Threads and concurrency

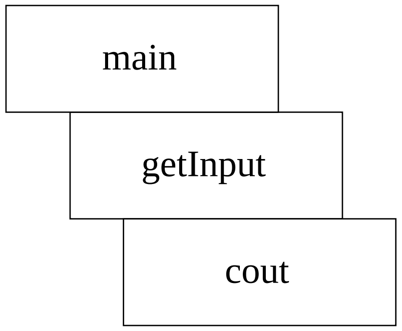
Motivation

- operating systems getting really complex
- multiple users, programs, I/O devices, etc.
- how to manage this complexity?

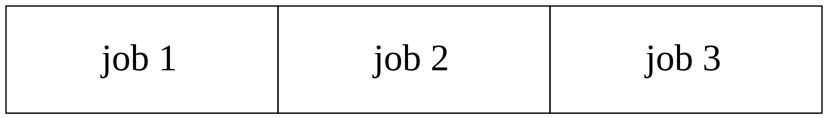
Decompose or separate hard problem into several simpler ones

Programs decompose into several rows (horizontal layers)

```
main() {
   getInput();
   computeResult();
   printOutput();
}

getInput() {
   cout();
   cin();
}

computeResult() {
   sqrt();
   pow();
}

printOutput() {
   cout();
}
```



Processes decompose mix of activities running on a processor into several parallel tasks (columns)



- each job can work independently of the others

Remember, for any area of OS, ask:

- What's the hardware interface?

- What's the application interface?

What’s in a process?

Definition of a process

- (informal) a program in execution. A running piece of code along with all the things the program can read/ write

  note that process != program

- (formal) one or more **threads** in their own **address space**

Play analogy

Thread

- sequence of executing instructions from a program (i.e. the running computation)
- active
- play analogy: the acting being done by an actor in the play

Address space

- all the data the process uses as it runs
- passive (acted upon by the thread)
- play analogy: all the objects on the stage in a play

Types of data in the address space

Multiple threads

Can have several threads in a single address space
- play analogy: several actors on a single set. Sometimes interact (e.g. dance together), sometimes do independent tasks

Private state for a thread vs. global state shared between threads
- what private thread must a thread have?

- other state is shared between all threads in a process

Upcoming lectures

Concurrency: multiple threads active at one time (multiple threads could come from one process, or from multiple processes)
- thread is the unit of concurrency
- two main topics:
  - how multiple threads can cooperate on a single task
  - how multiple threads can share a single CPU

Address space
- address space is the unit of state partitioning
- main topic: how multiple address spaces can share a single physical memory efficiently and safely

Can threads truly be independent?

Possible to have multiple threads on a computer system that don’t cooperate or interact at all?

- what about multiple programs that are related, e.g. mail program reads a PDF attachment and starts acroread process to display the attachment?

- what about multiple independent programs on a single computer, e.g. running Quake and 482 project at the same time?

Two possible sources of sharing

Correct example of non-interacting threads

Web server example

But if threads are cooperating, is it still a helpful abstraction to think of multiple threads? Or is it simpler to think of a single thread doing multiple things?

How to build a web server that receives multiple, simultaneous requests, and that needs to read web pages from disk to satisfy each request?

Handle one request at a time

- easy to program, but slow. Can’t overlap disk requests with computation or with network receive

Finite-state machine with asynchronous I/Os

- need to keep track of multiple outstanding requests

```
request 1 arrives
web server receives request 1
web server starts disk I/O 1a to satisfy
   request 1
request 2 arrives
web server receives request 2
web server starts disk I/O 2a to satisfy
   request 2
request 3 arrives
disk I/O 1a finishes
```

- At each point, web server must remember what requests have arrived and are being serviced, what disk I/Os are outstanding and which requests they belong to, and what disk I/Os still need to be done to satisfy each request.

Multiple cooperating threads

- each thread handles one request
- each thread can issue a **blocking** disk I/O, wait for I/O to finish, then continue with next part of its request
- even though thread blocks, other threads can make progress (and new threads can start to handle new requests)
- where is the state of each request stored?

Benefits and uses of threads

Thread system in operating system manages the sharing of the single CPU among several threads (e.g. allowing one thread to issue a blocking I/O and still allow other threads to make progress). Applications (or higher-level parts of the OS) get a simpler programming model)

Typical domains that use multiple threads

- program uses some slow resource, so it pays to have multiple things happening at once.

