Recall: Dispatch Loop

• Conceptually, the dispatching loop of the operating system looks as follows:

```c
Loop {
    RunThread();
    newTCB = 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:

```
RunThread()
```

```
LoadStateOfCPU(newTCB)
```

• 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

• How does the dispatcher 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

```java
computePI() {
    while(TRUE) {
        ComputeNextDigit();
        yield();
    }
}
```
- How do we run a new thread?
  ```c
  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
What Do the Stacks Look Like?

• Consider the following code blocks:

```plaintext
proc A() {
    B();
}
proc B() {
    while(TRUE) {
        yield();
    }
}
```

• Suppose we have 2 threads running same code:
  – Threads S and T

![Diagram showing stack growth and thread switching](image)
Saving/Restoring state (often called “Context Switch”)

```c
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
Some Numbers

• Frequency of performing context switches: 10-100ms
• Context switch time in Linux: 3-4 µsecs (Intel i7 & E5)
  – Thread switching faster than process switching (100 ns)
  – But switching across cores ~2x more expensive than within-core

• Context switch time increases sharply with size of working set*
  – Can increase 100x or more

*The working set is subset of memory used by process in a time window

• Moral: context switching depends mostly on cache limits and the process or thread’s hunger for memory
Some Numbers

- Many processes are multi-threaded, so thread context switches may be either **within-process** or **across-processes**
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
Example: Network Interrupt

- An interrupt is a hardware-invoked context switch
  - No separate step to choose what to run next
  - Always run the interrupt handler immediately
Use of Timer Interrupt to Return Control

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

