Recall: Namespaces for communication over IP

- **Hostname**
  - www.eecs.berkeley.edu
- **IP address**
  - 128.32.244.172 (IPv4 32-bit)
  - fe80::4ad7:5ff:efc:2607 (IPv6 128-bit)
- **Port Number**
  - 0-1023 are "well known" or "system" ports
    - Superuser privileges to bind to one
  - 1024 – 49151 are "registered" ports (registry)
    - Assigned by IANA for specific services
  - 49152–65535 (2^{15}+2^{14} to 2^{16}−1) are “dynamic” or “private”
    - Automatically allocated as “ephemeral Ports”

Recall: Use of Sockets in TCP

- **Socket**: an abstraction of a network I/O queue
  - Embodies one side of a communication channel
    - Same interface regardless of location of other end
    - Local machine (“UNIX socket”) or remote machine (“network socket”)
  - First introduced in 4.2 BSD UNIX; big innovation at time
    - Now most operating systems provide some notion of socket

- **Using Sockets for Client-Server (C/C++ interface):**
  - **On server**: set up “server-socket”
    - Create socket, Bind to protocol (TCP), local address, port
    - Call listen(): tells server socket to accept incoming requests
    - Perform multiple accept() calls on socket to accept incoming connection request
    - Each successful accept() returns a new socket for a new connection; can pass this off to handler thread
  - **On client:**
    - Create socket, Bind to protocol (TCP), remote address, port
    - Perform connect() on socket to make connection
    - If connect() successful, have socket connected to server

Recall: Socket Setup over TCP/IP

- **Server Socket**: Listens for new connections
  - Produces new sockets for each unique connection

- **Things to remember:**
  - Connection involves 5 values: [Client Addr, Client Port, Server Addr, Server Port, Protocol]
  - Often, Client Port “randomly” assigned
    - Done by OS during client socket setup
  - Server Port often “well known”
    - 80 (web), 443 (secure web), 25 (sendmail), etc
  - Well-known ports from 0—1023
Example: Server Protection and Parallelism

Client
- Create Client Socket
- Connect it to server (host:port)
- write request
- read response
- Close Client Socket

Server
- Create Server Socket
- Bind it to an Address (host:port)
- Listen for Connection
- Accept connection
- write response
- Close Connection Socket
- Close Server Socket

Recall: Server Protocol (v3)

```c
listen(lstnsockfd, MAXQUEUE);
while (1) {
    consockfd = accept(lstnsockfd, (struct sockaddr *)&cli_addr,
                      &clilen);
    cpid = fork();              /* new process for connection */
    if (cpid > 0) {             /* parent process */
        close(consockfd);
    } else if (cpid == 0) {     /* child process */
        close(lstnsockfd);       /* let go of listen socket */
        server(consockfd);
        close(consockfd);
        exit(EXIT_SUCCESS);      /* exit child normally */
    }
}
close(lstnsockfd);
```

Server Address - Itself

```c
struct sockaddr_in {
    short sin_family;
    unsigned short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
} serv_addr;
memset((char *)&serv_addr, 0, sizeof(serv_addr));
```

- Simple form
- Internet Protocol
- accepting any connections on the specified port
- In “network byte ordering”

Client: Getting the Server Address

```c
struct hostent *buildServerAddr(struct sockaddr_in *serv_addr,
                                 char *hostname, int portno) {
    struct hostent *server;
    /* Get host entry associated with a hostname or IP address */
    server = gethostbyname(hostname);
    if (server == NULL) {
        fprintf(stderr,"ERROR, no such host\n");
        exit(1);
    }
    memset((char *)&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr,
          (char *)&(serv_addr.sin_addr.s_addr), server->h_length);
    serv_addr.sin_port = htons(portno);
    return server;
}
```
Recall: Traditional UNIX Process

- Process: OS abstraction of what is needed to run a single program
  - Often called a “HeavyWeight Process”
  - No concurrency in a “HeavyWeight Process”

- Two parts:
  - Sequential program execution stream
    [ACTIVE PART]
    - Code executed as a sequential stream of execution (i.e., thread)
    - Includes State of CPU registers
  - Protected resources
    [PASSIVE PART]:
    - Main memory state (contents of Address Space)
    - I/O state (i.e. file descriptors)

How do we Multiplex Processes?

- The current state of process held in a process control block (PCB):
  - This is a “snapshot” of the execution and protection environment
  - Only one PCB active at a time
- Give out CPU time to different processes (Scheduling):
  - Only one process “running” at a time
  - Give more time to important processes
- Give pieces of resources to different processes (Protection):
  - Controlled access to non-CPU resources
  - Example mechanisms:
    - Memory Mapping: Give each process their own address space
    - Kernel/User duality: Arbitrary multiplexing of I/O through system calls

CPU Switch From Process A to Process B

- This is also called a “context switch”
- Code executed in kernel above is overhead
  - Overhead sets minimum practical switching time
  - Less overhead with SMT/hyperthreading, but... contention for resources instead

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
Process Scheduling

- PCBs move from queue to queue as they change state
  - Decisions about which order to remove from queues are Scheduling decisions
  - Many algorithms possible (few weeks from now)

Ready Queue And Various I/O Device Queues

- Process not running ⇒ PCB is in some scheduler queue
  - Separate queue for each device/signal/condition
  - Each queue can have a different scheduler policy

Modern Process with Threads

- Thread: a sequential execution stream within process
  (Sometimes called a “Lightweight process”)
  - Process still contains a single Address Space
  - No protection between threads

- Multithreading: a single program made up of a number of different concurrent activities
  - Sometimes called multitasking, as in Ada …

- Why separate the concept of a thread from that of a process?
  - Discuss the “thread” part of a process (concurrency)
  - Separate from the “address space” (protection)
  - Heavyweight Process = Process with one thread
Administrivia

- Group signups: 4 members/group
  - Groups need to be finalized today!
  - 36 students without groups
  - We may add a student to groups of 3
  - Sign up with the autograder

- TA Signup form
  - Form asks for 3 section options (ranked) for your group
  - Please sign up by Monday!
  - We will try to accommodate your needs, but may not be able to fill over-popular sections – give us options!