- physical control (e.g. airplane controller)
  slow component:

- window system (1 thread per window)
  slow component:

- network server
  slow component

- parallel programming (for using multiple CPUs)
  slow component:

Cooperating threads

First major topic in threads: how multiple threads can cooperate on a single task

- assume for now that we have enough physical processors to run each thread on its own processor
- later we’ll discuss how to give the illusion of infinite physical processors on a single processor

Ordering of events from different threads is non-deterministic

- processor speeds may vary

  e.g. after 10 seconds, different threads may have gotten differing amounts of work done

```
thread A --------------------------------->
thread B -         -        -          -       >
thread C -   -   -   -   -   -   -   -   -   -   ->
```

Non-deterministic ordering produces non-deterministic results

Printing example

- thread A: print ABC
- thread B: print 123
- possible outputs?

- impossible outputs?

- ordering within a thread is guaranteed to be sequential, but lots of ways to merge the ordering between threads
- what’s being shared between these two threads?

Arithmetic example

- (initially y=10)
- thread A: x = y + 1;
- thread B: y = y * 2;

- possible results?

Atomic operations

Example

- thread A: x=1
- thread B: x=2

- possible results?

- is 3 a possible output?

Before we can reason **at all** about parallel processes, we must know that some operation is **atomic**

Atomic: indivisible. Either happens in its entirety without interruption, or has yet to happen at all.

- no events from other threads can happen in between the start and end of an atomic event

In above example, if assignment to x is atomic, then only possible results are 1 and 2.

In print example above, what are the possible outputs if each print statement is atomic?

Print example above assumed printing a single character was atomic. What if printing a single character was **not** atomic?

On most machines, memory load and store are atomic

But many instructions are **not** atomic, e.g. double-precision floating point on a 32-bit machine (two separate memory operations)

If you don't have any atomic operations, you can't make one. Fortunately, the hardware folks give us atomic operations, and we can build up higher-level atomic primitives from there

Another example:

```
thread A                 thread B
i=0                      i=0
while (i < 10) {         while (i > -10) {
   i++                      i--
}                        }
print “A wins”           print “B wins”
```

Who will win?

Is it guaranteed that someone will win?

What if threads run at exactly the same speed and start close together? Is it guaranteed that it goes on forever?

- What if i++ and i-- are not atomic?

- Should you worry about this actually happening?

Non-deterministic interleaving makes debugging challenging

- Heisenbug: a bug that goes away when you look at it (via printf, via debugger, or just via re-running it)

Synchronizing between multiple threads

Must control the interleavings between threads

- order of some operations is irrelevant, because the operations are independent
- other operations are dependent and their order matters

All possible interleavings must yield a correct answer

- **a correct concurrent program will work no matter how fast the processors are that execute the various threads**

Try to constrain the thread executions as little as possible

Controlling the execution and order of threads is called “synchronization”

Too much milk

Problem definition

- Janet and Peter want to keep refrigerator stocked with at most one milk jug
- if either sees fridge empty, she/he goes to buy milk
- correctness properties: someone will buy milk if needed, but never more than one person buys milk

Solution #0 (no synchronization)

```
   Peter:                      Janet:
   if (noMilk) {               if (noMilk) {
      buy milk                    buy milk
   }                           }


      Peter                       Janet
3:00  look in fridge
         (no milk)
3:05  leave for Kroger
3:10  arrive at Kroger         look in fridge
                                  (no milk)
3:15  buy milk                 leave for Kroger
3:20  arrive home, put         arrive at Kroger
         in fridge
3:25                           buy milk
3:30                           arrive home, put
                                  milk in fridge.
                                  Too much milk!
```

First type of synchronization: mutual exclusion

Mutual exclusion

- ensure that only 1 thread is doing a certain thing at one time (others are excluded). E.g. only 1 person goes shopping at a time.

Critical section

- a section of code that needs to run atomically with respect to selected other pieces of code.
- if code A and code B are critical sections with respect to each other, then multiple threads should not be able to interleave events from A and B.
- critical sections must be atomic with respect to each other because they share data (or other resources, e.g. screen, refrigerator)
- e.g. in too much milk solution #0, critical section is “if (noMilk), buy milk. Peter and Janet’s critical sections must be atomic with respect to each other, i.e. events from these critical sections must not be interleaved.

<u>Too much milk (solution #1)</u>

Assume that the only atomic operations are load and store

Idea: leave note that you're going to check on the milk status, so other person doesn't also buy

```
Peter:                       Janet:
if (noNote) {                if (noNote) {
   leave note                   leave note
   if (noMilk) {                if (noMilk) {
      buy milk                     buy milk
   }                            }
   remove note                  remove note
}                            }
```

Does this work? If not, when could it fail?

Is solution #1 better than solution #0?

<u>Too much milk (solution #2)</u>

Idea: change the order of "leave note" and "check note". This requires labeled notes (otherwise you'll see your own note and think it was the other person's note)

```
Peter:                          Janet:
   leave notePeter                 leave noteJanet
   if (no noteJanet) {             if (no notePeter) {
      if (noMilk) {                   if (noMilk) {
         buy milk                        buy milk
      }                               }
   }                               }
   remove notePeter                remove noteJanet
```

Does this work? If not, when could it fail?

