CMSC 321: Operating Systems

Lecture 4

Processes & Threads;
Nachos Appetizer
Before Threads...

• Recall that a process consists of:
  – program(s)
  – data
  – stack
  – PCB
  all stored in the process image

• Process (context) switch is pure overhead
Process Characterization

Process has two characteristics:

1. resource ownership
   - address space to hold process image
   - I/O devices, files, etc.

2. execution
   - a single execution path (thread of control)
   - execution state, PC & registers, stack
Refining Terminology

• Distinguish the two characteristics
  – process: resource ownership
  – thread: unit of execution (dispatching)
    • AKA lightweight process (LWP)

• Multi-threading: support multiple threads of execution within a single process

• Process, as we have known it thus far, is a single-threaded process
Single- vs. Multi-threaded Model

(a) Single-threaded Model

(b) Multi-threaded Model
Multi-Threaded Environment

• **Process:**
  - virtual address space (for image)
  - protected access to resources
    • processors, other processes, I/O, files

• **Thread:** one or more w/in a process
  - execution state
  - saved context when not running (i.e., independent PC)
  - stack
  - access to memory & resources of the process
## Multi-Threaded Environment

<table>
<thead>
<tr>
<th>Per process items</th>
<th>Per thread items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child processes</td>
<td>State</td>
</tr>
<tr>
<td>Pending alarms</td>
<td></td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting information</td>
<td></td>
</tr>
</tbody>
</table>

- **Left**: shared by all threads in a process
- **Right**: private to each thread
Single- vs. Multi-threaded Model

- still a single PCB & addr space per process
- separate stacks, TCB for each thread
Single- vs. Multi-threaded Model
Remember...

- Different threads in a process have the same address space
- Every thread can access every memory address within the address space
  - No protection between threads
- Each thread has its own stack
  - One frame per procedure called but not completed (local vars, return address)
Why Threads?

- In many apps, multiple activities @ once
  - e.g., word processor

- Easier to create and destroy than processes
  - no resources attached to threads

- Allow program to continue if part is blocked
  - permit I/O- and CPU-bound activities to overlap
  - speeds up application

- Easy resource sharing (same addr space!)

- Take advantage of multiprocessors
Some Examples

- word processor
- Web server
- handle interrupts
Thread Functionality

- Scheduling done on a per-thread basis
- Terminate process --> kill all threads

- Four basic thread operations:
  - spawn (automatically spawned for new process)
  - block
  - unblock
  - terminate

- Synchronization:
  - all threads share same addr space & resources
  - must synchronize to avoid conflicts
  - process synchro techniques are same for threads (later)
Two Types Of Threads

User-Level

Kernel-Level
User-Level Threads

• Thread management done by an application

• Use thread library (e.g., POSIX Pthreads)
  – create/destroy, pass msgs, schedule execution, save/restore contexts

• Each process needs its own thread table

• Kernel is unaware of these threads
  – assigns single execution state to the process
  – unaware of any thread scheduling activity
User-Level Threads

• Advantages:
  – thread switch does not require kernel privileges
  – thread switch more efficient than kernel call
  – scheduling can be process (app) specific
    • without disturbing OS
  – can run on any OS
  – scales easily

• Disadvantages:
  – if one thread blocks, all are blocked (process switch)
    • e.g., I/O, page faults
  – cannot take advantage of multiprocessor
    • one process to one processor
  – programmers usually want threads for blocking apps
Kernel-Level Threads

• Thread management done by kernel
  – process as a whole (process table)
  – individual threads (thread table)

• Kernel schedules on a per-thread basis

• Addresses disadvantages of ULT:
  – schedule multi threads from one process on multiple CPUs
  – if one thread blocks, schedule another (no process switch)

• Disadvantage of KLT:
  – thread switch causes mode switch to kernel
What is (not are) Nachos?

- **Not Another Completely Heuristic Operating System**
- Non-trivial instructional software
- *You* modify, develop, and execute actual OS code!

- Supports:
  - threads
  - user-level processes (you will write your own!)
  - virtual memory
  - file systems
  - etc.
Your First Nachos Project

• Build a Thread System

• A partial system is given to you:
  – thread fork
  – thread completion
  – semaphores for synchronization (huh?)
    • for concurrency issues (to come)...

• Your job:
  – first, read and understand the code as given
  – second, implement additional synchro routines
  – finally, implement specific problems that use your routines
At This Point, You Should...

- Download `nachos1.tgz` to your directory
  - To extract: `tar -xzvf nachos1.tgz`
  - See `man tar`

- Resulting `nachos1/` directory will contain:
  - `Makefile`, `Makefile.common`, `Makefile.dep`
  - `threads/` directory
  - `machine/` directory

- Print out all the `.cc` and `.h` files
  - highlight, make notes, etc.
At This Point, You Should...

- Read, read, read!

- Read the “Roadmap” sections
  - “Nachos Machine”
  - “Nachos Threads”

- If you want to trace execution, find main()

- A great searching utility: grep
  - grep main threads/*.cc
  - grep main machine/*.cc

- Quiz next Thursday on this reading
  - specifically focus on “Nachos Threads”
Want to run Nachos?

- Execute `make` in `nachos1/` directory

- The executable `nachos` resides in `threads/`
  - `cd threads/`
  - `./nachos`

- Then, watch the magic!
  - actually, not very exciting at this point

- Trace through your hardcopies of the code