Operating Systems (A)
(Honor Track)

Lecture 7: Threads

Yao Guo (郭耀)

Peking University

Fall 2021

Acknowledgements: Prof. Xiangqun Chen & Tao Wang at PKU and Prof. Yuanyuan Zhou at UCSD

Review: Process

- What are the units of execution?
  - Processes
- How are those units of execution represented?
  - Process Control Blocks (PCBs)
- How is work scheduled in the CPU?
  - Process states, process queues, context switches
- What are the possible execution states of a process?
  - Running, ready, waiting
- How does a process move from one state to another?
  - Scheduling, I/O, creation, termination
- How are processes created?
  - CreateProcess (NT), fork/exec Unix)

Modeling Multiprogramming

- Multiprogramming can improve processor utilization
- Question: Suppose that a process spends a fraction *p* of its time waiting for I/O to complete. With *n* processes in memory at once, what is the expected CPU utilization?

- The probability that all n processes are waiting for I/O (in which case the CPU will be idle) is p^n.

CPU utilization = $1 - p^n$

Degree of multiprogramming

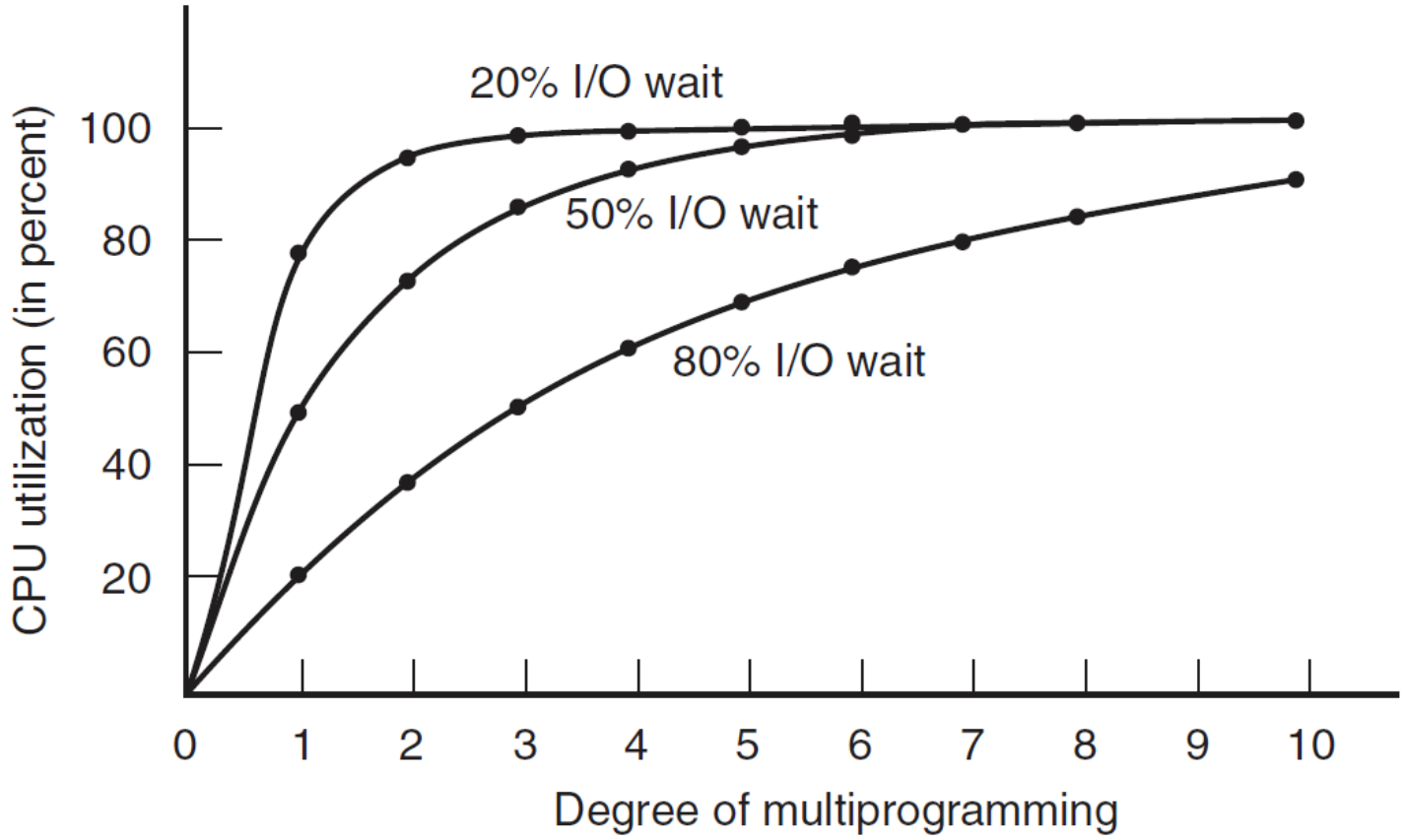


Figure 2-6. CPU utilization as a function of the number of processes in memory.

This Lecture

Threads

What is a thread?

Kernel-level and user-level threads

Implementing threads

Buzz Words

Thread

Multithreading

Sharing

Scheduling

This Lecture

Threads

What is a thread?

Kernel-level and user-level threads

Implementing threads

Issues in 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

Concurrent Programs

- 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 program, 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
  - 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 address space, etc.

Rethinking Processes

- 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 (Windows, Unix, Mac OS X) 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
  - A process, however, can have multiple threads
- Threads become the unit of scheduling
  - Processes are now the containers in which threads execute
  - Processes become static, threads are the dynamic entities

