Recall: Use of Threads

- Version of program with Threads (loose syntax):

```c
main() {
    ThreadFork(ComputePI, "pi.txt");
    ThreadFork(PrintClassList, "classlist.txt");
}
```

- What does ThreadFork() do?
  - Start independent thread running given procedure
- What is the behavior here?
  - Now, you would actually see the class list
  - This should behave as if there are two separate CPUs

```
CPU1  CPU2  CPU1  CPU2  CPU1  CPU2
Time
```

Recall: the Dispatch Loop

- Conceptually, the scheduling loop of the operating system looks as follows:

```c
Loop {
    RunThread();  // Needs to return to loop every now and then!
    ChooseNextThread();
    SaveStateOfCPU(curTCB);
    LoadStateOfCPU(newTCB);
}
```

- This is an infinite loop
  - One could argue that this is all that the OS does
- Should we ever exit this loop???
  - When would that be?

Running a thread

- Consider first portion: RunThread()
  - How do I run a thread?
    - Load its state (registers, PC, stack pointer) into CPU
    - Load environment (virtual memory space, etc)
    - Jump to the PC
  - Note: We give control of processor/core to user code!!
    - OS is not running because user code is running

- How does the OS get control back?
  - Internal events: thread returns control voluntarily
  - External events: thread gets preempted
Internal Events

- Blocking on I/O
  - The act of requesting I/O implicitly yields the CPU
- Waiting on a "signal" from other thread
  - Thread asks to wait and thus yields the CPU
- Thread executes a yield()
  - Thread volunteers to give up CPU

```
computePI() {
    while(TRUE) {
        ComputeNextDigit();
        yield();
    }
}
```

Stack for Yielding Thread

- How do we run a new thread?
  ```
  run_new_thread() {
      newThread = PickNewThread();
      switch(curThread, newThread);ThreadHouseKeeping(); /* Do any cleanup */
  }
  ```

- How does dispatcher switch to a new thread?
  - Save anything next thread may trash: PC, regs, stack pointer
  - Maintain isolation for each thread

Stacks for Yield with Multiple Threads

- Consider the following code blocks:
  ```
  proc A() {
    B();
  }
  proc B() {
    while(TRUE) {
        yield();
    }
  }
  ```

- Suppose we have 2 threads:
  - Threads S and T
  - Assume that both have been running for a while

Saving/Restoring state (often called "Context Switch")

```
Switch(tCur,tNew) {
    /* Unload old thread */
    TCB[tCur].regs.r7 = CPU.r7;
    ...
    TCB[tCur].regs.r0 = CPU.r0;
    TCB[tCur].regs.sp = CPU.sp;
    TCB[tCur].regs.retpc = CPU.retpc; /*return addr*/
    /* Load and execute new thread */
    CPU.r7 = TCB[tNew].regs.r7;
    ...
    CPU.r0 = TCB[tNew].regs.r0;
    CPU.sp = TCB[tNew].regs.sp;
    CPU.retpc = TCB[tNew].regs.retpc;
    return; /* Return to CPU.retpc */
}
```
Switch Details (continued)

- What if you make a mistake in implementing switch?
  - Suppose you forget to save/restore register 32
  - Get intermittent failures depending on when context switch occurred and whether new thread uses register 32
  - System will give wrong result without warning
- Can you devise an exhaustive test to test switch code?
  - No! Too many combinations and inter-leavings
- Cautionary tale:
  - For speed, Topaz kernel saved one instruction in switch()
  - Carefully documented! Only works as long as kernel size < 1MB
  - What happened?
    » Time passed, People forgot
    » Later, they added features to kernel (no one removes features!)
  - Very weird behavior started happening
- Moral of story: Design for simplicity

How expensive is context switching?

- Switching between threads in same process similar to switching between threads in different processes, but much cheaper:
  - No need to change address space
- Some numbers from Linux:
  - Frequency of context switch: 10-100ms
  - Switching between processes: 3-4 μsec.
  - Switching between threads: 100 ns
- Even cheaper: switch threads (using "yield") in user-space!

What happens when thread blocks on I/O?