Too much milk (solution #3)

Idea: have a way to decide who will buy milk when both leave notes at the same time. Have Peter hang around to make sure job is done.

```
Peter:                          Janet:
   leave notePeter                 leave noteJanet
   while (noteJanet) {
      do nothing
   }
                                   if (no notePeter) {
   if (noMilk) {                      if (noMilk) {
         buy milk                        buy milk
   }                                  }
                                   }
   remove notePeter                remove noteJanet
```

Peter's "while (noteJanet)" prevents him from running his critical section at same time as Janet's

Proof of correctness

- (Janet) if no notePeter, then it's safe to buy because Peter hasn't started yet. Peter will wait for Janet to be done before checking milk status.
- (Janet) if notePeter, then Peter is in the body of the code and will eventually buy the milk (if needed). Note that Peter may be waiting for Janet to quit.
- (Peter) if no noteJanet, it's safe to buy (because Peter has already left notePeter, and Janet will check notePeter in the future)
- (Peter) if noteJanet, Peter hangs around and waits to see if Janet buys milk. If Janet buys, we're done. If Janet doesn't buy, Peter will buy.

Correct, but ugly

- complicated (and non-intuitive) to prove correct
- asymmetric
- Peter consumes CPU time while waiting for Janet to remove note. This is called **busy-waiting**.

Higher-level synchronization

Solution: raise the level of abstraction to make life easier for programmer

concurrent programs
high-level synchronization operations provided by software (e.g. locks, semaphores, monitors)
low-level atomic operations provided by hardware (e.g. load/store, interrupt enable/disable, test&set)

Locks (mutexes)

A lock prevents another thread from entering a critical section.
e.g. before shopping, leave a note on the fridge, so that both Peter and Janet don't go shopping

Two operations

- **lock()**: wait until lock is free, then acquire it

```
do {
   if (lock is free) {
      acquire lock
      break
   }
} while (1)
```

- **unlock()**: release lock

Why was the "note" in Too Much Milk solutions #1 and #2 not a good lock?

Four elements of locking

- lock is initialized to be free
- acquire lock before entering critical section
- release lock when exiting critical section
- wait to acquire lock if another thread already holds it

All synchronization involves waiting

Thread can be **running**, or **blocked** (waiting for something)

Locks make “Too much milk” really easy to solve!

```
Peter:                      Janet:
lock()                      lock()
if (noMilk) {               if (noMilk) {
   buy milk                    buy milk
}                           }
unlock()                    unlock()
```

But this prevents Janet from doing stuff while Peter is shopping. I.e. critical section includes the shopping time.

How to minimize the critical section?

Thread-safe queue with locks

```
enqueue() {


   /* find tail of queue */
   for (ptr=head; ptr->next != NULL;
         ptr = ptr->next);

   /* add new element to tail of queue */
   ptr->next = new_element;
   new_element->next = NULL;

}

dequeue() {


   /* if something on queue, then remove it */
   if (head->next != NULL) {
      element = head->next;
      head->next = head->next->next;
   }


   return(element);


}
```

What bad things can happen if two threads manipulate queue at same time?

Invariants for multi-threaded queue

Can enqueue() unlock anywhere?

This stable state is called an **invariant**, i.e. something that is supposed to "always" be true for the linked list, e.g. each node appears exactly once when traversing list from head to tail.

Is the invariant ever allowed to be false?

Invariant can only be broken when lock is held

- only the lock holder should be able to see the broken invariant

In general, must hold lock whenever you're manipulating shared data (i.e. whenever you're breaking the invariant of the shared data)

What if you're only reading shared data (i.e. you're not breaking the invariant)?

What about the following locking scheme:

```
enqueue() {
   lock
   find tail of queue
   unlock

   lock
   add new element to tail of queue
   unlock
}
```

What if you wanted to have dequeue() **wait** if the queue is empty?

Two types of synchronization

Mutual exclusion
- ensure that only 1 thread (or more generally, less than N threads) is in critical section
- 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 in Section 2.3.7

Monitors use separate mechanisms for the two types of synchronization
- use **locks** for mutual exclusion
- use **condition variables** for ordering constraints

A monitor = a lock + the condition variables associated with that lock

Condition variables

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

- releasing lock
- putting thread on 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 have described Mesa monitors
- when waiter is woken, it must contend for the lock with other threads
- hence must re-check 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
- signalling thread gives up lock (hence signaller must re-establish invariant before calling signal())
- woken-up waiter acquires lock
- signalling thread re-acquires lock after waiter unlocks

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

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 complicated
- one lock usually enough in this class

Put lock...unlock calls around the code that 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

