CSE120
Principles of Operating Systems

Prof Yuanyuan (YY) Zhou

Lecture 4: Threads

Announcement

- Project 0 Due
- Project 1 out

- Homework 1 due on Thursday
  - Submit it to Gradescope online

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

Parallel Programs

- To execute these programs we need to
  - Create several processes that execute in parallel
  - 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.

- Solution: possible to have cooperating "*processes*"?

Rethinking Processes

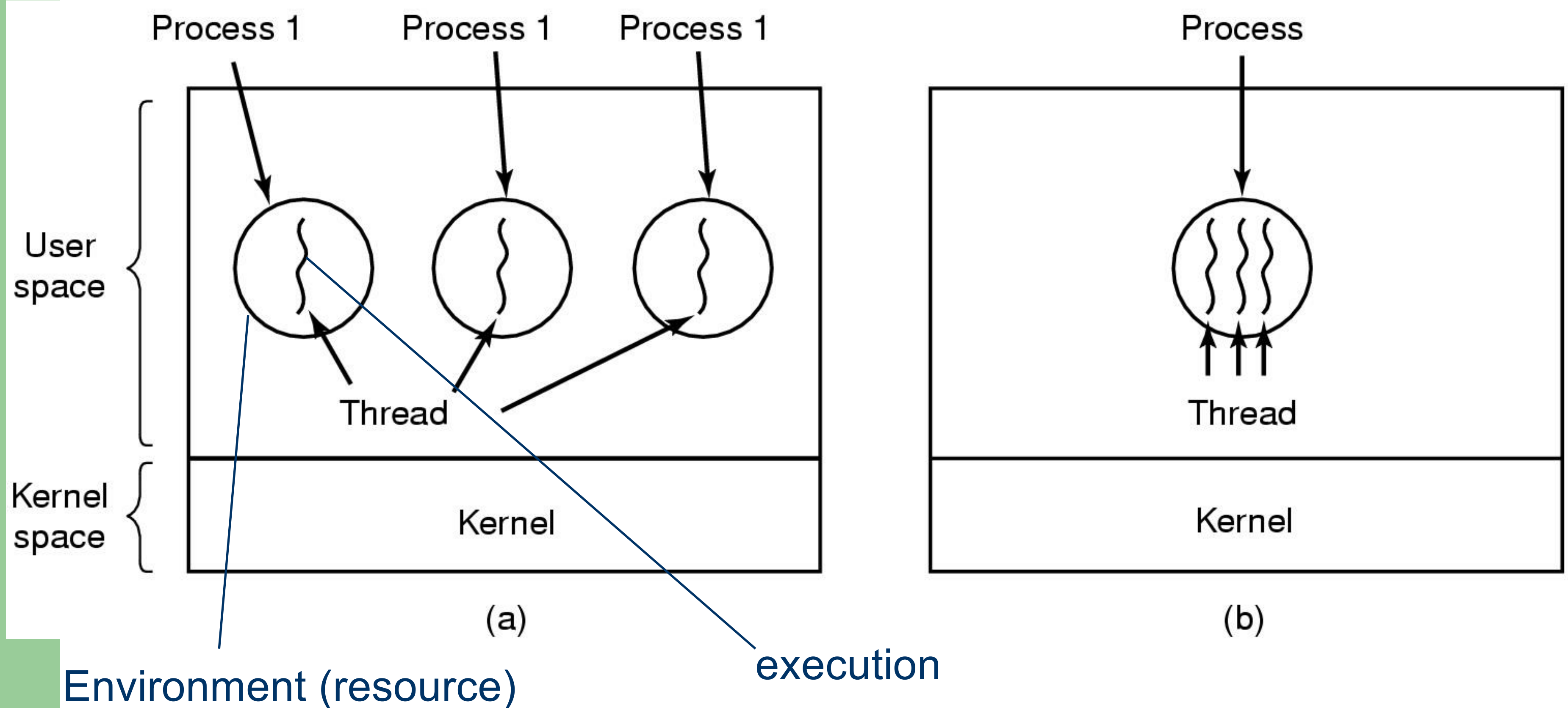
- Key idea: Why don't we separate the concept of a process from its execution state?
  - Process: address space, privileges, resources, etc.
  - Execution state: PC, SP, registers
- Exec state also called thread of control, or thread

Threads

- Thread vs Process
  - A thread defines a sequential execution stream within a process (PC, SP, registers)
  - A process defines the address space and general process attributes (everything but threads of execution)
- A thread is bound to a single process
  - A process, however, can have multiple threads
- Threads become the unit of scheduling
  - Processes are now the containers in which threads execute

