Adminstrivia

• Lab
  • Lab1: design documents can be submitted now

• Homework
  • Homework1 will be released tomorrow or Friday
Recap: threads

- Modern OS (Mac OS, Windows, Linux) separate the concepts of processes and threads
  - The **thread** defines a sequential execution stream (PC, SP, registers)
  - The **process** defines the address space and general process attributes (the subject abstraction)
- A thread is bound to a single process
  - Processes, however, can have multiple threads
- Threads become the unit of scheduling
Recap: process components

• Per process components
  • PID, Address space
  • Static states: code, data, etc.
  • OS resources: files, network connections, etc.

• Per thread components
  • Execution stack
  • Processors states: PC, general-purpose registers, etc.
  • Execution state: running, ready, wait, etc.
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
Concurrent servers using process

- 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
    }
}
```
Concurrent servers using threads

• 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 with 3 threads
Kernel-level threads

- After separated thread(s) from process, the OS now manages both 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
  - Windows: threads
  - Solaris: lightweight processes (LWP)
  - POSIX Threads (pthreads): PTHREAD_SCOPE_SYSTEM
Kernel-level threads 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.
User-level threads

- User-level threads are managed entirely by 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
  - 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
User-level threads limitations

- User-level threads are *invisible* to the OS, so 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 is important
  - For programming (correctness, performance)
Implementing threads

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

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

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

- `thread_exit()`
  - Terminate the calling thread

- Also `thread_destroy()`
User-level thread scheduling

- The thread scheduler determines when a thread runs
- It uses queues to keep track of what threads are doing
  - Just like the kernel scheduler
  - 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()`

```c
Ping Thread

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

```c
Pong Thread

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

- The semantic of `thread_yield()` is to give 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?
- Execution trace of ping/pong

```c
printf("ping\n");
thread_yield();
printf("pong\n");
thread_yield();
...```
Implementing thread_yield

```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)
    - 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
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()`
Summary

- Separation of process and thread
  - More efficient multiprocessing
- Tow implementation levels
  - Kernel-level threads vs. user-level threads
- Thread interface
  - Create, stop, \texttt{yield}, exit, etc.
- Thread scheduling
  - Non-preemptive vs. preemptive
Additional question to ponder

- How do we get our threads to correctly cooperate with each other?
  - Synchronization
For next class ...

- Synchronization
- Textbook
  - Module 28, 29