- What happens when a thread requests a block of data from the file system?
  - User code invokes a system call
  - Read operation is initiated
  - Run new thread/switch
- Thread communication similar
  - Wait for Signal/Join
  - Networking

External Events

- What happens if thread never does any I/O, never waits, and never yields control?
  - Could the ComputePI program grab all resources and never release the processor?
    » What if it didn’t print to console?
  - Must find way that dispatcher can regain control!
- Answer: utilize external events
  - Interrupts: signals from hardware or software that stop the running code and jump to kernel
  - Timer: like an alarm clock that goes off every some milliseconds
- If we make sure that external events occur frequently enough, can ensure dispatcher runs
Recall: Interrupt Controller

- Interrupts invoked with interrupt lines from devices
- Interrupt controller chooses interrupt request to honor
  - Interrupt identity specified with ID line
  - Mask enables/disables interrupts
  - Priority encoder picks highest enabled interrupt
- Software Interrupt Set/Cleared by Software
- CPU can disable all interrupts with internal flag
- Non-Maskable Interrupt line (NMI) can’t be disabled

Example: Network Interrupt

<table>
<thead>
<tr>
<th>IntID</th>
<th>Interrupt</th>
<th>Control</th>
<th>Software Interrupt</th>
<th>NMI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU</td>
<td>Priority Encoder</td>
<td>Timer</td>
<td></td>
</tr>
</tbody>
</table>

Network

Use of Timer Interrupt to Return Control

- Solution to our dispatcher problem
  - Use the timer interrupt to force scheduling decisions

ThreadFork(): Create a New Thread

- ThreadFork() is a user-level procedure that creates a new thread and places it on ready queue

- Arguments to ThreadFork()
  - Pointer to application routine (fcnPtr)
  - Pointer to array of arguments (fcnArgPtr)
  - Size of stack to allocate

- Implementation
  - Sanity check arguments
  - Enter Kernel-mode and Sanity Check arguments again
  - Allocate new Stack and TCB
  - Initialize TCB and place on ready list (Runnable)
How do we initialize TCB and Stack?

- Initialize Register fields of TCB
  - Stack pointer made to point at stack
  - PC return address \( \Rightarrow \) OS (asm) routine ThreadRoot()
  - Two arg registers \( (a0 \text{ and } a1) \) initialized to \( \text{fcnPtr} \) and \( \text{fcnArgPtr} \), respectively

- Initialize stack data?
  - Minimal initialization \( \Rightarrow \) setup return to go to beginning of ThreadRoot()
    * Important part of stack frame is in registers for RISC-V (ra)
    * X86: need to push a return address on stack
  - Think of stack frame as just before body of ThreadRoot() really gets started

How does a thread get started?

- Eventually, \( \text{run\_new\_thread()} \) will select this TCB and return into beginning of ThreadRoot()
  - This really starts the new thread

What does ThreadRoot() look like?

- ThreadRoot() is the root for the thread routine:
  ```c
  SetupNewThread(tNew) {
    TCB[tNew].regs.sp = newStackPtr;
    TCB[tNew].regs.retpc = &ThreadRoot;
    TCB[tNew].regs.r0 = fcnPtr
    TCB[tNew].regs.r1 = fcnArgPtr
  }
  ```
- Startup Housekeeping
  - Includes things like recording start time of thread
  - Other statistics
- Stack will grow and shrink with execution of thread
- Final return from thread returns into ThreadRoot() which calls ThreadFinish()
  - ThreadFinish() wake up sleeping threads
Threads vs Address Spaces: Options

<table>
<thead>
<tr>
<th># threads Per AS:</th>
<th>One</th>
<th>Many</th>
</tr>
</thead>
<tbody>
<tr>
<td># of addr spaces:</td>
<td>MS/DOS, early Macintosh</td>
<td>Traditional UNIX</td>
</tr>
<tr>
<td>One</td>
<td>Embedded systems (Geoworks, VxWorks, JavaOS etc)</td>
<td>Mach, OS/2, Linux Windows 10</td>
</tr>
<tr>
<td>Many</td>
<td>Win NT to XP, Solaris, HP-UX, OS X</td>
<td></td>
</tr>
</tbody>
</table>

- Most operating systems have either
  - One or many address spaces
  - One or many threads per address space

Administirivia

- Midterm Thursday 2/15
  - Closed book, but one page of handwritten notes, both sides
  - No class on day of midterm
  - 8-10PM
- Project 1 Design Document due next Saturday 2/10
  - No extensions of any sort on design documents!!
- Project 1 Design reviews upcoming
  - High-level discussion of your approach
    » What will you modify?
    » What algorithm will you use?
    » How will things be linked together, etc.
  - Do not need final design (complete with all semicolons!)
  - You will be asked about testing
    » Understand testing framework
    » Are there things you are doing that are not tested by tests we give you?
  - Do your own work!
    - Please do not try to find solutions from previous terms
    - We will be on the look out for anyone doing this…today