- Need to get to know your TAs
  - Consider moving out of really full sections!

Thread State

- State shared by all threads in process/address space
  - Content of memory (global variables, heap)
  - I/O state (file descriptors, network connections, etc)

- State “private” to each thread
  - Kept in TCB = Thread Control Block
  - CPU registers (including, program counter)
  - Execution stack – what is this?

- Execution Stack
  - Parameters, temporary variables
  - Return PCs are kept while called procedures are executing

Shared vs. Per-Thread State

<table>
<thead>
<tr>
<th>Shared State</th>
<th>Per-Thread State</th>
<th>Per-Thread State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
<td>Thread Control Block (TCB)</td>
<td>Stack Information</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Saved Registers</td>
</tr>
<tr>
<td>Global Variables</td>
<td>Thread Metadata</td>
<td>Stack Information</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Saved Registers</td>
</tr>
<tr>
<td>Code</td>
<td>Stack</td>
<td>Stack Metadata</td>
</tr>
</tbody>
</table>
Execution Stack Example

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

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

MIPS: Software conventions for Registers

- Before calling procedure:
  - Save caller-saves regs
  - Save v0, v1
  - Save ra
- After return, assume
  - Callee-saves reg OK
  - gp,sp,fp OK (restored!)
  - Other things trashed

Motivational Example for Threads

- Imagine the following C program:

  main() {
    ComputePI("pi.txt");
    PrintClassList("classList.txt");
  }

- What is the behavior here?
  - Program would never print out class list
  - Why? ComputePI would never finish

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
Memory Footprint: Two-Threads

- If we stopped this program and examined it with a debugger, we would see:
  - Two sets of CPU registers
  - Two sets of Stacks

- Questions:
  - How do we position stacks relative to each other?
  - What maximum size should we choose for the stacks?
  - What happens if threads violate this?
  - How might you catch violations?

Actual Thread Operations

- `thread_fork(func, args)`
  - Create a new thread to run `func(args)`
  - Pintos: `thread_create`
- `thread_yield()`
  - Relinquish processor voluntarily
  - Pintos: `thread_yield`
- `thread_join(thread)`
  - In parent, wait for forked thread to exit, then return
  - Pintos: `thread_join`
- `thread_exit`
  - Quit thread and clean up, wake up joiner if any
  - Pintos: `thread_exit`

- pThreads: POSIX standard for thread programming
  [POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995)]

Dispatch Loop

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

  ```c
  Loop {
    RunThread();
    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

- 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

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

Stack for Yielding Thread

• 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
  – Maintain isolation for each thread

What Do the Stacks Look Like?

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

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

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 \( \mu \)secs (Intel i7 & E5)
  - Thread switching faster than process switching (100 ns)
  - But switching across cores \( \sim 2 \times \) 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 many milliseconds
- If we make sure that external events occur frequently enough, can ensure dispatcher runs

Use of Timer Interrupt to Return Control

- Solution to our dispatcher problem
  - Use the timer interrupt to force scheduling decisions
- Timer Interrupt routine:
  ```c
  TimerInterrupt()
  {
    // Code
    run_new_thread();
  }
  ```

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
Thread Abstraction

Programmer Abstraction
Threads | 1 | 2 | 3 | 4 | 5 |
|-------|---|---|---|---|---|
Processors | 1 | 2 | 3 | 4 | 5 |

- Illusion: Infinite number of processors

Physical Reality
Threads | 1 | 2 | 3 | 4 | 5 |
|-------|---|---|---|---|---|
Processors | 1 | 2 | 3 | 4 | 5 |

Running Threads
Ready Threads

- Illusion: Infinite number of processors
- Reality: Threads execute with variable speed
  - Programs must be designed to work with any schedule

Programmer vs. Processor View

Programmer’s View
Possible Execution

#1
.
.
.
x = x + 1; x = x + 1;
y = y + x; y = y + x;
z = x + 5y; z = x + 5y;
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
**Programmer vs. Processor View**

**Programmer's View**

```
x = x + 1;
y = y + x;
z = x + 5y;
```

**Possible Executions**

**Possible Execution #1**

```
x = x + 1;
y = y + x;
z = x + 5y;
```

**Possible Execution #2**

```
x = x + 1
..............
thread is suspended
other thread(s) run
thread is resumed
..............
y = y + x
z = x + 5y
```

**Possible Execution #3**

```
x = x + 1
y = y + x
..............
thread is suspended
other thread(s) run
thread is resumed
................
z = x + 5y
```

---

**Thread Lifecycle**

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

**Summary**

- Processes have two parts
  - One or more Threads (Concurrency)
  - Address Spaces (Protection)

- Concurrency accomplished by multiplexing CPU Time:
  - Unloading current thread (PC, registers)
  - Loading new thread (PC, registers)
  - Such context switching may be voluntary \((yield())\), I/O operations or involuntary (timer, other interrupts)

- Guest lecturers next week