Lecture 4: Threads; weaving control flow

CSE 120: Principles of Operating Systems
Alex C. Snoeren

HW 1 Due NOW
Announcements

- Homework #1 due now
- Project 0 due tonight
- Project groups
  - Please send project group info to Jose
  - Project 1 will start on Thursday
Processes

- Recall that a process includes many things
  - An address space (defining all the code and data pages)
  - OS resources (e.g., open files) and accounting information
  - Execution state (PC, SP, regs, etc.)

- Creating a new process is costly because of all of the data structures that must be allocated and initialized
  - Recall struct proc in Solaris
  - ...which does not even include page tables, etc.

- Communicating between processes is costly because most communication goes through the OS
  - Overhead of system calls and copying data
Parallel Programs

- Also recall our Web server example that forks off copies of itself to handle multiple simultaneous requests
  - Or any parallel program that executes on a multiprocessor

- To execute these programs we need to
  - Create several processes that execute in parallel
  - Cause each to map to the same address space to share data
    - They are all part of the same computation
  - Have the OS schedule these processes in parallel (logically or physically)

- This situation is very inefficient
  - **Space**: PCB, page tables, etc.
  - **Time**: create data structures, fork and copy addr space, etc.
The Soul of a Process

- What is similar in these cooperating processes?
  - They all share the same code and data (address space)
  - They all share the same privileges
  - They all share the same resources (files, sockets, etc.)

- What don’t they share?
  - Each has its own execution state: PC, SP, and registers

- **Key idea**: Why don’t we separate the concept of a process from its execution state?
  - **Process**: address space, privileges, resources, etc.
  - **Execution state**: PC, SP, registers

- Exec state also called **thread of control**, or **thread**
Threads

- Modern OSes (Solaris, NT, modern Unix) separate the concepts of processes and threads
  - The thread defines a sequential execution stream within a process (PC, SP, registers)
  - The process defines the address space and general process attributes (everything but threads of execution)

- A thread is bound to a single process
  - Processes, however, can have multiple threads
  - Each process has at least one thread

- Threads become the unit of scheduling
  - Processes are now the containers in which threads execute
  - Processes become static, threads are the dynamic entities
  - Each CPU runs one thread at a time
Threads in a Process
Thread Design Space

<table>
<thead>
<tr>
<th>Address Space</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>One Thread per Process</td>
<td>One Thread per Process</td>
</tr>
<tr>
<td>One Address Space (MSDOS)</td>
<td>Many Address Spaces (Early Unix)</td>
</tr>
<tr>
<td>Many Threads per Process</td>
<td>Many Threads per Process</td>
</tr>
<tr>
<td>One Address Space (Java VM)</td>
<td>Many Address Spaces (Solaris, Linux, NT, MacOS)</td>
</tr>
</tbody>
</table>

CSE 120 – Lecture 4: Threads
Why Use Threads?

- Separating threads and processes makes it easier to support parallel applications
  - Creating concurrency does not require creating new processes
  - Low-overhead sharing between threads in same process
- Concurrency (multithreading) can be very useful
  - Improving program structure
  - Handling concurrent events (e.g., Web requests)
  - Taking advantage of multiple CPUs
  - Overlapping I/O with computation
- But, brings a whole new meaning to Spaghetti Code
  - Forcing OS students to learn about synchronization…
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
      }
  }
  ```
Instead, we can create a new thread for each request

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

```c
handle_request(int sock) {
    Process request
    close(sock);
}
```
Scheduling Threads

- No longer just scheduling processes, but threads
  - Kernel scheduler used to pick among PCBs
  - Now what?
- We have basically two options
  - Kernel explicitly selects among threads in a process
  - Hide threads from the kernel, and have a user-level scheduler inside each multi-threaded process
- Why do we care?
  - Think about the overhead of switching between threads
  - Who decides which thread in a process should go first?
  - What about blocking system calls?
Kernel-Level Threads

- 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
  - NT: threads
  - Solaris: lightweight processes (LWP)

- Scheduler deals in threads
  - PCBs are no longer scheduled
  - If a thread blocks, another thread in the same process can run
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
  - User-level threads are managed entirely by a run-time system (a.k.a. user-level thread library)
- Invisible to kernel
  - A thread represented inside process by a PC, registers, stack, and small thread control block (TCB)
  - Creating a new thread, switching, and synchronizing threads are done via user-level procedure call
  - User-level thread operations 100x faster than kernel threads
User 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
Another possibility is to use both kernel and user-level threads

- Can associate a user-level thread with a kernel-level thread
- Or, multiplex user-level threads on top of kernel-level threads

Java Virtual Machine (JVM)

- Java threads are user-level threads
- On older Unix, only one “kernel thread” per process
  - Multiplex all Java threads on this one kernel thread
- On NT, modern Unix
  - Can multiplex Java threads on multiple kernel threads
  - Can have more Java threads than kernel threads
  - Why?
Multiplexing user-level threads on a single kernel thread for each process

Multiplexing user-level threads on multiple kernel threads for each process
Implementing Threads

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

- Focus on user-level threads
  - Kernel-level threads are similar to original process management and implementation in the OS
  - What you will be dealing with in Nachos
  - Not only will you be *using* threads in Nachos, you will be *implementing* more thread functionality
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
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
  - But it is implemented at user-level in a library
- 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?
Wait a second. How does `thread_yield()` work?

The semantics of `thread_yield` are that it gives up the CPU to another thread
- In other words, it context switches to another thread

So what does it mean for `thread_yield` to return?
- It means that another thread called `thread_yield`!

Execution trace of ping/pong
- `printf("ping\n");`
- `thread_yield();`
- `printf("pong\n");`
- `thread_yield();`
- ...
Implementing thread_yield()

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()?
The context switch routine does all of the magic:
- Saves context of the currently running thread (old_thread)
  - Push all machine state onto its stack (not its TCB)
- Restores context of the next thread
  - Pop all machine state from the next thread’s stack
- The next thread becomes the current thread
- Return to caller as new thread

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
- See code/threads/switch.s in Nachos
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
  - Timer interrupt handler forces current thread to “call” thread_yield
    - How do you do this?
  - **Nachos is preemptive**
    - See use of thread->yieldOnReturn in code/machine/interrupt.cc
Threads Summary

- The operating system as a large multithreaded program
  - Each process executes as a thread within the OS
- Multithreading is also very useful for applications
  - Efficient multithreading requires fast primitives
  - Processes are too heavyweight
- 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
- Now, how do we get our threads to correctly cooperate with each other?
  - Synchronization…
Next time…

- Read Chapter 6.1—6.6
Announcements

- UCSD CSE Programming Contest Saturday (10/8)
  - Why should you go? To be like John:

http://www-cse.ucsd.edu/users/calder/UCSDProgramContest/