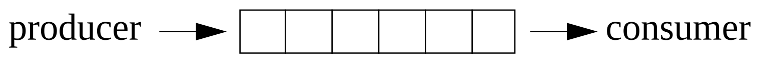
Call wait() when thread needs to wait for a condition to be true; use a while loop to re-check condition after wait returns

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

Make sure invariant is established whenever 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. coke machine
- delivery person (producer) fills machine with cokes
- students (consumer) buy cokes and drink them
- coke machine has finite space

Producer-consumer using monitors

Variables
- shared data for the coke machine (assume coke machine can hold “max” cokes)
- numCokes (number of cokes in machine)

One lock (cokeLock) 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 is 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 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

concurrent program coordinates its access to shared data by using reader/writer functions
even higher-level synchronization primitives (reader/writer functions)
high-level synchronization operations provided by software (e.g. locks, semaphores, monitors)
low-level atomic operations provided by hardware (e.g. load/store, interrupt enable/disable, test&set)

Monitor data (this is not the application data. Rather, it’s the data needed to implement readerStart, readerFinish, writer-Start, 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 readerStart)?

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 (if any)

Can use semaphores for both types of synchronization

Mutual exclusion
- initial value of semaphore is 1 (or more generally, N)

```
down()
<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() function calls matter in the consumer (or the producer)?

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

What (if anything) must change to allow 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 read and hard to get right

Monitor lock is just like a binary semaphore that is initialized to 1
- lock() = down()
- unlock() = up()

Condition variables vs. semaphores

condition variable	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 that 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 condition to wait
- semaphores: wait if semaphore value == 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 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 stored 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 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)

Dispatch loop

Main loop of the operating system runs threads

```
while(1) {

    load the context of the next thread
       that's ready to run (from its thread
       control block)

    run thread

    thread returns control to the dispatch loop

    save state of thread (into its thread
       control block)

    choose new thread to run

}
```

Or can think of it as a dispatch routine that each thread calls (sometimes involuntarily) to switch to the next thread

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

How to get control back to dispatch loop (so system can save the state of the current thread and run a new thread)?

How to save state of the 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 Solaris’s swapcontext()

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)

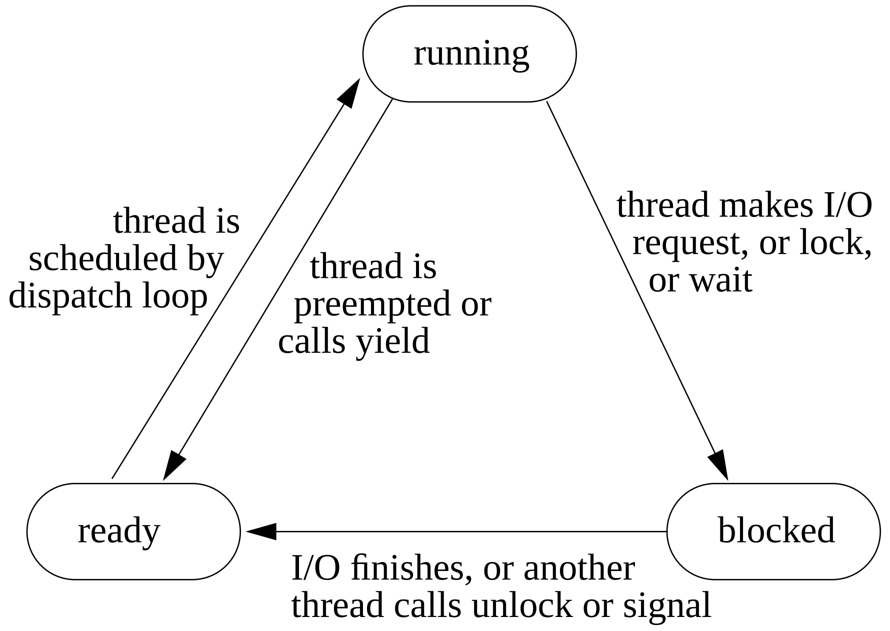
Example of thread switching

```
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)”


