Threads and concurrency

Motivation

- Operating systems are becoming extremely complex
- Multiple users, programs, I/O devices, etc
- How should we manage this complexity?

Decompose or separate hard problem into several simpler ones

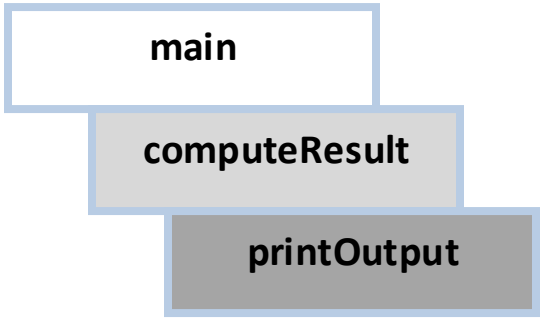
Programs decompose into several rows

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

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

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

void 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 the OS, ask:

- What interface does the hardware provide?

- What interface does the OS provide?

What is in a process?

Definition of a process

- Informal: a program in execution; a running piece of code along with all the things the code can read/write

  Note that a process is not a 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:

Address space

- All the data the process uses as it runs
- Passive (acted upon by threads)
- Play analogy: the objects on stage during a performance

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 they interact (e.g. do a little dance together) and sometimes they perform independently.

Private state for a thread vs. global state shared between threads

- What private state 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 can multiple threads cooperate on a single task?
  - How can multiple threads share a single CPU?

Address space

- Address space is the unit of state partitioning
- Main topic
  - How can multiple address spaces share a single physical memory efficiently and safely?

Can threads be truly independent?

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

- e.g. mail program reads a PDF attachment and starts acroread process to display the attachment?

- e.g. running an mp3 player and cps110 project on a computer 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 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 files 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 network service.

Event-driven 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 which 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, and continue with the next part of its request
- even though a thread blocks, other threads can make progress (and new threads can be started to handle in-coming requests)
- where is the state of each request stored?

Benefits and uses of threads

The thread system of an 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 controller (e.g. airplane controller)
  slow component:

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

- network server
  slow component:

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

Cooperating threads

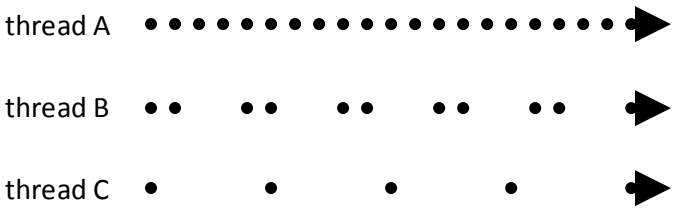
First major topic in threads: how multiple threads can cooperate on a single task

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

Ordering 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



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 there are lots of ways to merge the orderings between threads
- What is 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 cooperating threads, 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 the above example, if assignment to x is atomic, then the 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 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 finished";         print "B finished";
```

Who will win?

Is it guaranteed that someone will win?

What if threads start at exactly the same speed and start close together? Is it guaranteed to go 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 executing the various threads are**

Try to constrain the thread executions as little as possible

Controlling the execution and order of threads is called "synchronization"

Too much milk

Problem and definition

- Landon and Melissa want to keep their refrigerator stocked with at most one carton of milk
- If either sees the fridge empty, he/she has to go buy milk
- Correctness properties: someone will buy milk if needed, but never more than one person buys milk

Solution number zero (no synchronization)

```
      Landon:                    Melissa:
      if (noMilk) {              if (noMilk) {
      buy milk                   buy milk
      }                          }
```

```
      Landon                     Melissa
3:00  look in fridge
      (no milk)
3:05  go to Whole Foods
3:10                             look in fridge
                                 (no milk)
3:15  buy milk
3:20                             go to Whole Foods
3:25  arrive home, put
      milk in fridge
3:30                             buy milk
3:35                             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 such as a screen or refrigerator)
- In too much milk solution zero, the critical section is "if (noMilk), buy milk". Landon and Melissa's critical sections must be atomic with respect to each other, i.e. events from these critical sections must not be interleaved.

