Threads and concurrency

• Motivation: operating systems getting really complex
  – Multiple users, programs, I/O devices, etc.
  – How to manage this complexity?
• Main techniques to manage complexity in programs?
  – Divide and conquer
  – Modularity and abstraction

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

Processes

• Processes decompose mix of activities running on a computer into several “independent” tasks that run in parallel

```
  job 1  job 2  job 3
```

• Process is the key abstraction that made it simple to run multiple things simultaneously
  – Each process need not know about the others
• Remember, for each area of OS, ask
  – What interface does the hardware provide?
  – What interface does the OS provide?
What’s a process?

- Informal definition
  - A program in execution. A running piece of code along with all the things the program can read/write
  - Process != program
- Formal definition
  - One or more threads in an address space
- Thread
  - Sequence of execution instructions from a program (i.e., the running computation)
  - Active
  - Play analogy
- Address space
  - All the memory data the process uses as it runs
  - Passive
  - Play analogy

Categories of data in an address space
Multiple threads

- Can have several threads in a single address space
  - Sometimes they interact
  - Sometimes they work independently
- State that is private to a thread

- State that is shared between threads

Upcoming topics

- Concurrency: multiple threads active at one time
  - Thread is the unit of concurrency
  - How multiple threads can cooperate to accomplish a single task
  - How multiple threads can share a limited number of CPUs

- Address spaces
  - Address space is the unit of state partitioning
  - How multiple address spaces can share a single physical memory efficiently, flexibly, and safely?
Web server example

• How to build a web server
  – Receives multiple simultaneous requests
  – Reads web pages from disk to satisfy each request
• Option 1: handle one request at a time
  Request 1 arrives
  Server receives request 1
  Server starts disk I/O 1a
  Request 2 arrives
  Server waits for I/O 1a to finish
  – Easy to program, but slow
  – Can’t overlap disks requests with computation, or with network

Event-driven web server (asynchronous I/O)

• Issue I/Os, but don’t wait for them to complete
  Request 1 arrives
  Server receives request 1
  Server starts disk I/O 1a to satisfy request 1
  Request 2 arrives
  Server receives request 2
  Server starts disk I/O 2a to satisfy request 2
  Request 3 arrives
  Disk I/O 1a finishes
  – Web server must remember
    • What requests are being serviced, and what stage they’re in
    • What disk I/Os are outstanding (and which requests they belong to)
    • Lots of extra state!
Multi-threaded web server

- One thread per request
  - Thread issues disk (or network) I/O, then waits for it to finish
  - Even though thread is blocked on I/O, other threads can run
  - Where is the state of each request stored?

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request 1 arrives</td>
<td>Request 2 arrives</td>
<td>Request 3 arrives</td>
</tr>
<tr>
<td>Receive request 1</td>
<td>Receive request 2</td>
<td>Receive request 3</td>
</tr>
<tr>
<td>Start disk I/O 1a</td>
<td>Start disk I/O 2a</td>
<td></td>
</tr>
<tr>
<td>Disk I/O 1a finishes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Continue handling request 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Benefits of threads

- Thread manager takes care of sharing CPUs among several threads
  - A thread can issue blocking I/Os, while other threads can still make progress
  - Private state for each thread
- Applications get a simpler programming model
  - The illusion of a dedicated CPU per thread
- Domains that use multiple threads
  - Multiple things happening at once
  - Usually some slow resource
  - Examples
    - Network server
    - Controlling a physical system
    - Window system
    - Parallel programming
Can threads truly be independent?

- Example 1: Microsoft Word
  - One thread formats document
  - Another thread spell checks document

- Example 2: desktop computer
  - One thread plays World of Warcraft
  - Another thread compiles EECS 482 project

- Two types of sharing

- Example of non-interacting threads:

Cooperating threads

- How multiple threads can cooperate on a single task
  - Assume each thread has a dedicated processor
  - Later we’ll talk about how to provide the illusion of infinite processors

- Main problem: ordering of events from different threads is non-deterministic
  - Speed of each processor is unpredictable

  Thread A ------------------------------------>
  Thread B - - - - - - - - - - - - - - - - - - - >
  Thread C - - - - - - - - - - - - - - - - - - - - - >

  - Global ordering of events
  - Many possible orderings (some of which may produce incorrect results)
Non-deterministic ordering produces non-deterministic results

• Printing example

Thread A
Print ABC

Thread B
Print 123

– Possible outputs?

– Impossible outputs?

– Ordering within thread is sequential, but many ways to merge per-thread order into a global order
– What’s being shared between these threads?

Non-deterministic ordering produces non-deterministic results

• Arithmetic example (y is initially 10)

Thread A
\[ x = y + 1 \]

Thread B
\[ y = y \times 2 \]

– What’s being shared between these threads?
– Possible results?