Threads: Lightweight Processes

A sequential execution stream within a process



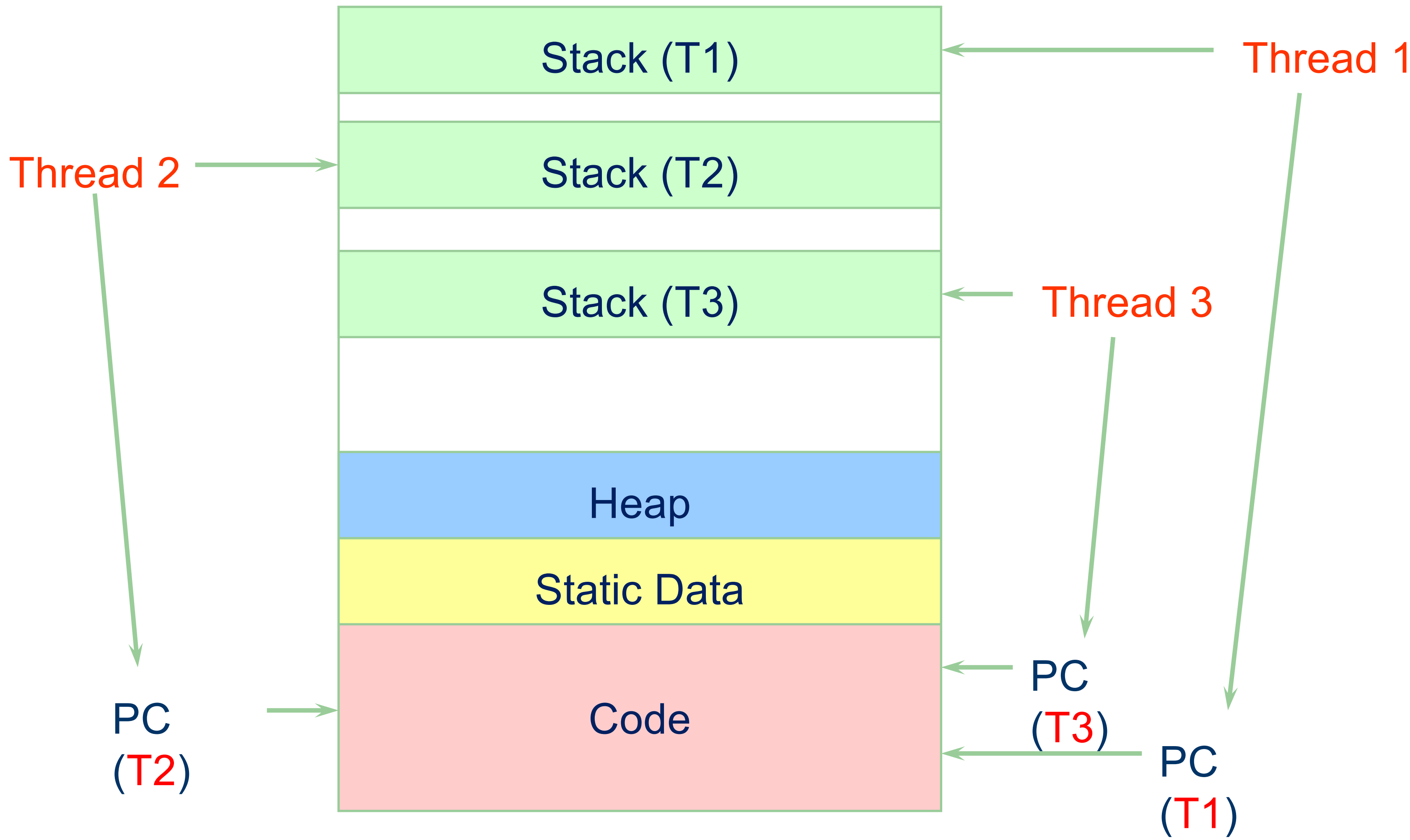
(a) Three processes each with one thread

(b) One process with three threads

The Thread Model

- Shared information
  - Processor info: parent process, time, etc
  - Memory: global data, heap, page table, and stats, etc
  - I/O and file: communication ports, directories and file descriptors, etc
- Private state
  - State (ready, running and blocked)
  - Registers
  - Program counter
  - Execution stack
  - **Why?**
- Each thread execute separately

Threads in a Process



Analogy

- Process: 3 projects for different classes (CSE120, CSE140, CSE110)
  - Each one has different text book, different web pages, different TAs/Instructors

- Threads: 3 activities in CSE120 (Homework, Lectures, Projects)
  - Share the same concepts (textbook)
  - Share TA/Tutors
  - All of them are going on in parallel (within one quarter)
  - Each has their own things, too

A 2-min Explainer Video

- https://www.youtube.com/watch?v=Dhf-DYO1K78

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:

