### CS362 Operating Systems Cooperating Threads

**Review: Per Thread State**
- Each Thread has a *Thread Control Block (TCB)*
  - Execution State: CPU registers, program counter, pointer to stack
  - Scheduling info: State (more later), priority, CPU time
  - Accounting Info
  - Various Pointers (for implementing scheduling queues)
  - Pointer to enclosing process? (PCB)?
  - Etc (add stuff as you find a need)
- OS Keeps track of TCBs in protected memory
  - In Arrays, or Linked Lists, or ...

**Review: Yielding through 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();
    }
  }
  ```
  - Note that yield() must be called by programmer frequently enough!

**Review: Stack for Yielding Thread**
- How do we run a new thread?
  ```
  run_new_thread() {
    newThread = PickNewThread();
    switch(curThread, newThread);
    ThreadHouseKeeping(); /* Later in lecture */
  }
  ```
- How does dispatcher switch to a new thread?
  - Save anything next thread may trash: PC, regs, stack
  - Maintain isolation for each thread
Review: Two Thread Yield Example

- Consider the following code blocks:
  ```c
  proc A() {
    B();
  }
  proc B() {
    while(TRUE) {
      yield();
    }
  }
  ```
- Suppose we have 2 threads:
  - Threads S and T

![Diagram showing stack growth for threads S and T]

Interrupt Controller

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

Review: Preemptive Multithreading

- Use the timer interrupt to force scheduling decisions

![Diagram showing interrupts and scheduling]

- Timer Interrupt routine:
  ```c
  TimerInterrupt() {
    DoPeriodicHouseKeeping();
    run_new_thread();
  }
  ```
- This is often called preemptive multithreading, since threads are preempted for better scheduling
  - Solves problem of user who doesn’t insert yield();

Review: Lifecycle of a Thread (or Process)

- As a thread executes, it changes state:
  - new: The thread is being created
  - ready: The thread is waiting to run
  - running: Instructions are being executed
  - waiting: Thread waiting for some event to occur
  - terminated: The thread has finished execution
- "Active" threads are represented by their TCBs
  - TCBs organized into queues based on their state
Every thread (and/or Process) has a parentage
- A “parent” is a thread that creates another thread
- A child of a parent was created by that parent

Kernel versus User-Mode threads
- We have been talking about Kernel threads
  - Native threads supported directly by the kernel
  - Every thread can run or block independently
  - One process may have several threads waiting on different things
- Downside of kernel threads: a bit expensive
  - Need to make a crossing into kernel mode to schedule
- Even lighter weight option: User Threads
  - User program provides scheduler and thread package
  - May have several user threads per kernel thread
  - User threads may be scheduled non-preemptively relative to each other (only switch on yield())
  - Cheap
- Downside of user threads:
  - When one thread blocks on I/O, all threads block
  - Kernel cannot adjust scheduling among all threads

ThreadJoin() system call
- One thread can wait for another to finish with the ThreadJoin(tid) call
  - Calling thread will be taken off run queue and placed on waiting queue for thread tid
- Where is a logical place to store this wait queue?
  - On queue inside the TCB
- Similar to wait() system call in UNIX
  - Lets parents wait for child processes

Threading models mentioned by book
Simple One-to-One Threading Model

Many-to-One

Many-to-Many
Multiprocessing vs Multiprogramming

- **Remember Definitions:**
  - Multiprocessing ≡ Multiple CPUs
  - Multiprogramming ≡ Multiple Jobs or Processes
  - Multithreading ≡ Multiple threads per Process

- **What does it mean to run two threads “concurrently” ?**
  - Scheduler is free to run threads in any order and interleaving: FIFO, Random, ...
  - Dispatcher can choose to run each thread to completion or time-slice in big chunks or small chunks

Correctness for systems with concurrent threads

- **If dispatcher can schedule threads in any way, programs must work under all circumstances**
  - Can you test for this?
  - How can you know if your program works?

- **Independent Threads:**
  - No state shared with other threads
  - Deterministic ⇒ Input state determines results
  - Reproducible ⇒ Can recreate Starting Conditions, I/O
  - Scheduling order doesn’t matter

- **Cooperating Threads:**
  - Shared State between multiple threads
  - Non-deterministic
  - Non-reproducible
  - Non-deterministic and Non-reproducible means that bugs can be intermittent
  - Sometimes called “Heisenbugs”

Interactions Complicate Debugging

- **Is any program truly independent?**
  - Every process shares the file system, OS resources, network, etc
  - Extreme example: buggy device driver causes thread A to crash “independent thread” B

- **You probably don’t realize how much you depend on reproducibility:**
  - Example: Evil C compiler
    - Modifies files behind your back by inserting errors into C program unless you insert debugging code
  - Example: Debugging statements can overrun stack

- **Non-deterministic errors are really difficult to find**
  - Example: Memory layout of kernel+user programs
    - Depends on scheduling, which depends on timer/other things
  - Example: Something which does interesting I/O
    - User typing of letters used to help generate secure keys

Why allow cooperating threads?

- **People cooperate; computers help/enhance people’s lives, so computers must cooperate**
  - By analogy, the non-reproducibility/non-determinism of people is a notable problem for “carefully laid plans”

- **Advantage 1: Share resources**
  - One computer, many users
  - One bank balance, many ATMs
    - What if ATMs got update at night?
    - Embedded systems (robot control)
    - Need to coordinate arm&hand

- **Advantage 2: Speedup**
  - Overlap I/O and computation
    - Many different file systems do read-ahead
  - Multiprocessors – chop up program into parallel pieces

- **Advantage 3: Modularity**
  - More important than you might think
  - Chop large problem up into simpler pieces
    - To compile, for instance, gcc calls cpp | cc1 | cc2 | as | ld
    - Makes system easier to extend
High-level Example: Web Server

• Server must handle many requests
• Non-cooperating version:
  serverLoop()
  con = AcceptCon();
  ProcessFork(ServiceWebPage(),con);

• What are some disadvantages of this technique?

Threaded Web Server

• Now, use a single process
• Multithreaded (cooperating) version:
  serverLoop()
  connection = AcceptCon();
  ThreadFork(ServiceWebPage(),connection);

• Looks almost the same, but has many advantages:
  - Can share file caches kept in memory, results of CGI
    scripts, other things
  - Threads are much cheaper to create than processes, so
    this has a lower per-request overhead
• Question: would a user-level (say one-to-many)
  thread package make sense here?
  - When one request blocks on disk, all block...
• What about DOS attacks or slash-dot effects?

Thread Pools

• Problem with previous version: Unbounded Threads
  - When web-site becomes too popular - throughput sinks
• Instead, allocate a bounded “pool” of threads,
  representing the maximum level of multiprogramming

master()
  allocThreads(slave,queue);
  while(TRUE)
  con=AcceptCon();
  Enqueue(queue,con);
  wakeUp(queue);

slave(queue)
  while(TRUE)
  con=Dequeue(queue);
  if (con=null)
  sleepOn(queue);
  else
  ServiceWebPage(con);