• Another arithmetic example (x is initially 0)

Thread A
\[ x = 1 \]

Thread B
\[ x = -1 \]

– Possible results?
– Impossible results?
Atomic operations

- Before we can reason at all about cooperating threads, we must know that some operation is **atomic**
  - Indivisible, i.e., happens in its entirety or not at all
  - No events from other threads can occur in between the start and end of an atomic operation
- Arithmetic example: what if assignment were atomic?
- Print example:
  - What if each print statement were atomic?
  - What if printing a single character were not atomic?

- Most computers
  - Memory load and store are atomic
  - Many other instructions are not atomic (e.g., double-precision floating point)
  - Need an atomic operation to build a bigger atomic operation

---

Example

<table>
<thead>
<tr>
<th>Thread A</th>
<th>x is shared and initialized to 0</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>while (x&lt;10) {</td>
<td>while (x&gt; -10) {</td>
<td></td>
</tr>
<tr>
<td>x++</td>
<td>x--</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>}</td>
<td></td>
</tr>
<tr>
<td>print “A finished”</td>
<td>print “B finished”</td>
<td></td>
</tr>
</tbody>
</table>

- Which thread will finish first?
- Is the winner guaranteed to print first?
- Is it guaranteed that someone will will?
- What if both threads run at exactly the same speed, and start close together? Is it guaranteed that both threads will loop forever?

- Non-deterministic interleaving makes debugging challenging
  - Heisenbug: a bug that occurs non-deterministically
Synchronization

• Must constrain the interleavings between threads
  – Some events are independent of each other, so their order is irrelevant
  – Other events are dependent, so their order matters
• All possible interleavings must produce a correct result
  – A correct concurrent program should work correctly no matter how fast the processors are that execute the threads
• Try to constrain 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 their refrigerator stocked with at most one milk jug
  – If either sees fridge empty, she/he goes to buy milk
• Solution #0 (no synchronization)

Peter
if (noMilk) {
  buy milk
}

Janet
if (noMilk) {
  buy milk
}
First type of synchronization: mutual exclusion

- Mutual exclusion
  - Ensure that only 1 thread is doing a certain thing at one time (other threads are excluded). E.g., only 1 person goes shopping at a time.
  - Constrains interleavings of threads: “not at the same time”
  - Does this remind you of any other concept we’ve talked about?

Critical section

- A section of code that needs to be 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 (code A and B mutually exclude each other). Often A and B are the same piece of code.
- Critical sections must be atomic with respect to each other because they access some shared resource
- In Too much milk, critical section is “if (no milk) buy milk”
Too much milk (solution #1)

- Assume the only atomic operations are load and store
- Leave note that you’re going to check on the milk, so other person doesn’t also buy

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

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

- Does this work?

- Is solution #1 better than solution #0?

Too much milk (solution #2)

- Change the order of “leave note” and “check note”. Notes need to be labelled (otherwise you’ll see your note and think the other person left it).

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

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

- Does this work?
Too much milk (solution #3)

- Decide who will buy milk when both leave notes at the same time. Peter hangs around to make sure job is done.

```java
Peter
leave notePeter
while (noteJanet) {
  do nothing
}
if (noMilk) {
  buy milk
}
remove notePeter

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

- Peter’s “while (noteJanet)” prevents him from entering the critical section at the same time as Janet

Proof of correctness

- Janet
  - if no notePeter, then Peter hasn’t started yet, so it’s safe to buy. Peter will wait for Janet to be done before checking.
  - if notePeter, then Peter is in the body of the code and will eventually buy milk if needed. Note that Peter may be waiting for Janet to exit.

- Peter
  - if no noteJanet, it’s safe to buy. Peter has already left notePeter, and Janet will check notePeter in the future.
  - if noteJanet, Peter waits to see what Janet does
    - Janet may have checked notePeter before Peter left note. In this case, Janet will buy milk.
    - Janet may have checked notePeter after Peter left note. In this case, Janet will not buy milk.
Analysis of solution #3

• Good
  – It works
  – What operations must be atomic?

• Bad
  – Complicated; not obviously correct
  – Asymmetric
  – Not obvious how to scale to three people
  – Peter consumes CPU time while waiting. This is called busy-waiting.

Higher-level synchronization

• Raise the level of abstraction to make life easier for programmers

<table>
<thead>
<tr>
<th>concurrent programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>high-level synchronization operations provided by software</td>
</tr>
<tr>
<td>e.g., locks, monitors, semaphores</td>
</tr>
<tr>
<td>low-level atomic operations provided by hardware</td>
</tr>
<tr>
<td>e.g., load/store, interrupt enable/disable, test&amp;set</td>
</tr>
</tbody>
</table>
Locks (mutexes)

- A lock prevents another thread from entering a critical section
  - e.g., lock fridge while you’re checking the milk status and 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?

Lock usage
- Lock is initialized to be free
- Thread acquires lock before entering critical section (waiting if needed)
- Thread that has acquired lock should release when done with critical section

All synchronization involves waiting
- Thread can be running or blocked

Locks make Too much milk easy to solve