```c
TimerInterrupt() {
    DoPeriodicHouseKeeping();
    run_new_thread();
}
```
Thread Abstraction

- Illusion: Infinite number of processors

Programmer Abstraction

<table>
<thead>
<tr>
<th>Threads</th>
<th>Processor Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processors</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Thread Abstraction

- Illusion: Infinite number of processors
- Reality: Threads execute with variable speed
  - Programs must be designed to work with any schedule
## Programmer vs. Processor View

<table>
<thead>
<tr>
<th>Programmer’s View</th>
<th>Possible Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Execution #1</strong></td>
</tr>
<tr>
<td></td>
<td>x = x + 1;</td>
</tr>
<tr>
<td></td>
<td>y = y + x;</td>
</tr>
<tr>
<td></td>
<td>z = x + 5y;</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Execution #2</strong></td>
</tr>
<tr>
<td></td>
<td>x = x + 1;</td>
</tr>
<tr>
<td></td>
<td>thread is suspended</td>
</tr>
<tr>
<td></td>
<td>other thread(s) run</td>
</tr>
<tr>
<td></td>
<td>thread is resumed</td>
</tr>
<tr>
<td></td>
<td>y = y + x</td>
</tr>
<tr>
<td></td>
<td>z = x + 5y</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Execution #3</strong></td>
</tr>
<tr>
<td></td>
<td>x = x + 1;</td>
</tr>
<tr>
<td></td>
<td>thread is suspended</td>
</tr>
<tr>
<td></td>
<td>other thread(s) run</td>
</tr>
<tr>
<td></td>
<td>thread is resumed</td>
</tr>
<tr>
<td></td>
<td>z = x + 5y</td>
</tr>
</tbody>
</table>
### Programmer vs. Processor View

<table>
<thead>
<tr>
<th>Programmer’s View</th>
<th>Possible Execution #1</th>
<th>Possible Execution #2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[x = x + 1;]</td>
<td>[x = x + 1;]</td>
</tr>
<tr>
<td></td>
<td>[y = y + x;]</td>
<td>[y = y + x;]</td>
</tr>
<tr>
<td></td>
<td>[z = x + 5y;]</td>
<td>[z = x + 5y;]</td>
</tr>
<tr>
<td></td>
<td>[\ldots]</td>
<td>[\ldots]</td>
</tr>
<tr>
<td></td>
<td>[y = y + x]</td>
<td>[\ldots]</td>
</tr>
<tr>
<td></td>
<td>[z = x + 5y]</td>
<td>[\ldots]</td>
</tr>
</tbody>
</table>

Possible Execution #2:
- Thread is suspended
- Other thread(s) run
- Thread is resumed

Possible Execution #3:
- \[y = y + x\]
- \[z = x + 5y\]
## Programmer vs. Processor View

<table>
<thead>
<tr>
<th>Programmer’s View</th>
<th>Possible Execution #1</th>
<th>Possible Execution #2</th>
<th>Possible Execution #3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = x + 1; y = y + x; z = x + 5y;</td>
<td>x = x + 1; y = y + x; z = x + 5y;</td>
<td>x = x + 1</td>
<td>x = x + 1</td>
</tr>
<tr>
<td>thread is suspended</td>
<td>other thread(s) run</td>
<td>thread is resumed</td>
<td>other thread(s) run</td>
</tr>
<tr>
<td>thread is resumed</td>
<td>thread is resumed</td>
<td>y = y + x</td>
<td>y = y + x</td>
</tr>
<tr>
<td>z = x + 5y</td>
<td>z = x + 5y</td>
<td>}</td>
<td>z = x + 5y</td>
</tr>
</tbody>
</table>
### Possible Executions

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **a)** One execution
- **b)** Another execution

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>❏</td>
<td></td>
<td></td>
</tr>
<tr>
<td>❏</td>
<td>❏</td>
<td></td>
</tr>
<tr>
<td>❏</td>
<td>❏</td>
<td>❏</td>
</tr>
</tbody>
</table>

- **c)** Another execution
Thread Lifecycle

- **Init**: Thread Creation
  - e.g., sthread_create()

- **Ready**: Scheduler Resumes Thread
  - e.g., sthread_join()
  - Event Occurs
    - e.g., other thread calls sthread_join()

- **Running**: Thread Yields/Scheduler Suspends Thread
  - e.g., sthread_yield()

- **Waiting**: Thread Waits for Event
  - e.g., sthread_join()

- **Finished**: Thread Exit
  - e.g., sthread_exit()
Administrivia

- Your section is your home for CS162
  - The TA needs to get to know you to judge participation
  - All design reviews will be conducted by your TA
  - You can attend alternate section by same TA, but try to keep the amount of such cross-section movement to a minimum

- First midterm: Monday, October 1, 5:00-6:30pm
BREAK
Per Thread Descriptor
(Kernel Supported Threads)

• Each Thread has a *Thread Control Block (TCB)*
  – Execution State: CPU registers, program counter (PC), pointer to stack (SP)
  – Scheduling info: state, priority, CPU time
  – Various Pointers (for implementing scheduling queues)
  – Pointer to enclosing process (PCB) – user threads
  – … (add stuff as you find a need)

• OS Keeps track of TCBs in “kernel memory”
  – In Array, or Linked List, or …
  – I/O state (file descriptors, network connections, etc)
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 ⇒ OS (asm) routine `ThreadRoot()`
  – Two arg registers (a0 and a1) initialized to `fcnPtr` and `fcnArgPtr`, respectively

• Initialize stack data?
  – No. Important part of stack frame is in registers (ra)
  – Think of stack frame as just before body of `ThreadRoot()` really gets started

```
ThreadRoot stub
```

Stack growth

Initial Stack
How does Thread get started?

- Eventually, `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
  ThreadRoot() {
    DoStartupHousekeeping();
    UserModeSwitch(); /* enter user mode */
    Call fcnPtr(fcnArgPtr);
    ThreadFinish();
  }
  ```

- **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
Multithreaded Processes

- Process Control Block (PCBs) points to multiple Thread Control Blocks (TCBs):

  - Switching threads within a block is a simple thread switch
  - Switching threads across blocks requires changes to memory and I/O address tables
Examples multithreaded programs

• Embedded systems
  – Elevators, planes, medical systems, smart watches
  – Single program, concurrent operations

• Most modern OS kernels
  – Internally concurrent because have to deal with concurrent requests by multiple users
  – But no protection needed within kernel

• Database servers
  – Access to shared data by many concurrent users
  – Also background utility processing must be done
Example multithreaded programs (con’t)