Goals for Rest of Today

- Challenges and Pitfalls of Concurrency
- Synchronization Operations/Critical Sections
- How to build a lock?
- Atomic Operations

Concurrency vs Parallelism

- Multithreading: Multiple threads per Process *(A programming strategy)*
- Multiplexing: Sharing a single resource (such as a core) among multiple threads
- What does it mean to run two threads “concurrently” *(regardless of process)*?
  - Scheduler is free to run threads in any order and interleaving: FIFO, Random, …
  - Unless synchronization is involved, multiple threads are concurrent!
  - Assume: if scheduler can produce the worst possible interleaving, IT WILL!
- What does it mean to run two threads “in parallel” *(regardless of process)*?
  - Threads are actually running at the same time
  - Parallel ⇒ Concurrent but Concurrent ⇒ Parallel
ATM Bank Server

- ATM server problem:
  - Service a set of requests
  - Do so without corrupting database
  - Don’t hand out too much money

ATM bank server example

- Suppose we wanted to implement a server process to handle requests from an ATM network:

  ```
  BankServer() {
    while (TRUE) {
      ReceiveRequest(&op, &acctId, &amount);
      ProcessRequest(op, acctId, amount);
    }
  }
  ProcessRequest(op, acctId, amount) {
    if (op == deposit) Deposit(acctId, amount);
    else if …
  }
  Deposit(acctId, amount) {
    acct = GetAccount(acctId); /* may use disk I/O */
    acct->balance += amount;
    StoreAccount(acct); /* Involves disk I/O */
  }
  
  How could we speed this up?
  - More than one request being processed at once
  - Event driven (overlap computation and I/O)
  - Multiple threads (multi-processor, or overlap comp and I/O)
  
Can Threads (in same Process) Make This Easier?

- Threads yield overlapped I/O and computation without “deconstructing” code into non-blocking fragments
  - One thread per request
- Requests proceeds to completion, blocking as required:
  ```
  Deposit(acctId, amount) {
    acct = GetAccount(acctId); /* May use disk I/O */
    acct->balance += amount;
    StoreAccount(acct); /* Involves disk I/O */
  }
  ```

  How could we speed this up?
  - More than one request being processed at once
  - Event driven (overlap computation and I/O)
  - Multiple threads (multi-processor, or overlap comp and I/O)

- Example:
  ```
  BankServer() {
    while(TRUE) {
      event = WaitForNextEvent();
      if (event == ATMRequest)
        StartOnRequest();
      else if (event == AcctAvail)
        ContinueRequest();
      else if (event == AcctStored)
        FinishRequest();
    }
  }
  ```

  – This technique is used for graphical programming

  - This technique is used for graphical programming

- Complication:
  - What if we missed a blocking I/O step?
  - What if we have to split code into hundreds of pieces which could be blocking?
Problem is at the Lowest Level

• Most of the time, threads are working on separate data, so scheduling doesn’t matter:
  
  Thread A                      Thread B
  \[ x = 1; \quad y = 2; \]

• However, what about (Initially, \( y = 12 \)):
  
  Thread A                      Thread B
  \[ x = 1; \quad y = 2; \]
  \[ x = y+1; \quad y = y^2; \]

  – What are the possible values of \( x \)?
  – Or, what are the possible values of \( x \) below?

  Thread A                      Thread B
  \[ x = 1; \quad x = 2; \]

  – \( X \) could be 1 or 2 (non-deterministic!)
  – Could even be 3 for serial processors:
    » Thread A writes 0001, B writes 0010 \( \rightarrow \) scheduling order ABABABBA yields 3!

Atomic Operations

• To understand a concurrent program, we need to know what the underlying indivisible operations are!

• Atomic Operation: an operation that always runs to completion or not at all
  – It is *indivisible*: it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  – Fundamental building block – if no atomic operations, then have no way for threads to work together

• On most machines, memory references and assignments (i.e. loads and stores) of words are atomic

  – Consequently – weird example that produces “3” on previous slide can’t happen

• Many instructions are not atomic
  – Double-precision floating point store often not atomic
  – VAX and IBM 360 had an instruction to copy a whole array

Another Concurrent Program Example

• Two threads, A and B, compete with each other

  – One tries to increment a shared counter
  – The other tries to decrement the counter