```
Peter
milk.lock();
if (!noMilk) {
  buy milk
}
milk.unlock()

Janet
milk.lock();
if (!noMilk) {
  buy milk
}
milk.unlock()
```
Efficiency

- But this prevents Janet from doing things while Peter is buying milk.
- How to minimize the time the lock is held?

Thread-safe queue with locks

```c
enqueue(new_element) {
    // find tail of queue
    for (ptr=head; ptr->next != NULL; ptr = ptr->next) {} 
    // add new element to tail of queue
    ptr->next = new_element;
}

dequeue() {
    element = NULL;
    // 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 thread-safe queue

• Can enqueue() unlock anywhere?

• This stable state is called an invariant: a condition that is “always” true for the linked list. E.g., each node appears exactly one when traversing from head to tail.
  – Is invariant ever allowed to be false?

• Hold lock whenever you’re manipulating shared data, i.e., whenever you’re breaking the invariant

• What if you’re only reading the data?

Don’t break assumptions

enqueue(new_element) {
  lock
  // find tail of queue
  for (ptr=head; ptr->next != NULL; ptr = ptr->next) {}
  unlock

  lock
  // add new element to tail of queue
  ptr->next = new_element;
  unlock
}
• Does this work?
Fine-grained locking

• Instead of one lock for the entire queue, use one lock per node of the queue
  – This is called fine-grained locking (not needed until Project 4)
  – Pros and cons

• Lock each node as the queue is traversed, then release as soon as it’s safe, so other threads can also access the queue

• lock A
• get pointer to B
• unlock A
• lock B
• read B
• unlock B

• What bad thing could occur?

How to fix?

• Hand-over-hand locking
  – lock next node before releasing last node
  – Used in Project 4
Ordering constraints

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

```c
dequeue() {
    queue.lock();
    // wait for queue to be non-empty

    // remove element
    element = head->next;
    head->next = head->next->next;

    queue.unlock();
    return(element);
}
```

Ordering constraints: solution?

```c
dequeue() {

    // remove element
    element = head->next;
    head->next = head->next->next;

    queue.unlock();
    return(element);
}
```
Ordering constraints: solution?

dehue() {

    // remove element
    element = head->next;
    head->next = head->next->next;

    queue.unlock();
    return(element);
}

Avoiding busy waiting

- Have waiting dequeuer “go to sleep”
  - Put dequeuer onto a waiting list, then go to sleep
    if (queue is empty) {

        add myself to waiting list

        go to sleep
    }

- enqueuer wakes up sleeping dequeuer
When should dequeuer unlock?

enqueue()
lock
find tail of queue
add new element to tail of queue
if (dequeuer is waiting) {
    take waiting dequeuer off waiting list
    wake up dequeuer
}
unlock
dequeue()
...
if (queue is empty) {
    unlock
    add myself to waiting list
    sleep
}
...

When should dequeuer unlock?

enqueue()
lock
find tail of queue
add new element to tail of queue
if (dequeuer is waiting) {
    take waiting dequeuer off waiting list
    wake up dequeuer
}
unlock
dequeue()
...
if (queue is empty) {
    add myself to waiting list
    unlock
    sleep
}
...
Two types of synchronization

• Mutual exclusion
  – Ensures that only one thread is in critical section
  – “Not at the same time”
  – lock/unlock

• Ordering constraints
  – Used when thread must wait for another thread to do something
  – “Before after”
  – E.g., dequeuer must wait for enqueuer to add something to queue

Monitors

• Separate mechanisms for the two types of synchronization
  – Locks for mutual exclusion
  – Condition variables for ordering constraints

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

• Enable a thread to sleep inside a critical section, by
  – Releasing lock
  – Putting thread onto waiting list
  – Going to sleep
  – After being woken, call lock()

• Each condition variable has a list of waiting threads
  – These threads are “waiting on that condition”
  – Each condition variable is associated with a lock

• Operations on a condition variable
  – wait(): atomically release lock, add thread to waiting list, go to sleep.
    • Thread must hold the lock when calling wait()
    • Should thread re-establish invariant before calling wait()?
  – signal(): wake up one thread waiting on this condition variable. If no
    thread is currently waiting, then signal does nothing
  – broadcast(): wake up all threads waiting on this condition variable. If no
    thread is currently waiting, then broadcast does nothing