• Network servers
  – Concurrent requests from network
  – Again, single program, multiple concurrent operations
  – File server, Web server, and airline reservation systems

• Parallel programming (more than one physical CPU)
  – Split program into multiple threads for parallelism
  – This is called Multiprocessing

• Some multiprocessors are actually uniprogrammed:
  – Multiple threads in one address space but one program at a time
A Typical Use Case

Client Browser
- process for each tab
- thread to render page
- GET in separate thread
- multiple outstanding GETs
- as they complete, render portion

Web Server
- fork process for each client connection
- thread to get request and issue response
- fork threads to read data, access DB, etc
- join and respond
Kernel Use Cases

• Thread for each user process

• Thread for sequence of steps in processing I/O

• Threads for device drivers

• …
Putting it Together: Process

(Unix) Process

A(int tmp) {
    if (tmp<2)
        B();
        printf(tmp);
    }
B() {
    C();
}
C() {
    A(2);
}
A(1);
...

Sequential stream of instructions

Resources

Memory

Stack

I/O State
(e.g., file, socket contexts)

CPU state
(PC, SP, registers..)

Stored in OS

A()
Putting it Together: Processes

- Switch overhead: **high**
  - CPU state: **low**
  - Memory/IO state: **high**
- Process creation: **high**
- Protection
  - CPU: **yes**
  - Memory/IO: **yes**
- Sharing overhead: **high**
  (involves at least a context switch)
Putting it Together: Threads

- Switch overhead: medium
  - CPU state: low
- Thread creation: medium
- Protection
  - CPU: yes
  - Memory/IO: no
- Sharing overhead: *low(ish)*
  (thread switch overhead low)
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

- Lighter weight option: User level Threads
User-Mode Threads

• Lighter weight option:
  – 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
  – Option: Scheduler Activations
    » Have kernel inform user level when thread blocks…
Some Threading Models

Simple One-to-One Threading Model

Many-to-One

Many-to-Many
Threads in a Process

- Threads are useful at user-level: parallelism, hide I/O latency, interactivity
- Option A (early Java): user-level library, one multi-threaded process
  - Library does thread context switch
  - Kernel time slices between processes, e.g., on system call I/O
- Option B (SunOS, Linux/Unix variants): many single-threaded processes
  - User-level library does thread multiplexing
- Option C (Windows): scheduler activations
  - Kernel allocates processes to user-level library
  - Thread library implements context switch
  - System call I/O that blocks triggers upcall
- Option D (Linux, MacOS, Windows): use kernel threads
  - System calls for thread fork, join, exit (and lock, unlock, …)
  - Kernel does context switching
  - Simple, but a lot of transitions between user and kernel mode
Putting it Together: Multi-Cores

- Switch overhead: low (only CPU state)
- Thread creation: low
- Protection
  - CPU: yes
  - Memory/IO: No
- Sharing overhead: low (thread switch overhead low, may not need to switch at all!)
Simultaneous MultiThreading/Hyperthreading

- Hardware technique
  - Superscalar processors can execute multiple instructions that are independent
  - Hyperthreading duplicates register state to make a second “thread,” allowing more instructions to run
- Can schedule each thread as if were separate CPU
  - But, sub-linear speedup!
- Original called “Simultaneous Multithreading”
  - Intel, SPARC, Power (IBM)
  - A virtual core on AWS’ EC2 is basically a hyperthread
Putting it Together: Hyper-Threading

- Switch overhead between hardware-threads: 
  very-low (done in hardware)

- Contention for ALUs/FPUs may hurt performance
**Classification**

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

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

• Processes have two parts
  – Threads (Concurrency)
  – Address Spaces (Protection)

• Various textbooks talk about *processes*
  – When this concerns concurrency, really talking about thread portion of a process
  – When this concerns protection, talking about address space portion of a process

• Concurrent threads are a very useful abstraction
  – Allow transparent overlapping of computation and I/O
  – Allow use of parallel processing when available

• Concurrent threads introduce problems when accessing shared data
  – Programs must be insensitive to arbitrary interleavings
  – Without careful design, shared variables can become completely inconsistent