thread 1’s output          thread 2’s output
-----------------          -----------------
```

Thread states

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)


running
thread is scheduled by dispatch loop
thread is preempted or calls yield
thread makes I/O request, or lock, or wait
ready
blocked
I/O finishes, or another thread calls unlock or signal

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 new thread as if it had been running and got switched out

Steps

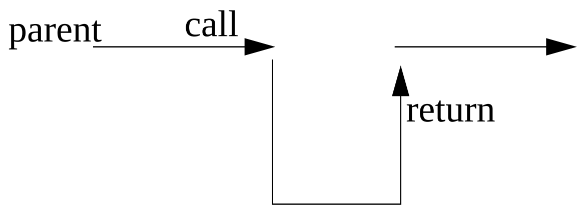
- allocate and initialize new thread control block
- allocate and initialize new stack

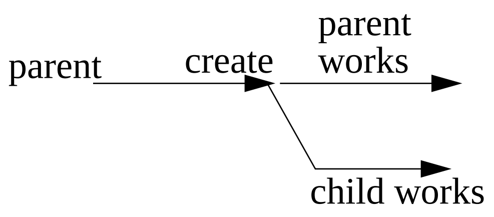
  allocate memory for stack with C++ new

  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

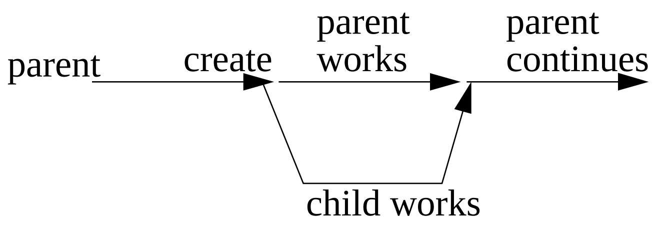
Unix's fork() is related but different. Unix's fork() creates a new process (a new thread in a new address space). In Unix, this new address space is a copy of the creator's address space.

thread_create is like an asynchronous procedure call





What if the parent thread wants to do some work in parallel with child thread, then wait for 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
}
```

Implementing locks — atomicity in the thread library

Concurrent programs use high-level synchronization operations

concurrent programs
high-level synchronization operations provided by software (e.g. locks, semaphores, monitors)
low-level atomic operations provided by hardware (e.g. load/store, interrupt enable/disable, test&set)

Implementing these high-level synchronization operations

- used by multiple threads, so they need to worry about atomicity (e.g. they use data structures shared across threads)
- can’t use the high-level synchronization operations themselves

Use interrupt disable/enable to ensure atomicity

On uniprocessor, operation is atomic as long as context switch doesn’t occur in middle of the operation

- how does thread get context switched out?

- prevent context switches at wrong time by preventing these events

With interrupt disable/enable to ensure atomicity, why do we need locks?

- user program calls interrupt disable before entering critical section, calls interrupt enable after leaving critical section (and makes sure not to call yield in the critical section)

Lock implementation #1 (disable interrupts with busy waiting)

```
lock() {
   disable interrupts
   while (value != FREE) {
      enable interrupts
      disable interrupts
   }
   value = BUSY
   enable interrupts
}

unlock() {
   disable interrupts
   value = FREE
   enable interrupts
}
```

Why does lock() disable interrupts in the beginning of the function?

Why is it ok to disable interrupts in lock()’s critical section (it wasn’t ok to disable interrupts while user code was running)?

Do we need to disable interrupts in unlock()?

Why does the body of the while enable, then disable interrupts?

Another atomic primitive: read-modify-write instructions

Interrupt disable works on a uniprocessor by preventing the current thread from being switched out

But this doesn't work on a multi-processor

- disabling interrupts on one processor doesn't prevent other processors from running
- not acceptable (or provided) to modify interrupt disable to stop other processors from running

Could use atomic load / atomic store instructions (remember Too Much Milk solution #3)

Modern processors provide an easier way with atomic read-modify-write instructions

- atomically {reads value from memory into a register, then writes new value to that memory location}

Test & set: atomically writes 1 to a memory location (set) and returns the value that used to be there (test)

```
test&set(X) {
   tmp = X
   X = 1
   return(tmp)
}
```

- note that only 1 process can see a transition from 0 -> 1

Exchange (x86)

- swaps value between register and memory

Lock implementation #2 (test&set with busy waiting)

```
(value is initially 0)

lock() {
   while (test&set(value) == 1) {
   }
}

unlock() {
   value = 0
}
```

If lock is free (value = 0), test&set sets value to 1 and returns 0, so the while loop finishes

If lock is busy (value = 1), test&set doesn't change the value and returns 1, so loop continues

Busy waiting

Problem with lock implementation #1 and #2

- waiting thread uses lots of CPU time just checking for the lock to become free. This is called "busy waiting"
- better for thread to go to sleep and let other threads run
- strategy for reducing busy-waiting: integrate the lock implementation with the thread dispatcher data structures and have lock code manipulate thread queues

Lock implementation #3 (interrupt disable, no busy-waiting)

Waiting thread gives up processor so that other threads (e.g. the thread with the lock) can run more quickly. Someone wakes up thread when the lock is free.

```
lock() {
   disable interrupts
   if (value == FREE) {
      value = BUSY
   } else {
      add thread to queue of threads waiting for
         this lock

      switch to next runnablethread
   }
   enable interrupts
}

unlock() {
   disable interrupts
   value = FREE
   if (any thread is waiting for this lock) {
      move waiting thread from waiting queue to
         ready queue
      value = BUSY
   }
   enable interrupts
}
```

This is a **handoff lock**

- thread calling unlock() gives lock to the waiting thread

Why have a separate waiting queue? Why not put waiting thread onto ready queue?