Thread-safe queue with monitors

```java
enqueue()
        queueMutex.lock()
        find tail of queue
        add new element to tail of queue
        if (dequuer is waiting) {
            take waiting dequeuer off waiting list
            wake up dequeuer
        }
        queueMutex.unlock()
}

dequeue()
        queueMutex.lock()
        remove item from queue
        queueMutex.unlock()
        return removed item
```
Mesa versus Hoare monitors

• So far, I’ve described Mesa monitors
  – When waiter is woken, it must contend for the lock with other threads
  – So it must re-check the condition it was waiting for
• 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 immediately gives up lock to woken-up waiter
  – Signalling thread reacquires lock after waiter unlocks

• We (and most operating systems) use Mesa monitors
  – Waiter is solely responsible for ensuring condition is met

How to program with monitors

• List the shared data needed for the problem
• Assign locks to each group of shared data
• Each thread tries to go as fast as possible, without worrying about other threads
• Two reasons a thread needs to pause
  – Mutual exclusion. Enforce with lock/unlock.
  – Before-after conditions
    • A thread can’t proceed because the condition of the shared state isn’t satisfactory
    • Some other thread must do something
    • Assign a condition variable for each situation
      – Condition variable belongs to the lock that protects the shared data used to evaluate the condition
• Use while(!condition) wait
• Call signal() or broadcast() when a thread changes something that another thread might be waiting for.
How to program with monitors

• Typical code

lock
while (!condition) {
    wait
}
do stuff
signal about the stuff you did
unlock

Producer-consumer (bounded buffer)

• Producer puts things into a shared buffer; consumer takes them out of
  shared buffer. Need to synchronize actions of producer and consumer.

  producer → [buffers] → consumer

• The buffer allows producers and consumers to operate somewhat
  independently. Used in many situations
  – Unix pipes
  – Communication systems
  – Coke machine
    • Delivery person (producer) fills coke machines with cokes.
      Produce 1 coke at a time.
    • Students (consumer) buy cokes.
    • Coke machine has finite space.
Producer-consumer with monitors

- Variables
  - Shared data describing state of coke machine buffers
  - numCokes (assume coke machine can hold at most MAX cokes)
  - One lock (cokeLock) to protect this data
- When must a thread pause?
  - Mutual exclusion (when acquiring a lock)
  - Consumer must wait if all buffers are empty
    - Use condition variable waitingConsumers
  - Producer must wait if all buffers are full
    - Use condition variable waitingProducers

Producer

Consumer

Producer

Consumer

take coke out of machine

Producer

add coke to machine
Producer-consumer with monitors

• 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 (waitingProducersAndConsumers)?

• Can we always use broadcast instead of signal?

Reader-writer locks

• With standard locks, threads acquire the lock to read (or write) shared data. This prevents other threads from accessing the data.

• Can we allow more concurrency without risking the viewing of unstable data?

• Problem definition
  – Shared data will be read and written by multiple threads
  – Allow multiple readers to access shared data, if no threads are writing data
  – A thread can write shared data only when no other thread is reading or writing the shared data
Reader-writer locks

- Implement a set of functions that a program can use to follow the “multiple-reader, single-writer” constraint
  - readerStart()
  - readerFinish()
  - writerStart()
  - writerFinish()

- Examples for Wolverine Access

  readerStart()  writerStart()
  print catalog  change catalog
  readerFinish() writerFinish()

Reader-writer locks with monitors

<table>
<thead>
<tr>
<th>concurrent programs coordinates its accesses to shared data by using readerStart, readerFinish, writerStart, writerFinish</th>
</tr>
</thead>
<tbody>
<tr>
<td>even higher-level synchronization primitives (readerStart, readerFinish, writerStart, writerFinish)</td>
</tr>
<tr>
<td>high-level synchronization operations provided by software (e.g., locks, monitors, semaphores)</td>
</tr>
<tr>
<td>low-level atomic operations provided by hardware (load/store, interrupt enable/disable, test&amp;set)</td>
</tr>
</tbody>
</table>
Implementing reader-writer locks with monitors

- Shared data needed to implement readerStart, readerFinish, writerStart, writerFinish
- Use one lock (rwLock)
- Condition variables
Implementing reader-writer locks with monitors

• In readerFinish, could I switch the order of numReaders-- and broadcast?

• What will happen if a writer finishes and there are several waiting readers and writers? Will writerStart return, or will 1 readerStart return, or will all readerStart return?

• How long will a writer wait?

• How to give priority to a waiting writer?

• Why use broadcast?

