CS 153
Design of Operating Systems
Winter 2016
Lecture 6: Threads
Recap: Process Components

- A process is named using its process ID (PID)
- A process contains all of the state for a program in execution

<table>
<thead>
<tr>
<th>Per-Process State</th>
</tr>
</thead>
<tbody>
<tr>
<td>✮ An address space</td>
</tr>
<tr>
<td>✮ The code for the executing program</td>
</tr>
<tr>
<td>✮ The data for the executing program</td>
</tr>
<tr>
<td>✮ A set of operating system resources</td>
</tr>
<tr>
<td>» Open files, network connections, etc.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Per-Thread State</th>
</tr>
</thead>
<tbody>
<tr>
<td>✮ An execution stack encapsulating the state of procedure calls</td>
</tr>
<tr>
<td>✮ The program counter (PC) indicating the next instruction</td>
</tr>
<tr>
<td>✮ A set of general-purpose registers with current values</td>
</tr>
<tr>
<td>✮ Current execution state (Ready/Running/Waiting)</td>
</tr>
</tbody>
</table>
Threads in a Process

- Stack (T1)
- Stack (T2)
- Stack (T3)
- Heap
- Static Data
- Code

Thread 1
Thread 2
Thread 3
PC (T1)
PC (T2)
PC (T3)
Process/Thread Separation

- Separating threads and processes makes it easier to support multithreaded applications
  - Concurrency does not require creating new processes
- **Concurrency** (multithreading) can be very useful
  - Improving program structure
  - Handling concurrent events (e.g., Web requests)
  - Writing parallel programs
- So multithreading is even useful on a uniprocessor
Threads: Concurrent Servers

- Using fork() to create new processes to handle requests in parallel is overkill for such a simple task.
- Recall our forking Web server:

```c
while (1) {
    int sock = accept();
    if ((child_pid = fork()) == 0) {
        Handle client request
        Close socket and exit
    } else {
        Close socket
    }
}
```
Threads: Concurrent Servers

- Instead, we can create a new thread for each request

```c
web_server() {
    while (1) {
        int sock = accept();
        thread_fork(handle_request, sock);
    }
}

handle_request(int sock) {
    Process request
    close(sock);
}
```
A Word Process w/3 Threads

Display thread

Backup thread

Input thread

Kernel

Keyboard

Disk
Kernel-Level Threads

- We have taken the execution aspect of a process and separated it out into threads
  - To make concurrency cheaper
- As such, the OS now manages threads and processes
  - All thread operations are implemented in the kernel
  - The OS schedules all of the threads in the system
- OS-managed threads are called kernel-level threads or lightweight processes
  - Windows: threads
  - Solaris: lightweight processes (LWP)
  - POSIX Threads (pthreads): PTHREAD_SCOPE_SYSTEM
Kernel Thread Limitations

- Kernel-level threads make concurrency much cheaper than processes
  - Much less state to allocate and initialize

- However, for fine-grained concurrency, kernel-level threads still suffer from too much overhead
  - Thread operations still require system calls
    - Ideally, want thread operations to be as fast as a procedure call
  - Kernel-level threads have to be general to support the needs of all programmers, languages, runtimes, etc.

- For such fine-grained concurrency, need even “cheaper” threads
User-Level Threads

- To make threads cheap and fast, they need to be implemented at user level
  - Kernel-level threads are managed by the OS
  - User-level threads are managed entirely by the run-time system (user-level library)

- User-level threads are small and fast
  - A thread is simply represented by a PC, registers, stack, and small thread control block (TCB)
  - Creating a new thread, switching between threads, and synchronizing threads are done via procedure call
    » No kernel involvement
  - User-level thread operations 100x faster than kernel threads
  - pthreads: PTHREAD_SCOPE_PROCESS
User and Kernel Threads

Multiplexing user-level threads on a single kernel thread for each process

Multiplexing user-level threads on multiple kernel threads for each process
**U/L Thread Limitations**