Interrupt disable/enable pattern

When should lock() re-enable interrupts before calling switch?

Enable interrupts before adding thread to wait queue?

```
lock() {
   disable interrupts
   ...
   if (lock is busy) {
      enable interrupts
      add thread to lock wait queue
      switch to next runnable thread
   }
```

When could this fail?

Enable interrupts after adding thread to wait queue, but before switching to next thread?

```
lock() {
   disable interrupts
   ...
   if (lock is busy) {
      add thread to lock wait queue
      enable interrupts
      switch to next runnable thread
   }
```

But this fails if interrupt happens after thread enable interrupts

- lock() adds thread to wait queue
- lock() enables interrupts
- interrupt causes preemption, i.e. switch to another thread. Preemption moves thread to ready queue. Now thread is on two queues (wait and ready)!

Also, switch is likely to be a critical section

Adding thread to wait queue and switching to next thread must be **atomic**

Solution: waiting thread leaves interrupts disabled when it calls switch. Next thread to run has the responsibility of re-enabling interrupts before returning to user code. When waiting thread wakes up, it returns from switch with interrupts disabled (from the last thread).

Invariant

- all threads promise to have interrupts disabled when they call switch
- all threads promise to re-enable interrupts after they get returned to from switch

```
Thread A                   Thread B

                           yield() {
                              disable interrupts
                              switch

   enable interrupts
}
<user code runs>
lock() {
   disable interrupts
   ...
   switch
                              back from switch
                              enable interrupts
                           }
                           <user code runs>
                           unlock() (move thread
                              A to ready queue)
                           yield() {
                              disable interrupts
                              switch
   back from switch
   enable interrupts
}
```

Lock implementation #4 (test&set, minimal busy-waiting)

Can’t implement locks using test&set without some amount of busy-waiting, but can minimize it

Idea: use busy waiting only to atomically execute lock code. Give up CPU if busy.

```
lock() {
   while(test&set(guard)) {
   }

   if (value == FREE) {
      value = BUSY
   } else {
      add thread to queue of threads waiting for
         this lock

      switch to next runnablethread
   }
   guard = 0
}
```

```
unlock() {
   while (test&set(guard)) {
   }

   value = FREE
   if (any thread is waiting for this lock) {
      move waiting thread from waiting queue to
         ready queue
      value = BUSY
   }
   guard = 0
}
```

Deadlock

Resources

- something needed by a thread
- a thread **waits** for resources
- e.g. locks, disk space, memory, CPU

Deadlock

- a circular waiting for resources, leading to the threads involved not being able to make progress

Example

```
thread A                 thread B
lock(x)                  lock(y)
lock(y)                  lock(x)
...                      ...
unlock(y)                unlock(x)
unlock(x)                unlock(y)
```

- can deadlock occur with code?

- will deadlock always occur with this code?

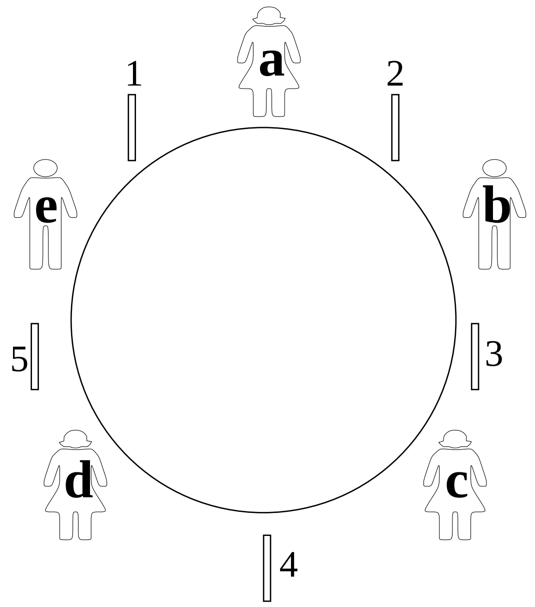
General structure of thread code

```
phase 1. while (not done) {
               acquire some resources
               work
         }
phase 2. release all resources
```

Assume phase 1 has finite amount of work

Dining philosophers

5 philosophers sitting around a round table, 1 chopstick in between each pair of philosophers (5 chopsticks total). Each philosopher needs two chopsticks to eat.



Algorithm for each philosopher

```
wait for chopstick on right to be free, then
   pick it up
wait for chopstick on left to be free, then
   pick it up
eat
put both chopsticks down
```

Can this deadlock?

