Introduction to Networking (Finished), Concurrency (Processes and Threads)

September 14th, 2015
Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

Acknowledgments: Lecture slides are from the Operating Systems course taught by John Kubiatowicz at Berkeley, with few minor updates/changes. When slides are obtained from other sources, a reference will be noted on the bottom of that slide, in which case a full list of references is provided on the last slide.
Recall: Namespaces for communication over IP

- **Hostname**
  - www.eecs.berkeley.edu

- **IP address**
  - 128.32.244.172 (ipv4 format)

- **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
    » Could be local machine (called “UNIX socket”) or remote machine (called “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, ServerAddr, 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
- Close Listen Socket
- Read request
- **write response**
- Close Connection Socket
- Close Server Socket

(child)
Recall: Server Protocol (v3)

while (1) {
    listen(lstnsockfd, MAXQUEUE);
    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
memset((char *) &serv_addr,0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(portno);
```

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

    /* Construct an address for remote server */
    memset((char *) serv_addr, 0, sizeof(struct sockaddr_in));
    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;
}
BIG OS Concepts so far

- Processes
- Address Space
- Protection
- Dual Mode
- Interrupt handlers (including syscall and trap)
- File System
  - Integrates processes, users, cwd, protection
- Key Layers: OS Lib, Syscall, Subsystem, Driver
  - User handler on OS descriptors
- Process control
  - fork, wait, signal, exec
- Communication through sockets
- Client-Server Protocol
Recall: Traditional UNIX Process

- Process: Operating system abstraction to represent 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
    » Code executed as a sequential stream of execution (i.e., thread)
    » Includes State of CPU registers
  - Protected resources:
    » 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
CPU Switch From Process to Process

- This is also called a “context switch”
- Code executed in kernel above is overhead
  - Overhead sets minimum practical switching time
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

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

```
Ready Queue
  Head
  Tail

USB Unit 0
  Head
  Tail

Disk Unit 0
  Head
  Tail

Disk Unit 2
  Head
  Tail

Ether Netwk 0
  Head
  Tail
```

```
Link
Registers
Other State TCB₀

Link
Registers
Other State TCB₆

Link
Registers
Other State TCB₁₆
```

```
Link
Registers
Other State TCB₂

Link
Registers
Other State TCB₃
```

```
Link
Registers
Other State TCB₈
```
Administrivia

- **Group signups: 4 members/group**
  - Groups need to be finished by next Wednesday!

- **Finding info on your own is a good idea!**
  - Learn your tools, like “man”
  - Can even type “man xxx” into google!
    » Example: “man ls”
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

• 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
Single and Multithreaded Processes

- Threads encapsulate concurrency: “Active” component
- Address spaces encapsulate protection: “Passive” part
  - Keeps buggy program from trashing the system
- Why have multiple threads per address space?
Thread State

- State shared by all threads in process/addr 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
Execution Stack Example

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

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages
Motivational Example for Threads

• Imagine the following C program:

```c
main() {
    ComputePI("pi.txt");
    PrintClassList("clist.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):

```c
main() {
    ThreadFork(ComputePI("pi.txt"));
    ThreadFork(PrintClassList("clist.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
CPU2   CPU1   CPU2   CPU1   CPU2   CPU1
Time   
```
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

- `thread_exit`
  - Quit thread and clean up, wake up joiner if any
  - Pintos: `thread_exit`

- pThreads: POSIX standard for thread programming
Dispatch Loop

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

  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

```cpp
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
  - Maintain isolation for each thread
What do the stacks look like?

- Consider the following code blocks:

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

- Suppose we have 2 threads:
  - Threads S and T
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 4
  - Get intermittent failures depending on when context switch occurred and whether new thread uses register 4
  - 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 tail:
  - 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 (Current Intel i7 & E5).
  - Thread switching faster than process switching (100 ns).
  - But switching across cores about 2x more expensive than within-core switching.
- Context switch time increases sharply with the size of the working set*, and can increase 100x or more.

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

- Moral: Context switching depends mostly on cache limits and the process or thread's hunger for memory.
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?
  - 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
Thread Abstraction

- Infinite number of processors
- 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><strong>x = x + 1</strong></td>
<td><strong>x = x + 1</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>y = y + x</strong></td>
<td><strong>y = y + x</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>z = x + 5y</strong></td>
<td><strong>z = x + 5y</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>thread is suspended</strong></td>
<td><strong>thread is suspended</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>other thread(s) run</strong></td>
<td><strong>other thread(s) run</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>thread is resumed</strong></td>
<td><strong>thread is resumed</strong></td>
</tr>
</tbody>
</table>

- **x = x + 1; y = y + x; z = x + 5y;**
- **y = y + x; z = x + 5y; thread is suspended**
- **other thread(s) run; thread is resumed;**
- **y = y + x; z = x + 5y;**
Possible Executions

Thread 1  
Thread 2  
Thread 3  

a) One execution

Thread 1  
Thread 2  
Thread 3  

b) Another execution

c) Another execution
Thread Lifecycle

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

- **Ready**
  - Scheduler Resumes Thread
  - Thread Yields/Scheduler Suspends Thread
    - e.g., `sthread_yield()`
  - Event Occurs
    - e.g., other thread calls `sthread_join()`

- **Running**
  - Thread Exit
    - e.g., `sthread_exit()`

- **Waiting**

- **Finished**
# 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></td>
<td>Thread Control Block (TCB)</td>
<td>Thread Control Block (TCB)</td>
</tr>
<tr>
<td>Heap</td>
<td>Stack Information</td>
<td>Stack Information</td>
</tr>
<tr>
<td>Global Variables</td>
<td>Saved Registers</td>
<td>Saved Registers</td>
</tr>
<tr>
<td>Code</td>
<td>Thread Metadata</td>
<td>Thread Metadata</td>
</tr>
<tr>
<td></td>
<td>Stack</td>
<td>Stack</td>
</tr>
</tbody>
</table>
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
  - Etc (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

- PCB points to multiple TCBs:

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

• Processes have two parts
  – 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)

• Protection accomplished restricting access:
  – Memory mapping isolates processes from each other
  – Dual-mode for isolating I/O, other resources

• 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