choosing the next thread to run

If no ready threads, just loop idly

- loop switches to a thread when one becomes ready

If 1 ready thread, run it

If more than 1 ready thread, choose one to run

- FIFO
- Priority queue according to some priority (more on this in CPU scheduling)

Saving state of current thread

How to save state of current thread?

- Save registers, PC, stack pointer (SP)
- This is very tricky assembly-language code
- Why won't the following code work?

```
100 save PC // i.e. value 100
101 switch to next thread
```

- In Project 1, we'll use Unix's swapcontext()

Loading context of next thread and running it

How to load the context of the next thread to run it?

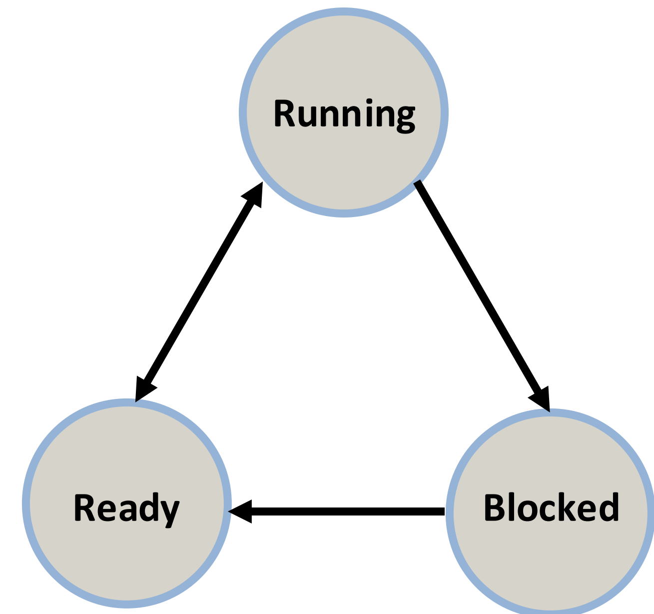
Example of thread switching

```
Thread 1
  print "start thread 1"
  yield ()
  print "end thread 1"

Thread 2
  print "start thread 2"
  yield()
  print "end thread 2"

yield
  print "start yield (thread %d)"
  switch to next thread (swapcontext)
  print "end yield (current thread %d)"
```

<u>thread 1's output</u>   <u>thread 2's output</u>

3 thread states

- Running (is currently using the CPU)
- Ready (waiting for the CPU)
- Blocked (waiting for some other event, e.g. I/O to complete, another thread to call unlock)



Creating a new thread

Overall: create state for thread and add it to the ready queue

- When saving a thread to its thread control block, we remembered its current state
- We can construct the state of a new thread as if it had been running and got switched out

Steps

- Allocate and initialize new thread control block
- Allocate and initialize new stack

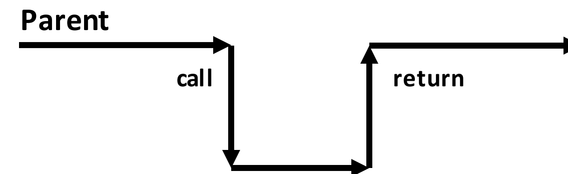
  Allocate memory for stack with C++ **`new`**

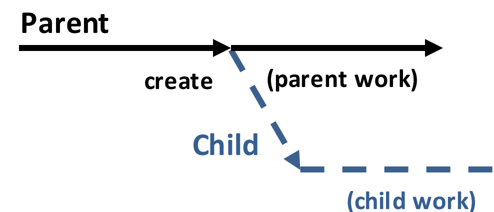
  Initialize the stack pointer and PC so that it looks like it was going to call a specified function. This is done with makecontext in Project 1.

- Add thread to ready queue

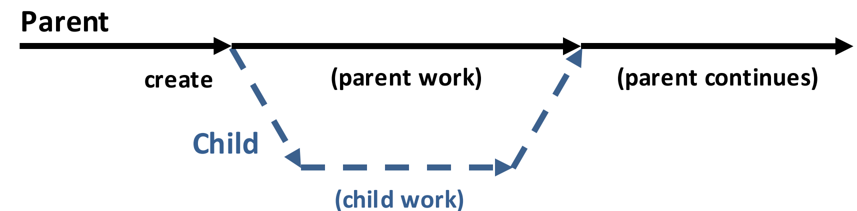
Unix's fork() is related but different. Unix's fork() creates a new process (a new thread in a new address space). In Unix, this new adderss space is a copy of the creator's address space.

thread_create is like an asynchronous procedure call





What if the parent thread wants to do some work in parallel with the child thread and then wait for the child thread to finish?



Does the following work?

```
parent () {
  thread_create
  print "parent works"
  print "parent continues"
}

child () {
  print "child works"
}
```

Does the following work?

```
parent () {
  thread_create
  print "parent works"
  thread_yield
  print "parent continues"
}

child () {
  print "child works"
}
```

Does the following work?

```
parent () {
  thread_create
  lock
  print "parent works"
  wait
  print "parent continues"
  unlock
}

child () {
  lock
  print "child works"
  signal
  unlock
}
```

Join(): wait for another thread to finish

```
parent () {
  thread_create
  lock
  print "parent works"
  unlock
  join
  print "parent continues"
}

child() {
  lock
  print "child works"
  unlock
}
```