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

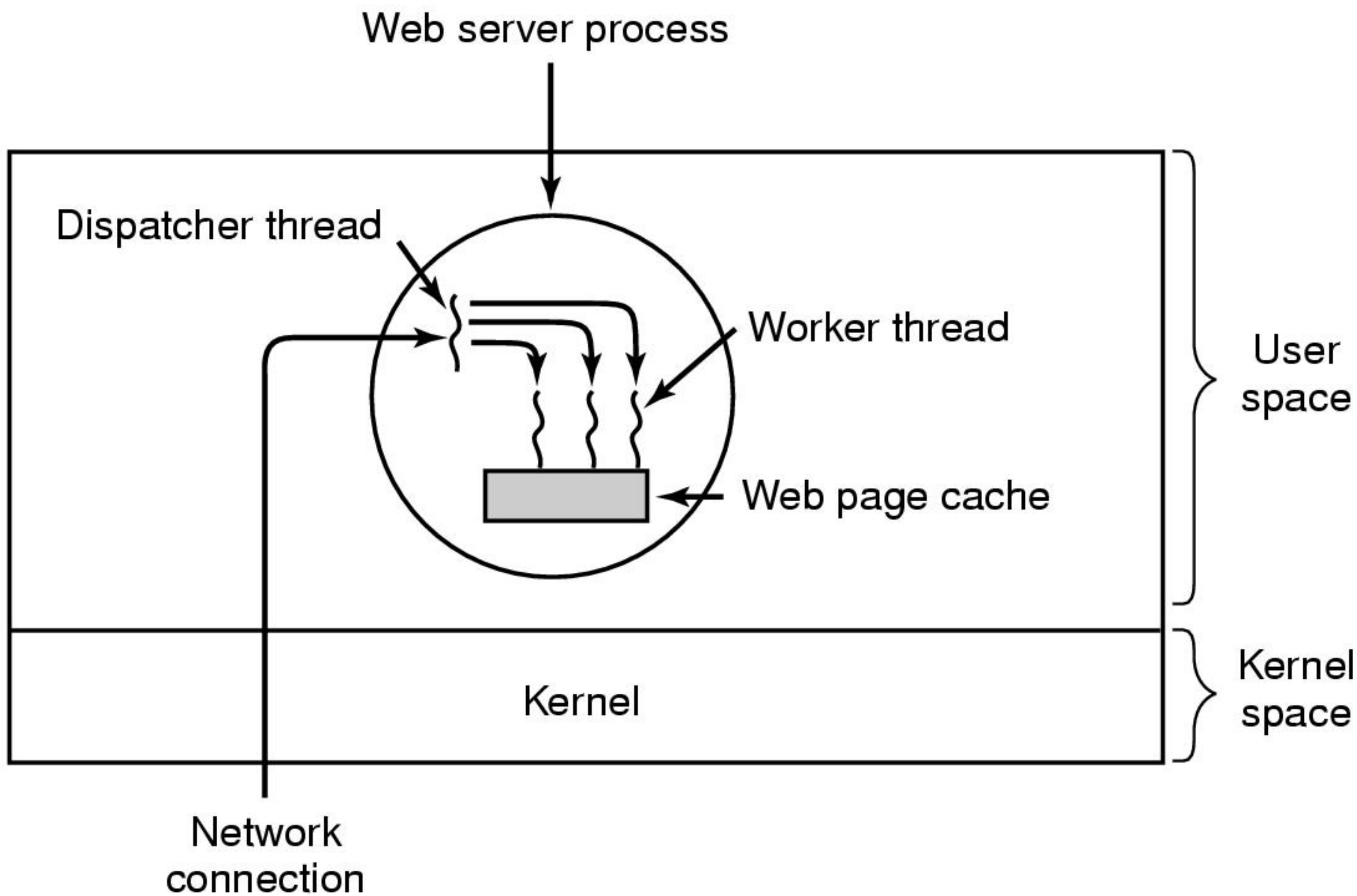
Threads: Concurrent Servers

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

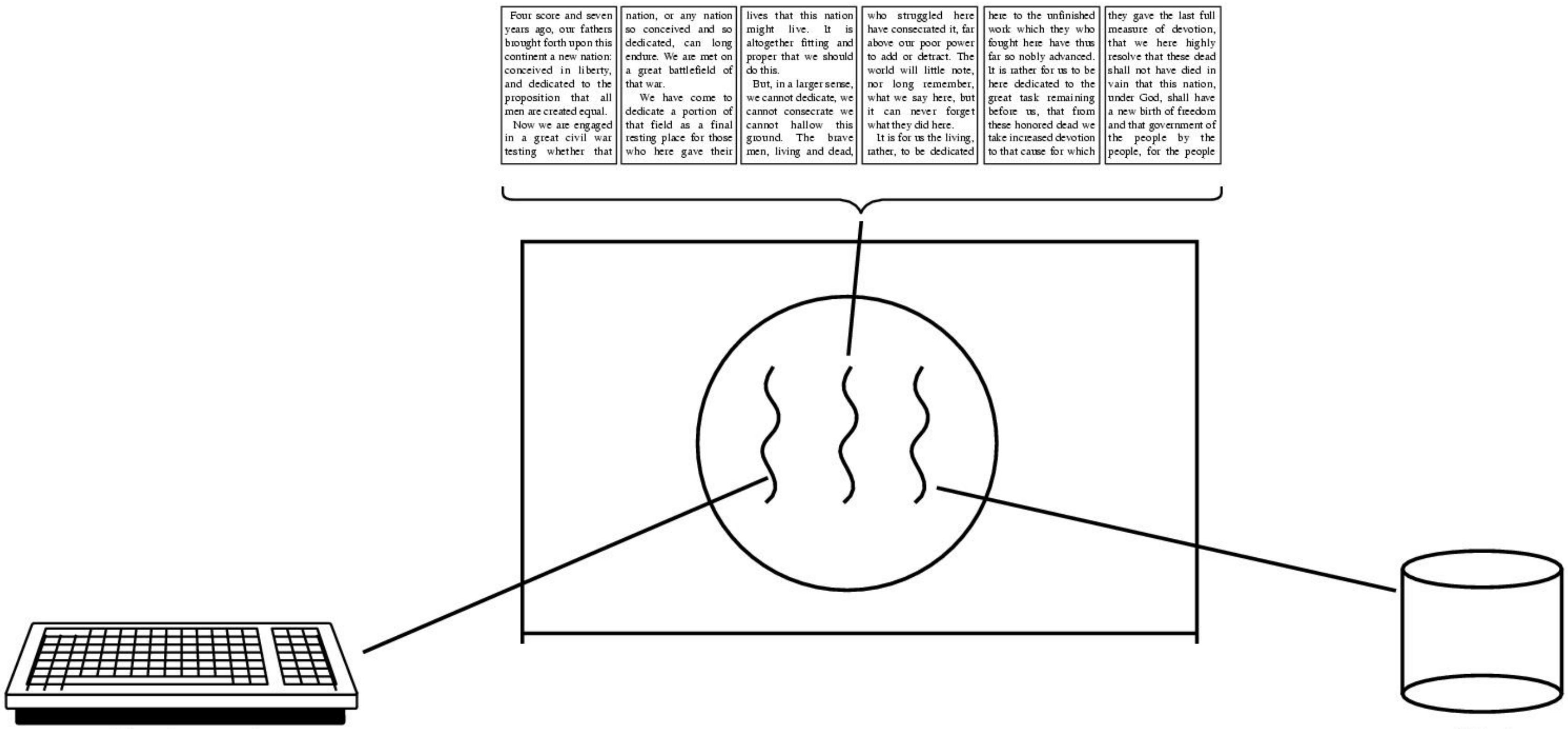
```
web_server() {
   while (1) {
     int sock = accept();
     thread_fork(handle_request, sock);
   }
}

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

Difference from fork()?

Thread Usage: Web Server

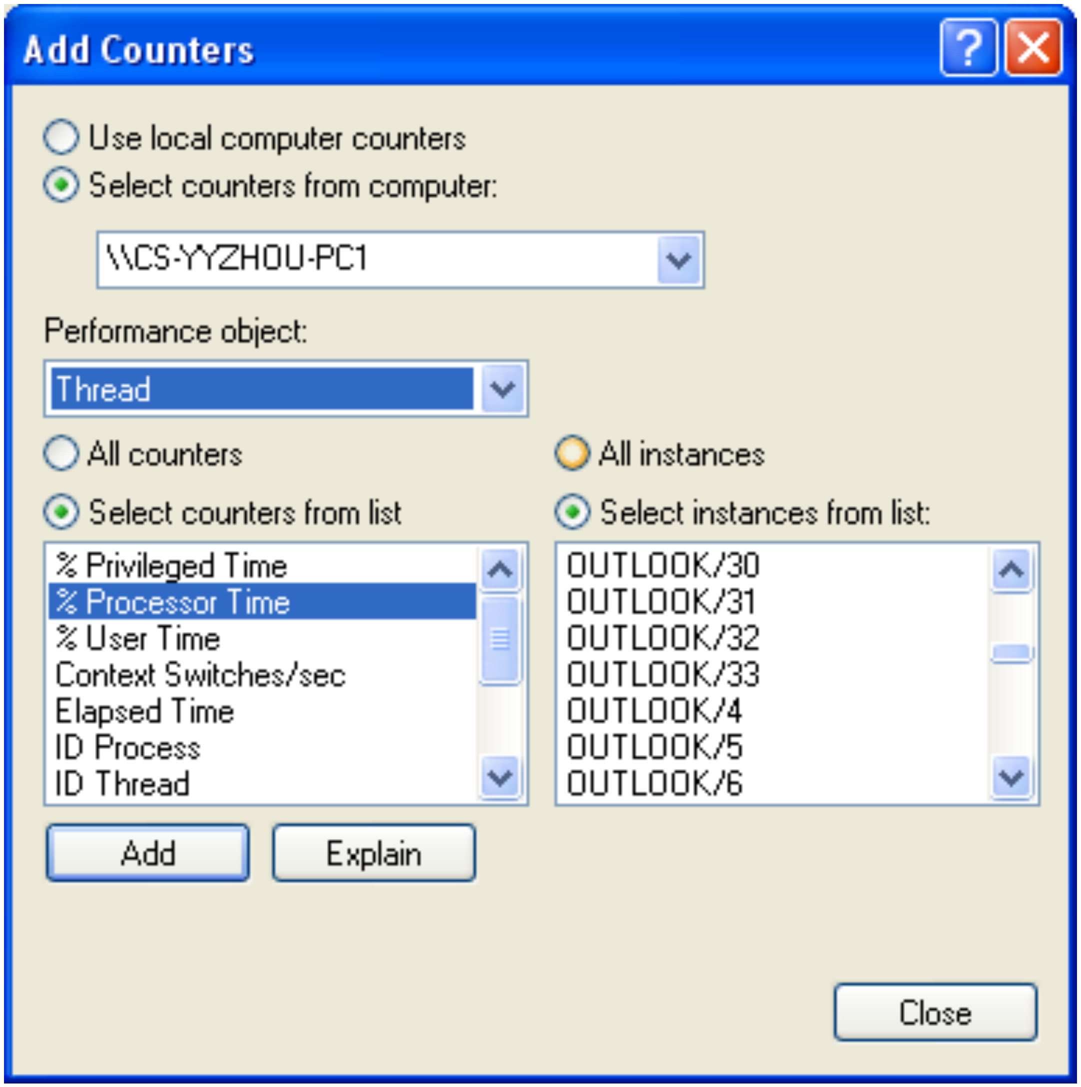


Thread Usage: word processor

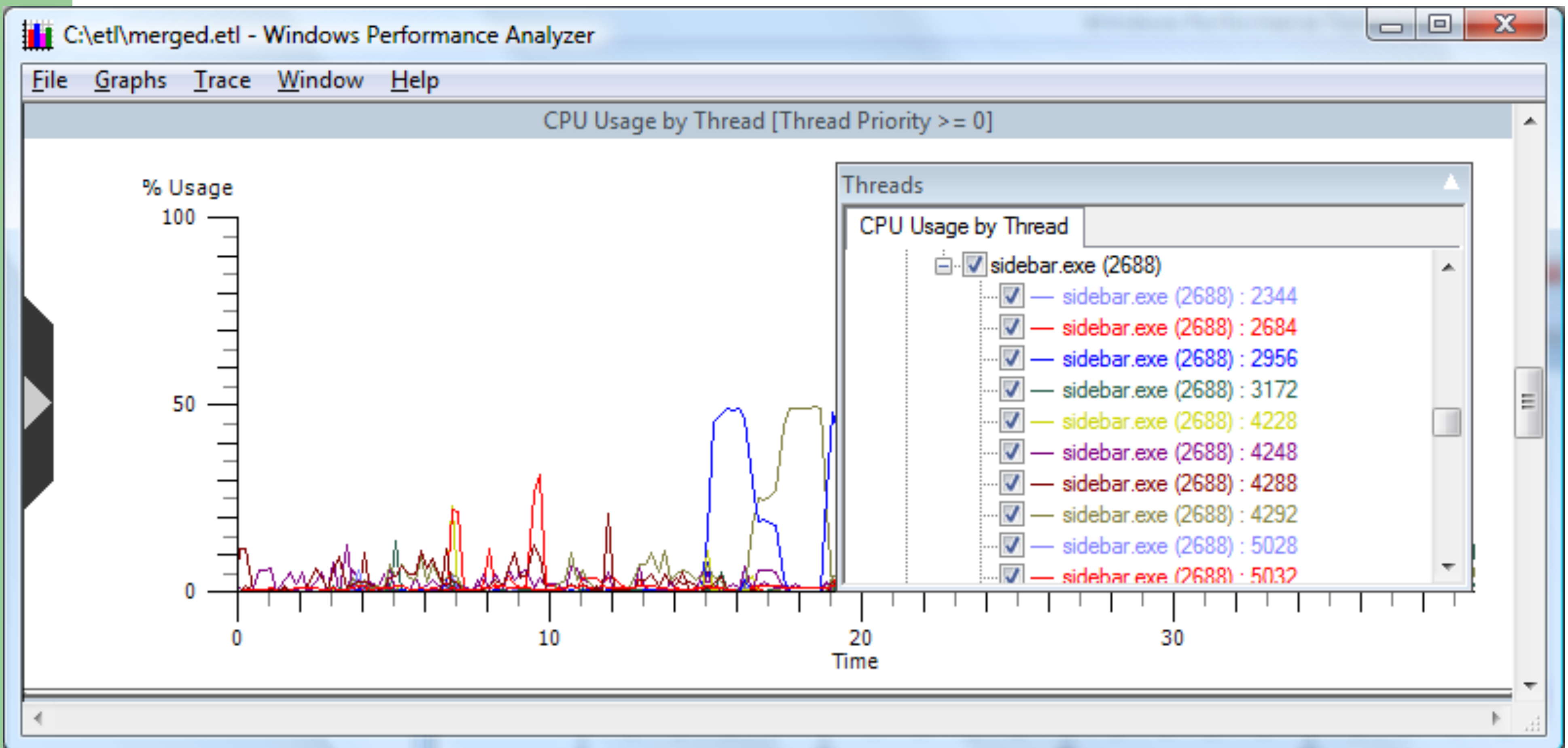


- A thread can wait for I/O, while the other threads can still running.
- What if it is single-threaded?

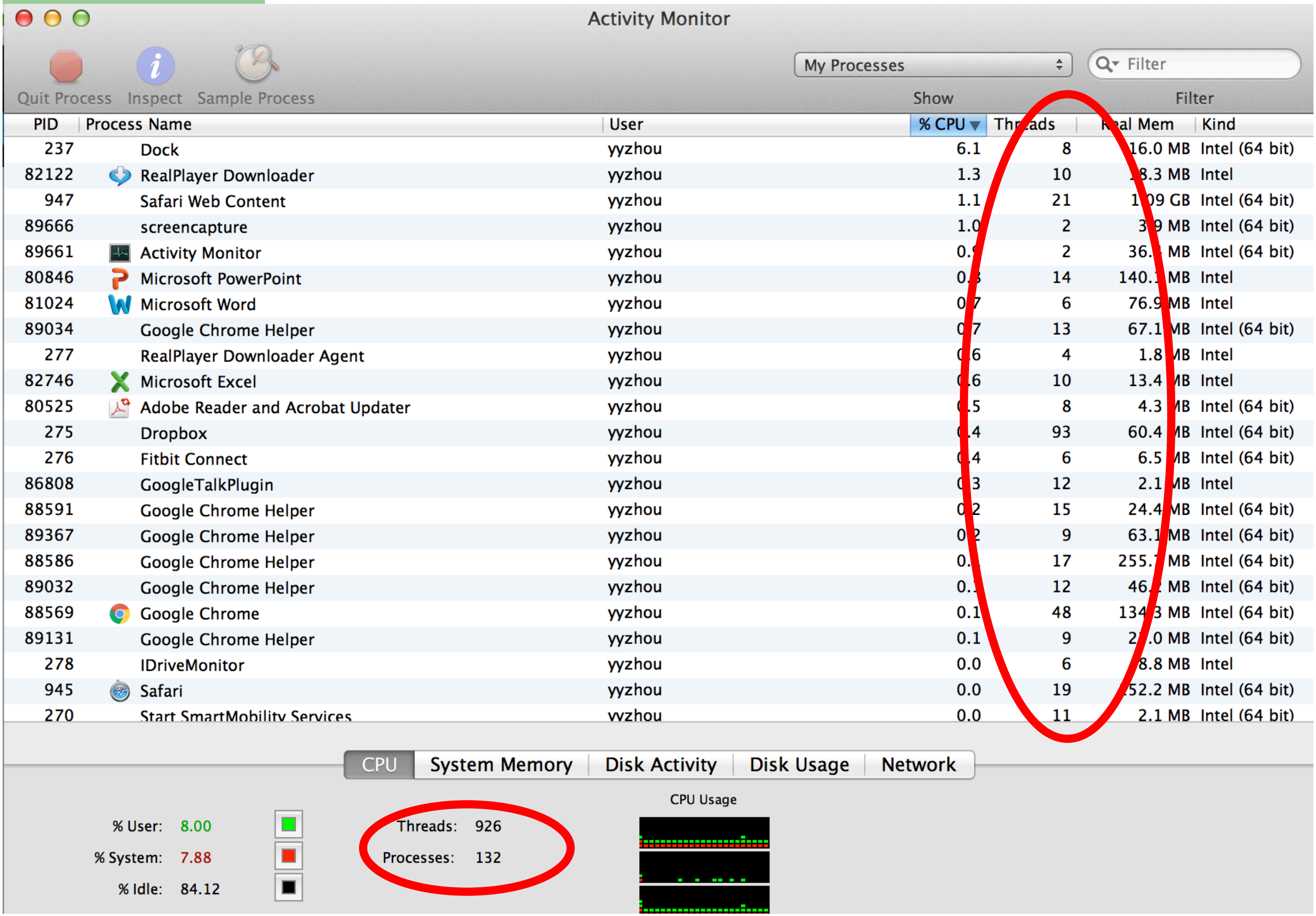
Windows Thread Lists from Performance Monitor