• Note that at least 1 thread is woken up each time any thread leaves. How to decrease the number of spurious wakeups?
Semaphores

- Semaphores are a generalized lock/unlock
- Definition
  - A non-negative integer (initialized to user-specified value)
  - down(): wait for semaphore value to become positive, then atomically decrement semaphore value by 1.

  - up(): increment semaphore value by 1.
  - key parts are atomic, so two down() calls can’t decrement value below 0
- Binary semaphore: value is 0 or 1
  - down waits for value to be 1, then sets it to 0
  - up sets value to 1

Semaphores can implement both mutual exclusion and ordering

- Mutual exclusion
  - Initial value is 1
down()
critical section
up()

- Ordering constraints
  - Usually, initial value is 0
  - Example: ensure that Thread A’s task is done before Thread B’s task

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task A</td>
<td>down()</td>
</tr>
<tr>
<td>up()</td>
<td>Task B</td>
</tr>
</tbody>
</table>
Implementing producer-consumer with semaphores

- Semaphore assignments
  - mutex: ensures mutual exclusion around code that manipulates coke machine. Initialized to 1.
  - fullSlots: counts the number of full slots in the coke machine. Initialized to 0.
  - emptySlots: counts of the number of empty slots in the coke machine. Initialized to MAX.

Consumer
take coke out of machine

Producer
add coke to machine
Implementing producer-consumer with semaphores

• Why do we need different semaphores for fullSlots and emptySlots?

• Does the order of down() matter?

• Does the order of up() matter?

• What needs to change to allow multiple producers (or multiple consumers)?

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

Comparing monitors and semaphores

• Semaphores provide 1 mechanism that can accomplish both mutual exclusion and ordering (monitors use different mechanism for each)
  – Elegant mechanism
  – Can be difficult to use
• Monitor lock = binary semaphore (initialized to 1)
  – lock() = down()
  – unlock() = up()
Condition variable versus semaphore

<table>
<thead>
<tr>
<th>Condition variable</th>
<th>Semaphore</th>
</tr>
</thead>
<tbody>
<tr>
<td>while(cond) {wait();}</td>
<td>down()</td>
</tr>
<tr>
<td>Conditional code in user program</td>
<td>Conditional code in semaphore definition</td>
</tr>
<tr>
<td>User writes customized condition. More flexible</td>
<td>Condition specified by semaphore definition (wait if value == 0)</td>
</tr>
<tr>
<td>User provides shared variable; protects with lock</td>
<td>Semaphore provides shared variable (integer) and thread-safe operations on that variable (down, up)</td>
</tr>
<tr>
<td>No memory of past signals</td>
<td>Remembers past up calls</td>
</tr>
</tbody>
</table>

Implementing custom waiting condition with semaphores

- Semaphores work best if the shared integer and waiting condition (value==0) map naturally to problem domain
- How to implement custom waiting condition with semaphores
  - Create one semaphore per waiting thread
  - Implement condition variable as a queue of semaphores
  - To wait, push that thread’s semaphore onto waiting queue
  - To signal, up one semaphore from the waiting queue and remove that thread from the queue.
Producer-consumer with semaphores (monitor style)

Consumer
mutex.down()
// wait for full slot
while(numCokes==0)
take coke out of machine
// signal producer
...
mutex.up()

Producer
mutex.down()
// wait for empty slot
...
add coke to machine
// signal consumer

take coke out of machine
// signal producer
...
mutex.up()

Implementing threads

- We’ve been assuming that each thread has a dedicated processor
- But what if there are more threads than processors?
Ready threads

• What to do with thread while it’s not running?
  – What is a non-running thread?
  – Must save its private state somewhere

• 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. Instead, use multiple stacks and just copy the stack pointer to the thread control block.

  – Keep track of ready threads (e.g., on a queue)
  – Thread state can now be running (on the CPU), ready (waiting for the CPU), or blocked (waiting for a synchronization operation from a thread).

Thread states

• 3 thread states
  – Running (currently using CPU)
  – Ready (waiting for CPU)
  – Blocked (waiting for a synchronization operation from another thread, e.g., unlock, signal, broadcast, up)
Switching threads

• Steps to switch to a different thread
  – Current thread returns control to OS
  – OS chooses new thread to run
  – OS saves state of current thread from CPU to its thread control block
  – OS loads context of next thread from its thread control block
  – OS runs next thread

Returning control to OS

• How does thread return control back to OS, so we can save the state of the current thread and run a new thread?
Choosing the next thread to run

• Many ways to choose which thread to run next (CPU scheduling)
  – FIFO
  – Priority

• What should CPU do if there are no other ready threads?

