Recall: Lifecycle of a Process

- As a process executes, it changes state:
  - new: The process is being created
  - ready: The process is waiting to run
  - running: Instructions are being executed
  - waiting: Process waiting for some event to occur
  - terminated: The process has finished execution

Recall: Use of Threads

- Version of program with Threads (loose syntax):
  ```
  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

Recall: Multithreaded Stack Switching

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

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

Example of using a timer interrupt to return control

- 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

Example of Timer Interrupt routine:

```}
TimerInterrupt() {
    DoPeriodicHouseKeeping();
    run_new_thread();
}
```
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 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

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:
  
  ```
  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
Administrivia

- Waitlist was closed last Friday/Early Drop passed Friday
- Recommendation: Read assigned readings before lecture
- Group sign up this week
  - Get finding groups ASAP – deadline Friday 2/8 at 11:59PM
  - 4 people in a group!
- Continue to attend whichever section is convenient
  - Next week, we start official section attendance!
- TA preference signup form due Tuesday 2/12 at 11:59PM
  - Everyone in a group must have the same TA!
    » Preference given to same section
    » Participation: Get to know your TA!

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

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

**Possible Executions**

- **Thread 1**
- **Thread 2**
- **Thread 3**

a) One execution

- **Thread 1**
- **Thread 2**
- **Thread 3**

c) Another execution

**Thread Lifecycle**

- **Init**
- **Ready**
- **Running**
- **Waiting**
- **Finished**

- Thread Creation: e.g., `sthread_create()`
- Ready:
- Scheduler Resumes Thread
  - Thread Yields/Suspends Thread: e.g., `sthread_yield()`, `sthread_join()`
- Waiting:
- Finished:
- Thread Exit: e.g., `sthread_exit()`
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)

Multithreaded Processes

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

Example 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

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

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**

1. Sequential stream of instructions
2. CPU state (PC, SP, registers..)
3. I/O State (e.g., file, socket contexts)
4. Memory
5. Stack

**Resources**

**Stored in OS**

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

### Putting it Together: Processes

**Process 1**

- Mem.
- IO state
- CPU state

**Process 2**

- Mem.
- IO state
- CPU state

... **Process N**

- Mem.
- IO state
- CPU state

**CPU state**

- 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)

**CPU sched.**

1 process at a time

### Putting it Together: Threads

**Process 1**

- Mem.
- IO state
- CPU state

**Process N**

- Mem.
- IO state
- CPU state

**threads**

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

**CPU sched.**

1 thread at a time

### 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/I/O: 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</th>
<th>One</th>
<th>Many</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>MS/DOS, early Macintosh</td>
<td>Traditional UNIX</td>
<td></td>
</tr>
<tr>
<td>Many</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>
<td></td>
</tr>
</tbody>
</table>

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

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 (if \texttt{switch()} works!!!)
- 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
    - Original UNIX had a bunch of non-deterministic errors
  - 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 were only updated at night?
  - Embedded systems (robot control: 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 \texttt{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 Denial of Service attacks or digg / 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 worker threads, representing the maximum level of multiprogramming

```python
master() {
    allocThreads(worker, queue);
    while (TRUE) {
        con = AcceptCon();
        Enqueue(queue, con);
        wakeUp(queue);
    }
}
```

```python
worker(queue) {
    while (TRUE) {
        con = Dequeue(queue);
        if (con == null)
            sleepOn(queue);
        else
            ServiceWebPage(con);
    }
}
```

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:

```python
BankServer() {
    while (TRUE) {
        ReceiveRequest(&op, &acctId, &amount);
        ProcessRequest(op, acctId, amount);
    }
}
```

```python
ProcessRequest(op, acctId, amount) {
    if (op == deposit) Deposit(acctId, amount);
    else if ...
}
```

```python
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-proc, or overlap comp and I/O)
Event Driven Version of ATM server

- Suppose we only had one CPU
  - Still like to overlap I/O with computation
  - Without threads, we would have to rewrite in event-driven style
- Example

```c
BankServer() {
    while (TRUE) {
        event = WaitForNextEvent();
        if (event == ATMRequest)
            StartOnRequest();
        else if (event == AcctAvail)
            ContinueRequest();
        else if (event == AcctStored)
            FinishRequest();
    }
}
```
- What if we missed a blocking I/O step?
- What if we have to split code into hundreds of pieces which could be blocking?
- This technique is used for graphical programming

Can Threads 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:

```c
Deposit(acctId, amount) {
    acct = GetAccount(acctId); /* May use disk I/O */
    acct->balance += amount;
    StoreAccount(acct); /* Involves disk I/O */
}
```
- Unfortunately, shared state can get corrupted:

```
Thread 1
load r1, acct->balance
add r1, amount1
store r1, acct->balance

Thread 2
load r1, acct->balance
add r1, amount2
store r1, acct->balance
```

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