Windows Performance Analyzer



Mac OS – Activity Monitor



PID	Process Name	User	% CPU	Threads	Real Mem	Kind
237	Dock	yyzhou	6.1	8	16.0 MB	Intel (64 bit)
82122	RealPlayer Downloader	yyzhou	1.3	10	[illegible]8.3 MB	Intel
947	Safari Web Content	yyzhou	1.1	21	1.09 GB	Intel (64 bit)
89666	screencapture	yyzhou	1.0	2	3.9 MB	Intel (64 bit)
89661	Activity Monitor	yyzhou	0.[illegible]	2	36.[illegible] MB	Intel (64 bit)
80846	Microsoft PowerPoint	yyzhou	0.[illegible]	14	140.[illegible] MB	Intel
81024	Microsoft Word	yyzhou	0.7	6	76.9 MB	Intel
89034	Google Chrome Helper	yyzhou	0.7	13	67.1 MB	Intel (64 bit)
277	RealPlayer Downloader Agent	yyzhou	0.6	4	1.8 MB	Intel
82746	Microsoft Excel	yyzhou	0.6	10	13.4 MB	Intel
80525	Adobe Reader and Acrobat Updater	yyzhou	0.5	8	4.3 MB	Intel (64 bit)
275	Dropbox	yyzhou	0.4	93	60.4 MB	Intel (64 bit)
276	Fitbit Connect	yyzhou	0.4	6	6.5 MB	Intel (64 bit)
86808	GoogleTalkPlugin	yyzhou	0.3	12	2.1 MB	Intel
88591	Google Chrome Helper	yyzhou	0.2	15	24.4 MB	Intel (64 bit)
89367	Google Chrome Helper	yyzhou	0.2	9	63.1 MB	Intel (64 bit)
88586	Google Chrome Helper	yyzhou	0.[illegible]	17	255.[illegible] MB	Intel (64 bit)
89032	Google Chrome Helper	yyzhou	0.[illegible]	12	46.[illegible] MB	Intel (64 bit)
88569	Google Chrome	yyzhou	0.1	48	134.3 MB	Intel (64 bit)
89131	Google Chrome Helper	yyzhou	0.1	9	2[illegible].0 MB	Intel (64 bit)
278	IDriveMonitor	yyzhou	0.0	6	[illegible]8.8 MB	Intel
945	Safari	yyzhou	0.0	19	[illegible]52.2 MB	Intel (64 bit)