Saving state of current thread

• Save registers, PC, stack pointer
• Typically very tricky assembly-language code
  – E.g., why won’t the following code work?
    100   save PC (i.e., value 100)
    101   switch to next thread
• In Project 2, we’ll use Linux’s swapcontext()
Loading context of next thread and running it

• How to load registers?

• How to load stack?

• How to resume execution?

• Who is carrying out these steps?

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 (thread %d)”
Example of thread switching

Thread 1 output

Thread 2 output

Creating a new thread

• Create state for thread and it to ready queue
  – When pausing a running thread, we saved its state to its thread control block.
  – We can **construct** the state in the thread control block as if it had been running and got paused.

• Steps
  – Allocate and initialize new thread control block
  – Allocate and initialize new stack
  – In Project 2, this is done via makecontext()
  – Add thread control block to ready queue

• Unix fork() is related but different.
  – Unix fork() creates a new process (new thread + new address space)
How to use new thread

- Creating a thread is like an asynchronous procedure call

Synchronizing with child

- What if parent wants to work for a while, then wait for child to finish?
Synchronizing with child

- Does this work?

parent()

create child thread

print "parent works"

print "parent continues"

child()

print "child works"

Synchronizing with child

- Does this work?

parent()

create child thread

lock

print "parent works"

wait

print "parent continues"

unlock

child()

lock

print "child works"

signal

unlock
Synchronizing with child

- join(): wait for another thread to finish

```java
parent()
create child thread
lock
print "parent works"
unlock

join

lock
print "parent continues"
unlock
```

```java
child()
lock
print "child works"
unlock
```

Implementing locks -- atomicity in the thread library

- Concurrent programs use high-level synchronization operations

<table>
<thead>
<tr>
<th>concurrent programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>high-level synchronization operations provided by software (e.g., locks, monitors, semaphores)</td>
</tr>
<tr>
<td>low-level atomic operations provided by hardware (load/store, interrupt enable/disable, test&amp;set)</td>
</tr>
</tbody>
</table>

- How to implement these high-level synchronization operations?
  - Used by multiple threads, so must be thread safe
  - Can’t use high-level synchronization operations
Can use interrupt disable/enable to ensure atomicity (on uniprocessor)

- On uniprocessor, operation is atomic if context switch doesn’t occur
  - How does a thread get switched out?
  - Can prevent context switches during an operation by preventing these events
- User code could disable interrupts to ensure atomicity
  
  ```c
  disable interrupts
  if (no milk) {
    buy milk
  }
  enable interrupts
  ```
- Problems?

Lock implementation #1 (uniprocessor): disable interrupts, busy wait

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

unlock() {
    disable interrupts
    status = FREE
    enable interrupts
}
```
Another atomic primitive: read-modify-write instruction

- Interrupt disable works on uniprocessor by preventing the current thread from being switched out
- But this doesn’t work on a multiprocessor
  - Other processors are still running threads
  - Not acceptable to stop all other processors from executing
- Could use atomic load/store (Too much milk solution #3)
- Modern processors provide an easier way: atomic read-modify-write instructions
  - atomically \( \{ \text{read value from memory into register; write new value to memory} \} \)
Atomic read-modify-write instructions

• test_and_set: atomically write 1 to a memory location (set) and return the value that was there before (test)

```c
void test_and_set(int *X) {
    int tmp = *X;
    *X = 1;
    return(tmp);
}
```

• exchange
  – swap value between register and memory

Lock implementation #2 (test_and_set with busy waiting)

```c
// status=0 means lock is free
void lock() {
    while (test_and_set(&status) == 1) {
    }
}

void unlock() {
    status = 0
}
```

• If lock is free, test_and_set changes status to 1 and returns 0, so while loop finishes
• If lock is busy, test_and_set leaves status as 1 and returns 1, so while loop continues
• test_and_set is atomic, so only one thread will see transition from 0 to 1.
Busy waiting

• Problem with lock implementation #1 and #2
  – Waiting thread uses lots of CPU time just checking for lock to become free.
  – Better for thread to sleep and let other threads run
  – Solution: integrate lock implementation with thread dispatch; have lock implementation manipulate thread queues
    • Waiting thread gives up processor, so other threads can run. Someone wakes up thread when lock is free.

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

```c
lock() {
    disable interrupts
    if (status == FREE) {
        status = BUSY
    } else {
        add thread to queue of threads waiting for lock
        switch to next ready thread
    }
    enable interrupts
}
```
```c
unlock() {
    disable interrupts
    status = FREE
    if (any thread is waiting for this lock) {
        move waiting thread to ready queue
        status = BUSY
    }
    enable interrupts
}
```
• Handoff lock
  – Unlocker is giving the lock to the waiting locker
• What does it mean for lock() to add current thread to lock wait queue?
• Why have a separate waiting queue for the lock? Why not put waiting thread onto ready queue?

• Normally, when you lock (or disable interrupts), you unlock (or enable interrupts) to clean up. When should lock() re-enable interrupts before calling switch?

Interrupt enable/disable pattern

• Enable interrupts before adding thread to wait queue?

lock() {
    disable interrupts
    ...
    if (lock is busy) {
        enable interrupts
        add thread to queue of threads waiting for lock
        switch to next ready thread
    }
    enable interrupts
}

• When could this fail?
Interrupt enable/disable pattern

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