Too much milk (solution #1)

Assume the only atomic operations are load and store

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

```
Landon:                  Melissa:
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 zero?

Too much milk (solution #2)

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)

```
Landon:                      Melissa:
leave noteLandon             leave noteMelissa
if (no noteMelissa) {        if (no noteLandon) {
  if (noMilk) {                if (noMilk) {
    buy milk                     buy milk
  }                            }
}                            }
remove noteLandon            remove noteMelissa
```

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 Landon hang around to make sure the job is done.

```
Landon:                     Melissa:
leave noteLandon            leave noteMelissa
while (noteMelissa) {
  do nothing
}

                            if (no noteLandon) {
  if (noMilk) {               if (noMilk) {
    buy milk                    buy milk
  }                           }
}                           }
remove noteLandon           remove noteMelissa
```

Landon's "while (noteMelissa)" prevents him from running his critical section at the same time as Melissa's.

Proof of correctness:

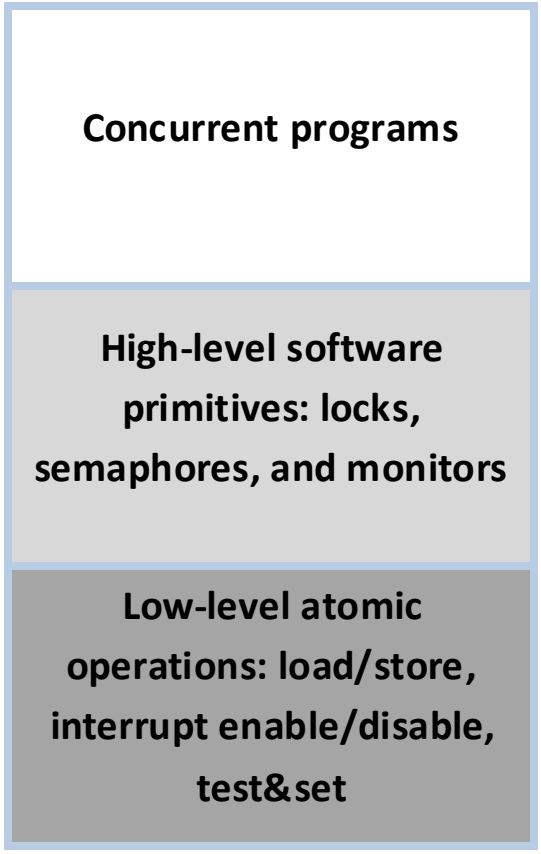
- Melissa: if no noteLandon, then it is safe to buy because Landon hasn't started yet. Landon will just wait for Melissa to be done before checking milk status.
- Melissa: if noteLandon, then Landon is in the body of the code and will eventually buy milk (if needed). Note that Landon may be waiting for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

Correct, but ugly

- Complicated and non-intuitive to prove correct
- Asymmetric
- Landon consumes CPU time while waiting for Melissa to remove her note. This is called **busy-waiting**.

Higher-level synchronization

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



Locks (mutexes)

A lock prevents another thread from entering a critical section. For example, lock the fridge while you're shopping for milk to prevent Landon and Melissa from both shopping.

Two operations

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

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

- **unlock ()**: release the lock

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

Four elements of locking

- lock is initially free
- acquire lock before entering critical section
- release lock when exiting a critical section
- wait to acquire lock if another thread already owns it

All synchronization involves waiting.

Threads can be **running** or **blocked** (waiting for something).

Locks make Too Much Milk really easy to solve!

```
Landon:                    Melissa:
lock ()                    lock ()
if (noMilk) {              if (noMilk) {
  buy milk                   buy milk
}                          }
unlock ()                  unlock ()
```

But this prevents Melissa from doing anything while Landon is shopping because the critical section includes the shopping time.

How can we minimize the time the lock is held?

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 () {
  element = NULL;

  /* if something on queue, 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 the queue at the 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. For example, each node must appear exactly once when traversing the list from head to tail.

Is the invariant ever allowed to be false?

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?