270	Start SmartMobility Services	yyzhou	0.0	11	2.1 MB	Intel (64 bit)

Thread Information on Linux

- Process information:
  - Read **/proc/[your PID]/stat** file
- Thread information (2.6 kernel):
  - Read **/proc/[your PID]/task/[thread ID]/stat**

Kernel-managed 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 Kernel managed threads or lightweight processes
  - NT: threads
  - Solaris: lightweight processes (LWP)
  - POSIX Threads (pthreads): PTHREAD_SCOPE_SYSTEM

Kernel-managed Thread Limitations

- Kernel-managed threads make concurrency much cheaper than processes
  - Much less state to allocate and initialize
- However, for fine-grained concurrency, kernel-managed 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 more fine-grained concurrency, need even “cheaper” threads

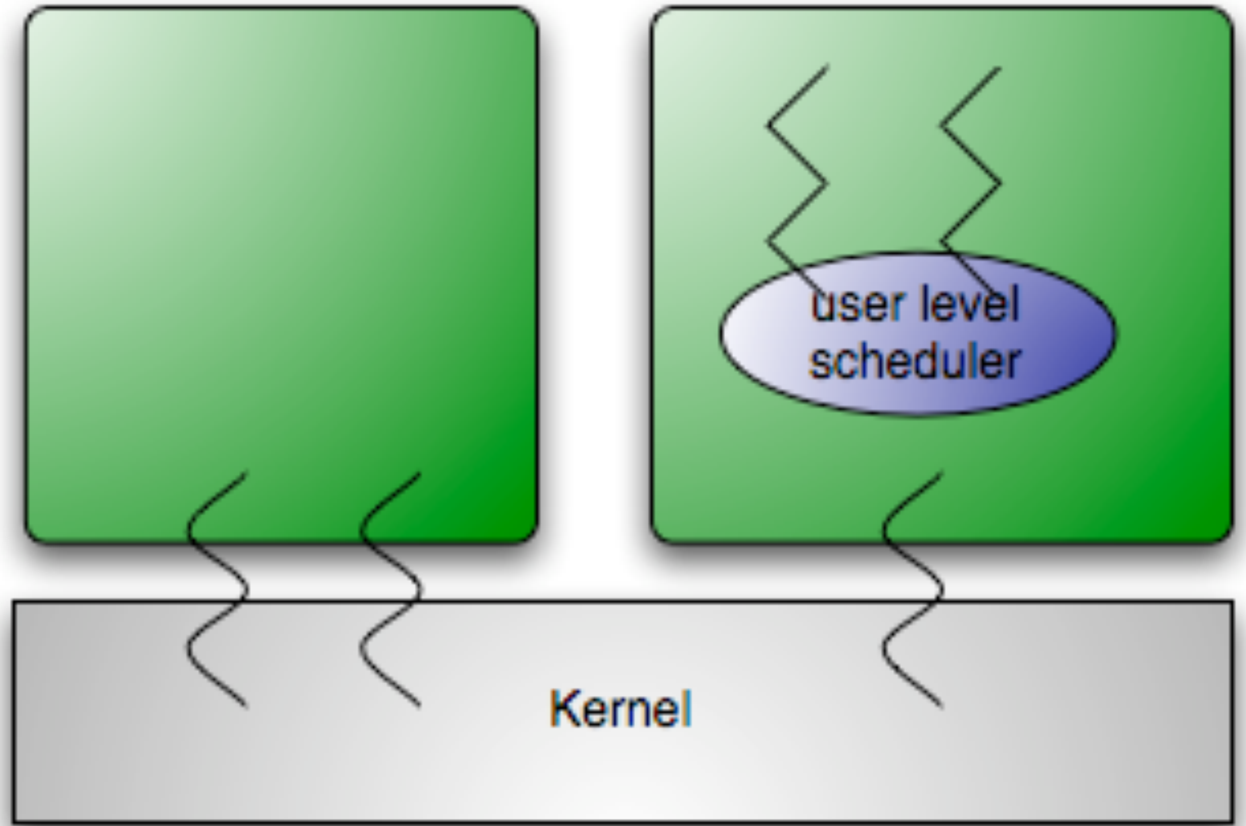
User-Level-Managed Threads

- To make threads cheap and fast, they need to be implemented at user level
  - Kernel-level managed threads are managed by the OS
  - User-level managed threads are managed entirely by the run-time system (user-level library)
- User-level-managed 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 managed thread operations 100x faster than kernel managed threads
  - pthreads: PTHREAD_SCOPE_PROCESS

User level threads

Kernel level Managed threads

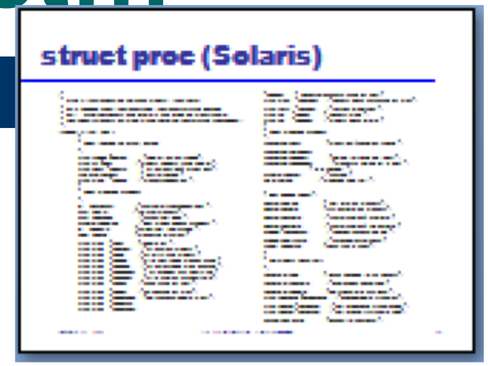
User level Managed threads

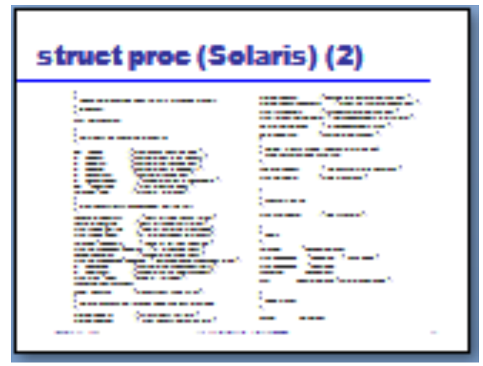


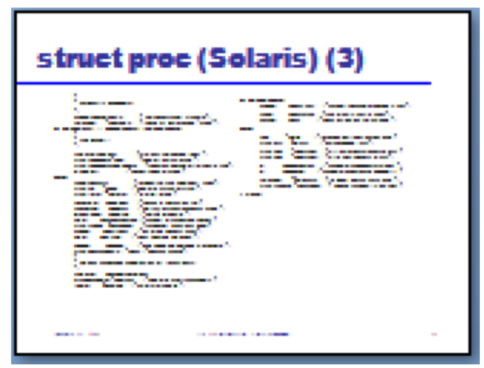
Small and Fast…

- Nachos thread control block