```c
lock() {
    disable interrupts
    ...
    if (lock is busy) {
        add thread to queue of threads waiting for lock
        enable interrupts
        switch to next ready thread
    }
    enable interrupts
}
```

- This fails if interrupt occurs before switch
  - lock() adds thread to wait queue
  - lock() enables interrupts
  - interrupt occurs, causing switch to another thread. Now thread is on two queues (ready and lock waiting)!

---

Interrupt enable/disable pattern

- So, adding thread to lock wait queue and switching must be **atomic**
- Thread leaves interrupts disabled when calling switch

- What can lock() assume about the state of interrupts after switch returns?

- How does lock() wake up from switch?

- Switch invariant
  - All threads promise to have interrupts disabled when calling switch
  - All threads assume interrupts are disabled when returning from switch
Lock implementation #4 (test_and_set, minimal busy waiting)

- Can’t implement locks using test_and_set without some busy-waiting, but can minimize it
- Use busy waiting only to atomically execute lock code. Give up CPU if busy.

```c
lock() {
    disable interrupts
    while (test_and_set(guard)) {}

    if (status == FREE) {
        status = BUSY
    } else {

        add thread to queue of threads waiting for lock

        switch to next ready thread
    }
    guard = 0
    enable interrupts
}
```
unlock() {
    disable interrupts
    while (test_and_set(guard)) {} 

    status = FREE 
    if (any thread is waiting for this lock) { 
        move waiting thread to ready queue 
        status = BUSY 
    }

    guard = 0 
    enable interrupts 
}

• What’s the switch invariant for multiprocessors?

How did we atomically \{add thread to waiting list and sleep\}?

• When there’s another thread to run 

• When there’s no other thread to run
Deadlock

- Resources
  - Something needed by a thread
  - A thread waits for resources
  - E.g., locks, disk space, memory, CPU
- Deadlock
  - A cyclical waiting for resources, which prevents the threads involved from making progress
- Example

  Thread A
  - x.lock
  - y.lock
  - ...
  - y.unlock
  - x.unlock

  Thread B
  - y.lock
  - x.lock
  - ...
  - x.unlock
  - y.unlock

Generic example of multi-threaded program

phase 1:
while (!done) {
    acquire some resource
    work
}

phase 2:
release all resources

Assume phase 1 has finite amount of work
Dining philosophers

- 5 philosophers sit at round table. 1 chopstick between each pair of philosophers (5 chopsticks total). Each philosopher needs 2 chopsticks to eat.

- Algorithm for 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?
Four necessary conditions for deadlock

- 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
- Cyclical chain of requests

Strategies for handling deadlock

- General strategies
  - Ignore
  - Detect and fix
  - Prevent
- Detect and fix
  - Detect cycles in the wait-for graph
  - How to fix once detected?
Deadlock prevention

• Eliminate one of the four necessary conditions
• Increase resources to decrease waiting

• Eliminate hold and wait
  – Move resource acquisition to beginning

  Phase 1a: acquire all resources
  Phase 1b: while (!done) {
    work
  }

  Phase 2: release all resources

• Wait until all resources you’ll need are free, then grab them all atomically

• If you encounter a busy resource, release all acquired resources and start over
  – Problems?

• Allow preemption
  – Preempt CPU by saving its state to thread control block and resuming later
  – Preempt memory by swapping memory to disk and loading it back later
  – Can you preempt a lock?
Eliminate cyclical chain of requests

Banker’s algorithm

- Similar to reserving all resources at beginning, but allows more concurrency
- State maximum resource needs in advance (but don’t actually acquire the resources). May block when thread tries to acquire the resource.
- General structure of thread code

Phase 1a: state maximum resource needed
Phase 1b: while (!done) {
    acquire some resource (blocking if not safe)
    work
}

Phase 2: release all resources
- Delays when resources are acquired (and when blocking may be needed), relative to acquiring all resources at beginning
- Phase 1a informs system, so it can tell when it’s safe to satisfy a resource acquisition in Phase 1b.
- “Safe” means deadlock is impossible (i.e., all threads are guaranteed to be able to finish).
Example of Banker’s algorithm

- Model a bank loaning money to its customers
  - Bank has $6000
  - Customers establish credit limit (i.e., maximum resources needed)
  - Customers borrow money in stages. When finished, they return all money.

- Solution 1: Bank promises to give money immediately upon request, up to customer’s credit limit.