Threads: Lightweight Processes

A sequential execution stream within a process

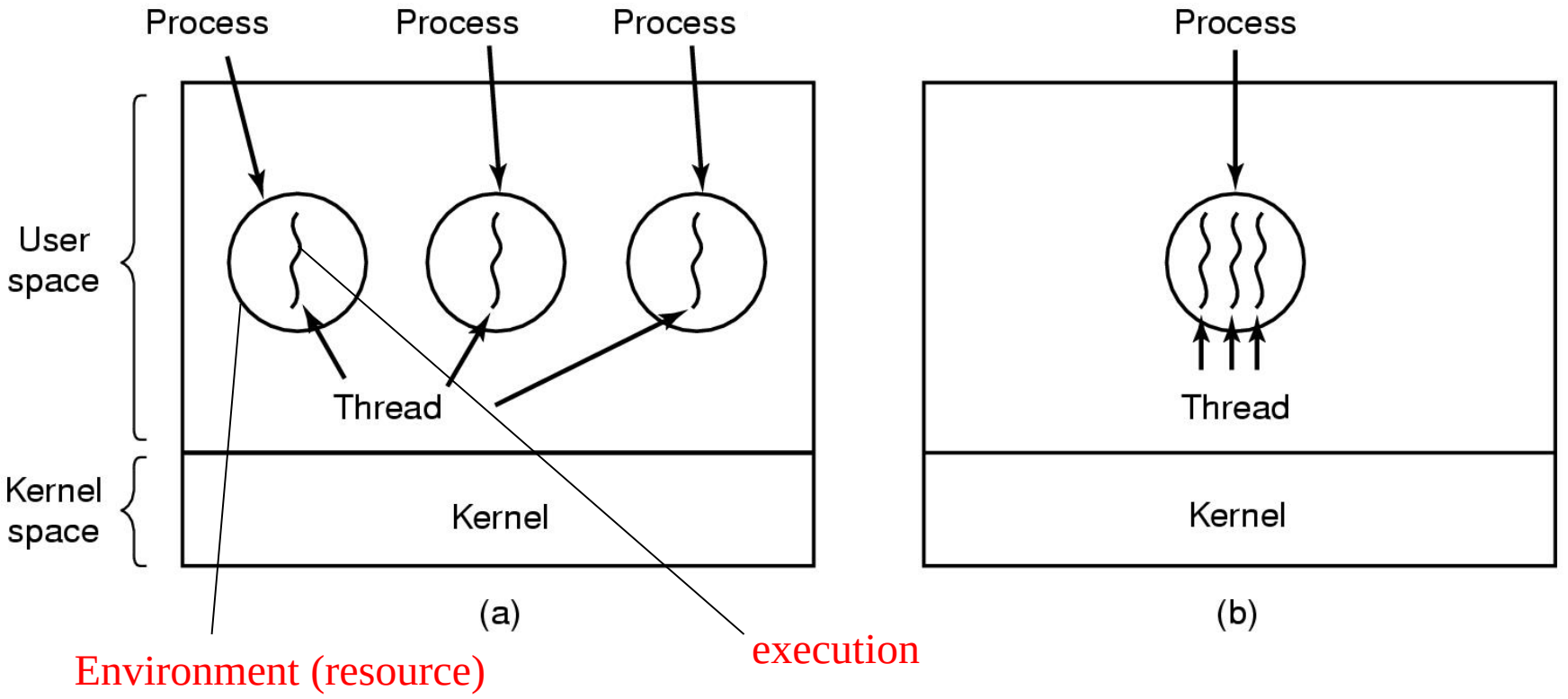


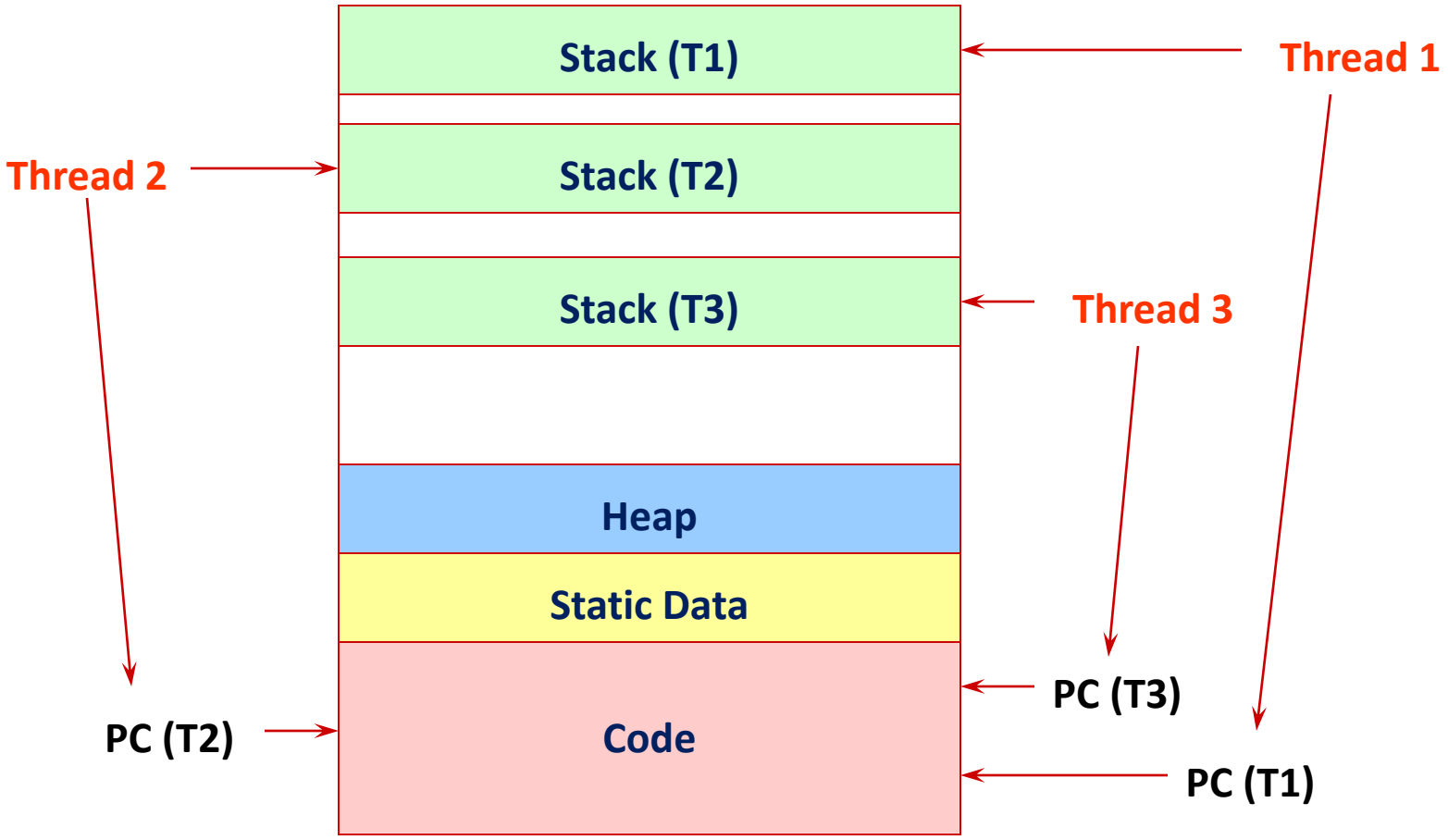
Figure 2-11. (a) Three processes each with one thread. (b) One process with three threads.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