```
class Thread {
    int *stack;
    int *stackTop;
    int machineState[MachineStateSize];
    ThreadStatus status;
    char *name;
    <Methods>
};
```







User Level Managed Thread Limitations

- But, user-level managed threads are not a perfect solution
  - As with everything else, they are a tradeoff
- User-level managed 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

Tradeoffs between the two

- Kernel-level managed threads
  - Integrated with OS (informed scheduling)
  - Slow to create, manipulate, synchronize
- User-level managed threads
  - Fast to create, manipulate, synchronize
  - Not integrated with OS (uninformed scheduling)
- Understanding the differences between kernel and user-level managed threads is important
  - For programming (correctness, performance)

Kernel and User Threads

- Or 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) (also pthreads)
  - 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

Implementing Threads

- Implementing threads has a number of issues
  - Interface
  - Context switch
  - Preemptive vs. non-preemptive
  - Scheduling
  - Synchronization (next lecture)
- Focus on user-level managed threads
  - Kernel-level managed 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

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
- Thread_join(t) or t.join()
  - causes the current thread to pause execution until t's thread terminates

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

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

Pong Thread

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

- What is the output of running these two threads?

thread_yield()

- 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;
}
```

As old thread

As new thread

- 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 the 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 CPU
  - 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
  - How?
    - Use timer/alarm interrupt

Blocking Vs. non-blocking System Calls

- Blocking system call
  - Usually I/O related: read(), fread(), getc(), write()
  - Doesn't return until the call completes
  - The process/thread is switched to blocked state
  - When the I/O completes, the process/thread becomes ready
  - Simple
  - Real life example: attending a lecture
- Using non-blocking system call for I/O
  - Asynchronous I/O
  - Complicated
  - The call returns once the I/O is initiated, and the caller continue
  - Once the I/O completes, an interrupt is delivered to the caller
  - Real life example: apply for job

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 managed threads much better, but still significant overhead
  - User-level managed threads even better, but not well integrated with OS
- Now, how do we get our threads to correctly cooperate with each other?
  - Synchronization…