- But, user-level threads are not a perfect solution
  - As with everything else, they are a tradeoff
- User-level threads are *invisible* to the OS
  - They are not well integrated with the OS
- As a result, the OS can make poor decisions
  - Scheduling a process with idle threads
  - Blocking a process whose thread initiated an I/O, even though the process has other threads that can execute
  - Unscheduling a process with a thread holding a lock
- Solving this requires communication between the kernel and the user-level thread manager
Kernel vs. User Threads

- Kernel-level threads
  - Integrated with OS (informed scheduling)
  - Slow to create, manipulate, synchronize

- User-level threads
  - Fast to create, manipulate, synchronize
  - Not integrated with OS (uninformed scheduling)

- Understanding the differences between kernel and user-level threads is important
  - For programming (correctness, performance)
  - For test-taking
Implementing Threads

Implementing threads has a number of issues
- Interface
- Context switch
- Preemptive vs. non-preemptive
- Scheduling
- Synchronization (next lecture)

Focus on kernel-level threads
- What you will be dealing with in Pintos
- Not only will you be using threads in Pintos, you will be implementing more thread functionality (e.g., sleep)
Sample Thread Interface

- `thread_fork(procedure_t)`
  - Create a new thread of control
  - Also `thread_create()`, `thread_setstate()`

- `thread_stop()`
  - Stop the calling thread; also `thread_block`

- `thread_start(thread_t)`
  - Start the given thread

- `thread_yield()`
  - Voluntarily give up the processor

- `thread_exit()`
  - Terminate the calling thread; also `thread_destroy`

- Where are they called? User-space or kernel-space?
Thread Scheduling

- The thread scheduler determines when a thread runs
- It uses queues to keep track of what threads are doing
  - Just like the OS and processes
  - Implemented at user-level in a library for user-level threads
- Run queue: Threads currently running (usually one)
- Ready queue: Threads ready to run
- Are there wait queues?
  - How would you implement thread_sleep(time)?
Non-Preemptive Scheduling

- Threads voluntarily give up the CPU with `thread_yield`

Ping Thread
```c
while (1) {
    printf("ping\n");
    thread_yield();
}
```

Pong Thread
```c
while (1) {
    printf("pong\n");
    thread_yield();
}
```

- What is the output of running these two threads?
The semantics of thread_yield are that it gives up the CPU to another thread.

- In other words, it \textit{context switches} to another thread.

So what does it mean for thread_yield to return?

- Execution trace of ping/pong:
  - \texttt{printf(“ping\n”);}
  - \texttt{thread_yield();}
  - \texttt{printf(“pong\n”);}
  - \texttt{thread_yield();}
  - …
Implementing `thread_yield()` (PintOS hint)

```c
thread_yield() {
    thread_t old_thread = current_thread;
    current_thread = get_next_thread();
    append_to_queue(ready_queue, old_thread);
    context_switch(old_thread, current_thread);
    return;
}
```

- The magic step is invoking `context_switch()`
- Why do we need to call `append_to_queue()`?
Thread Context Switch

- The **context switch routine** does all of the magic
  - Saves context of the currently running thread (old_thread)
  - Restores context of the next thread
  - The next thread becomes the current thread
  - Return to caller as new thread
  - In Pintos, it is the `switch_threads()` in switch.S

- This is all done in assembly language
  - It works at the level of the procedure calling convention, so it cannot be implemented using procedure calls
Preemptive Scheduling

- Non-preemptive threads have to voluntarily give up CPU
  - A long-running thread will take over the machine
  - Only voluntary calls to thread_yield(), thread_stop(), or thread_exit() causes a context switch

- Preemptive scheduling causes an involuntary context switch
  - Need to regain control of processor asynchronously
  - Use timer interrupt (How do you do this?)
  - Timer interrupt handler forces current thread to “call” thread_yield
Threads Summary

- Processes are too heavyweight for multiprocessing
  - Time and space overhead
- Solution is to separate threads from processes
  - Kernel-level threads much better, but still significant overhead
  - User-level threads even better, but not well integrated with OS
- Scheduling of threads can be either preemptive or non-preemptive
- Now, how do we get our threads to correctly cooperate with each other?
  - Synchronization…
Next time...

- Read
  - Chapter 5.1—5.3 in book