  - Ann asks for credit limit of $2000
  - Bob asks for credit limit of $4000
  - Charlie asks for credit limit of $6000
  - Can bank approve all these credit limits?

- Solution 2: Banker’s algorithm
  - Bank approves all credit limits, but may block customer when he/she asks for the money.
    - Ann asks for credit limit of $2000 (bank ok’s)
    - Bob asks for credit limit of $4000 (bank ok’s)
    - Charlie asks for credit limit of $6000 (bank ok’s)
  - 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 only if, after giving the money, there exists way to fulfill all maximum resource requests
    - Give $2000 to Charlie (bank has $1000 left)
    - Give $1000 to Ann, then Ann finishes (bank has $2000)
    - Give $2000 to Bob, then Bob finishes (bank has $4000)
    - Give $4000 to Charlie, then Charlie finishes
Example of Banker’s algorithm

• Another scenario
  – Ann asks for credit limit of $2000 (bank ok’s)
  – Bob asks for credit limit of $4000 (bank ok’s)
  – Charlie asks for credit limit of $6000 (bank ok’s)

  – 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’s algorithm allows system to overcommit resources without deadlock
  – Sum of max resource needs can be greater than total resources, as long as threads are guaranteed to be able to finish

• How to apply Banker’s algorithm to dining philosophers?

• But difficult to anticipate maximum resources needed

CPU scheduling

• How to choose next thread to run? What are the goals of a CPU scheduler?

• Minimize average response time
  – 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 preemptions
  - Thread runs until it calls yield() or blocks
  - No timer interrupts
- Pros and cons
  + simple
  - short jobs can get stuck behind long jobs
  - not interactive
- 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
- 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; 1 second time slice)
  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
Round robin

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

Pros and cons
  + good for interactive computing
  - more context-switching overhead

How to choose time slice?
  - Big time slice: degenerates to FCFS
  - Small time slice: more overhead due to context switching
  - Typically a compromise, e.g., 10 ms (if each context switch takes 0.1 ms, then this leads to 1% overhead)

STCF (shortest time to completion first)

- Run whichever job has least amount of work to do before finishing or blocking
  - Preempt current job if shorter job arrives
- Finish short jobs first
  - Improves response time of short jobs (by a lot)
  - Hurts response time of long jobs (by a little)
- STCF gives optimal response time
- Example (Job A takes 100 seconds; job B takes 1 second; 1 second time slice)
  Time 0: Job A arrives and starts
  Time 0+: Job B arrives and preempts job A
  Time 1: Job B ends (response time = 1)
  Time 101: Job A ends (response time = 101)
  Average response time = 51
STCF

• Pros and cons
  + optimal average response time
  - long jobs can get stuck \textit{forever} behind short jobs (starvation)
  - needs knowledge of future

• How to predict the future?

Example

• Job A: compute for 1000 seconds
• Job B: compute for 1000 seconds
• Job C

\begin{verbatim}
while (1) {
  compute for 1 ms
  Disk I/O for 10 ms
}
\end{verbatim}

• A and B can each use 100\% of CPU; C can use 91\% of disk I/O. What happens when we run them together?
• Goal: keep both CPU and disk busy

• FCFS
  – If A or B run before C, disk will be idle for 1000-2000 seconds
• Round robin with 100 ms time slice
CA---------B---------CA---------B---------
| - |                   | - |
C’s       C’s       I/O       I/O

   – Disk is idle most of the time

• Round robin with 1 ms time slice
CABABABABABCABABABABABC-
|---------|---------|
C’s I/O   C’s I/O

   – Disk is utilized about 90% of the time, but lots of context switches

• STCF
CA---------C
|--------|
C’s I/O

Real-time scheduling

• So far, we’ve focused on **average** response time
• But sometimes, the goal is to meet deadlines (irrelevant how far ahead of time the job completes)
   – Video or audio output
   – Control of physical systems
• This requires **worst-case** analysis
• How do we schedule for deadlines in life?
Earliest-deadline first (EDF)

- Always run job with the earliest deadline
- Preempt current job if a new job arrives with earlier deadline
- Optimal: will meet all deadlines if it’s possible to do so
- Example
  - Job A: Takes 15 seconds; deadline is 20 seconds after arrival
  - Job B: Takes 10 seconds; deadline is 30 seconds after arrival
  - Job C: Takes 5 seconds; deadline is 10 seconds after arrival

0  5  10  15  20  25  30  35  40  45  50  55  60  65  70  75  80  85
A  +
B  +
C  +