Conditions for deadlock

Four conditions must all be true for deadlock to occur
- limited resource: not enough resources to serve all threads simultaneously
- hold and wait: threads hold resources while waiting to acquire other resources
- no preemption: thread system can't force thread to give up resource
- circular chain of requests

$$\begin{tikzcd}
 & \text{thread A} \arrow[dl] & \\
\text{resource 1} \arrow[dr] & & \text{resource 2} \arrow[ul] \\
 & \text{thread B} \arrow[ur] &
\end{tikzcd}$$

Strategies for handling deadlock

3 general strategies
- ignore
- detect and fix
- prevent

Detect and fix
- can detect by looking for cycles in the wait-for graph
- how to fix once detected?

Deadlock prevention

Idea is to eliminate one of the four necessary conditions

- increase resources to decrease waiting (this minimizes chance of deadlock)

- eliminate hold and wait

  wait until all resources you'll need are free, then grab them all at once

  this moves all the waiting to the beginning (when you aren't holding any resources)

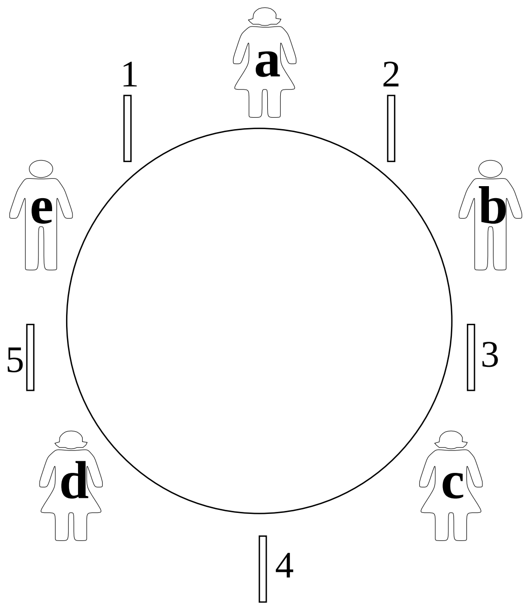
- allow preemption

  can preempt CPU by saving its state to thread control block and resuming later

  can preempt memory by swapping memory out to disk and loading it back later

  can we preempt the holding of a lock?

- eliminate circular chain of requests



Banker's algorithm

Similar to reserving all resources at beginning, but more efficient

State maximum resource needs in advance (but don't actually acquire the resources). When thread later tries to acquire a resource, banker's algorithm determines when it's safe to satisfy the request (and blocks the thread when it's not safe).

General structure of thread code

```
1. state maximum resource needed
2. while (not done) {
   acquire some resources
   work
}
3. release all resources
```

Preventing deadlock by requesting all resources at beginning would block thread in step #1 above (but step #2 can proceed without waiting)

In banker's algorithm, step #1 provides the information needed to determine when it's safe to satisfy each resource request in step #2.

"Safe" means guaranteeing the ability for all threads to finish (no possibility of deadlock)

Example: use banker's algorithm to model a bank loaning money to its customers

Bank has $6000. Customers sign up with bank and establish a credit limit (maximum resource needed). They borrow money in stages (up to their credit limit). When they're done, the return all the money.

Solution #1: reserve all resources when customer starts

```
Ann asks for credit limit of $2000 (bank oks)

Bob asks for credit limit of $4000 (bank oks)

Charlie asks for credit limit of $6000 (bank
   must say no, because this could lead to
   deadlock)
```

Solution #2: banker's algorithm

- bank approves all credit limits, but customer may have to wait when actually asking for the money.

```
Ann asks for credit limit of $2000 (bank oks)

Bob asks for credit limit of $4000 (bank oks)

Charlie asks for credit limit of $6000 (bank
   oks)


Ann takes out $1000 (bank has $5000 left)
Bob takes out $2000 (bank has $3000 left)
Charlie wants to take out $2000. Is this
   allowed?
```

Allow iff, after giving the money, there exists some sequential order of fulfilling all maximum resources (worst-case analysis)

- if give $2000 to Charlie, bank will have $1000 left
- Ann can finish even if she takes out her max (i.e. another $1000). When Ann finishes, she returns her money (bank will have $2000)
- After Ann finishes, Bob can take out his max (another $2000), then finish
- Then Charlie can finish, even if he takes out his max (another $4000).

What about this scenario?

```
Ann asks for credit limit of $2000 (bank oks)

Bob asks for credit limit of $4000 (bank oks)

Charlie asks for credit limit of $6000 (bank
   oks)

Ann takes out $1000 (bank has $5000 left)
Bob takes out $2000 (bank has $3000 left)
Charlie wants to take out $2500. Is this
   allowed?
```

Banker allows system to overcommit resources without introducing the possibility of deadlock. Sum of max resource needs of all current threads can be greater than total resources, as long as there's some way for the all the threads to finish without getting into deadlock.

How to apply banker's algorithm to dining philosophers?

Unfortunately, it's difficult to anticipate maximum resources needed

CPU scheduling

How should dispatch loop choose next thread to run? What are the goals of the CPU scheduler?