  Thread A                      Thread B
  \[ i = 0; \quad i = 0; \]
  \[ \text{while} \ (i < 10) \quad \text{while} \ (i > -10) \]
  \[ i += 1; \quad i -= 1; \]
  \[ \text{printf} (“A wins!”); \quad \text{printf} (“B wins!”); \]

• Assume that memory loads and stores are atomic, but incrementing and decrementing are *not* atomic

• Who wins? Could be either
• Is it guaranteed that someone wins? Why or why not?

Hand Simulation Multiprocessor Example

• Inner loop looks like this:

  Thread A                      Thread B
  \[ r1=0 \quad \text{load} \ r1, M[i] \quad r1=0 \quad \text{load} \ r1, M[i] \]
  \[ r1=1 \quad \text{add} \ r1, r1, 1 \quad r1=1 \quad \text{sub} \ r1, r1, 1 \]
  \[ M[i]=1 \quad \text{store} \ r1, M[i] \quad M[i]=-1 \quad \text{store} \ r1, M[i] \]

• Hand Simulation:

  – And we’re off. A gets off to an early start
  – B says “hmph, better go fast” and tries really hard
  – A goes ahead and writes “1”
  – B goes and writes “-1”
  – A says “HUH??! I could have sworn I put a 1 there”

  • Could this happen on a uniprocessor? With Hyperthreads?
  – Yes! Unlikely, but if you are depending on it not happening, it will and your system will break…
Definitions

- **Synchronization**: using atomic operations to ensure cooperation between threads
  - For now, only loads and stores are atomic
  - We are going to show that it's hard to build anything useful with only reads and writes

- **Mutual Exclusion**: ensuring that only one thread does a particular thing at a time
  - One thread excludes the other while doing its task

- **Critical Section**: piece of code that only one thread can execute at once. Only one thread at a time will get into this section of code
  - Critical section is the result of mutual exclusion
  - Critical section and mutual exclusion are two ways of describing the same thing

Locks

- **Lock**: prevents someone from doing something
  - Lock() before entering critical section and before accessing shared data
  - Unlock() when leaving, after accessing shared data
  - Wait if locked
    » Important idea: all synchronization involves waiting

- Locks need to be allocated and initialized:
  - structure Lock mylock or pthread_mutex_t mylock;
  - lock_init(&mylock) or mylock = PTHREAD_MUTEX_INITIALIZER;

- Locks provide two **atomic** operations:
  - acquire(&mylock) — wait until lock is free; then mark it as busy
  - After this returns, we say the calling thread holds the lock
  - release(&mylock) — mark lock as free
  - Should only be called by a thread that currently holds the lock
  - After this returns, the calling thread no longer holds the lock

Fix banking problem with Locks!

- Identify critical sections (atomic instruction sequences) and add locking:
  Display code:
  ```
  Deposit(acctId, amount) {
    acquire(&mylock)  // Wait if someone else in critical section!
    acct = GetAccount(acctId);
    acct->balance += amount;
    StoreAccount(acct);
    release(&mylock)  // Release someone into critical section
  }
  ```

- Must use SAME lock (mylock) with all of the methods (Withdraw, etc…)
  - Shared with all threads!

Correctness Requirements

- Threaded programs must work for all interleavings of thread instruction sequences
  - Cooperating threads inherently non-deterministic and non-reproducible
  - Really hard to debug unless carefully designed!

- Example: Therac-25
  - Machine for radiation therapy
    » Software control of electron accelerator and electron beam/X-ray production
    » Software control of dosage
  - Software errors caused the death of several patients
    » A series of race conditions on shared variables and poor software design
    » "They determined that data entry speed during editing was the key factor in producing the error condition: If the prescription data was edited at a fast pace, the overdose occurred."
Conclusion

- Every thread has both a user and kernel stack
  - Showed more details about context-switching mechanisms
- Concurrent threads introduce problems when accessing shared data
  - Programs must be insensitive to arbitrary interleavings
  - Without careful design, shared variables can become completely inconsistent
- Important concept: Atomic Operations
  - An operation that runs to completion or not at all
  - These are the primitives on which to construct various synchronization primitives
- Introduced the Lock API: acquire() and release()
  - Next time: How do we make a lock?