Minimize average response time
- average elapsed time to do each job

Maximize throughput of entire system
- rate at which jobs complete in the system

Fairness
- share CPU among threads in some "equitable" manner

First-come, first-served (FCFS)

FIFO ordering between jobs

No preemption (run until done)
- thread runs until it calls yield() or blocks on I/O
- no timer interrupts

Pros and cons

    + simple

    - short jobs get stuck behind long jobs

    - what about the user's interactive experience?

Example
- job A takes 100 seconds
- job B takes 1 second

```
time 0: job A arrives and starts
time 0+: job B arrives
time 100 job A ends (response time = 100); job
         B starts
time 101: job B ends (response time = 101)

average response time = 100.5
```

Round robin

Goal: improve average response time for short jobs

Solution: periodically preempt all jobs (viz. long-running ones)

Is FCFS or round robin more "fair"?

Example
- job A takes 100 seconds
- job B takes 1 second
- time slice of 1 second (a job is preempted after running for 1 second)

```
time 0: job A arrives and starts
time 0+: job B arrives
time 1: job A is preempted; job B starts
time 2: job B ends (response time = 2)
time 101: job A ends (response time = 101)

average response time = 51.5
```

Does round-robin always achieve lower response time than FCFS?

Pros and cons

+ good for interactive computing

- round robin has more overhead due to context switches

How to choose time slice?

- big time slice: degrades to FCFS
- small time slice: each context switch wastes some time

- typically a compromise, e.g. 10 milliseconds (ms)
- if context switch takes .1 ms, then round robin with 10 ms time slice wastes 1% of the CPU

STCF (shortest time to completion first)

STCF: run whatever job has the least amount of work to do before it finishes (or blocks for an I/O)

STCF-P: preemptive version of STCF

- if a new job arrives that has less work than the current job has remaining, then preempt the current job in favor of the new one

Idea is to finish the short jobs first

- improves response time of shorter jobs by a lot
- doesn't hurt the response time of longer jobs by too much

STCF gives optimal response time among non-preemptive policies

STCF-P gives optimal response time among preemptive policies (and non-preemptive policies)

I/O

- is the following job a "short" or "long" job?

```
while(1) {
   use CPU for 1 ms
   use I/O for 10 ms
}
```

Pros and cons

+ optimal average response time

- unfair. Short jobs can prevent long jobs from ever getting any CPU time (starvation)

- needs knowledge of future

STCF and STCF-P need knowledge of future

- it's often very handy to know the future :-)

- how to find out this information about the future time required by a job?

Example

```
job A
   compute for 1000 seconds

job B
   compute for 1000 seconds

job C
   while(1) {
      use CPU for 1 ms
      use I/O for 10 ms
   }
```

C can use 91% of the disk by itself. A or B can each use 100% of the CPU. What happens when we run them together?

Goal: keep both CPU and disk busy

FCFS

- if A or B run before C, they prevent C from issuing its disk I/O for up to 2000 seconds

Round robin with 100 ms time slice

```
CA---------B---------CA---------B---------...
 |--|
  C’s
  I/O
```

- disk is idle most of the time that A and B are running (about 10 ms disk time every 200 ms)

Round robin with 1 ms time slice

```
CABABABABABCABABABABABC...
 |--------| |--------|
  C’s       C’s
  I/O       I/O
```

- C runs more often, so it can issue its disk I/O almost as soon as its last disk I/O is done
- disk is utilized almost 90% of the time
- little effect on A or B’s performance
- general principle: first start the things that can run in parallel
- problem: lots of context switches (and context switch overhead)

STCF-P

- runs C as soon as its disk I/O is done (because it has the shortest next CPU burst)

```
CA---------CA---------CA--------- ...
 |--------| |--------| |---------|
   C’s         C’s         C’s
   I/O         I/O         I/O
```

Real-time scheduling

So far, we've focused on **average-case** analysis (average response time, throughput)

Sometimes, the right goal is to get each job done before its deadline (irrelevant how much before the deadline the job completes)

- video or audio output. E.g. NTSC (National Television Standards Committee) outputs 1 TV frame every 33 ms
- control of physical systems, e.g. auto assembly, nuclear power plants

This requires **worst-case** analysis

How do we do this in real life?

Earliest-deadline first (EDF)

Always run the job that has the earliest deadline (i.e. the deadline coming up next)

If a new job arrives with an earlier deadline than the currently running job, preempt the running job and start the new one

EDF is optimal—it will meet all deadlines if it's possible to do so

Example

```
   job A: takes 15 seconds, deadline is 20
      seconds after entering system
   job B: takes 10 seconds, deadline is 30
      seconds after entering system
   job C: takes 5 seconds, deadline is 10
      seconds after entering system
```

```
time--->
   0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85
A  +

B  +

C  +
```