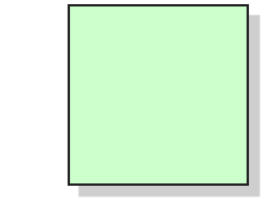
The Thread Model

- Shared information
  - Process info: parent process, child processes, time, etc
  - Memory: segments, page table, and stats, etc
  - I/O and file: communication ports, directories and file descriptors, etc
- Private state
  - State (ready, running and waiting)
  - Registers
  - Program counter
  - Execution stack
- Each thread executes separately

Threads in a Process



Thread Design Space

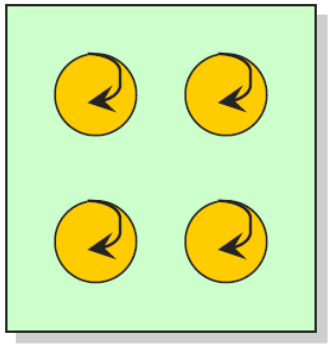


Address Space

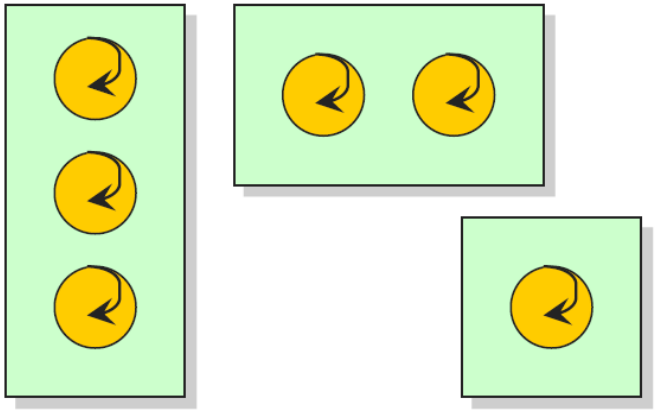
Thread

One Thread/Process
One Address Space
(MSDOS)

One Thread/Process
Many Address Spaces
(Early Unix)



Many Threads/Process
One Address Space
(Pilot, Java)



Many Threads/Process
Many Address Spaces (Mach, Unix, Windows, OS X)

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
  - Although today even cell phones are multicore

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);
}
```

Thread Usage: Web Server

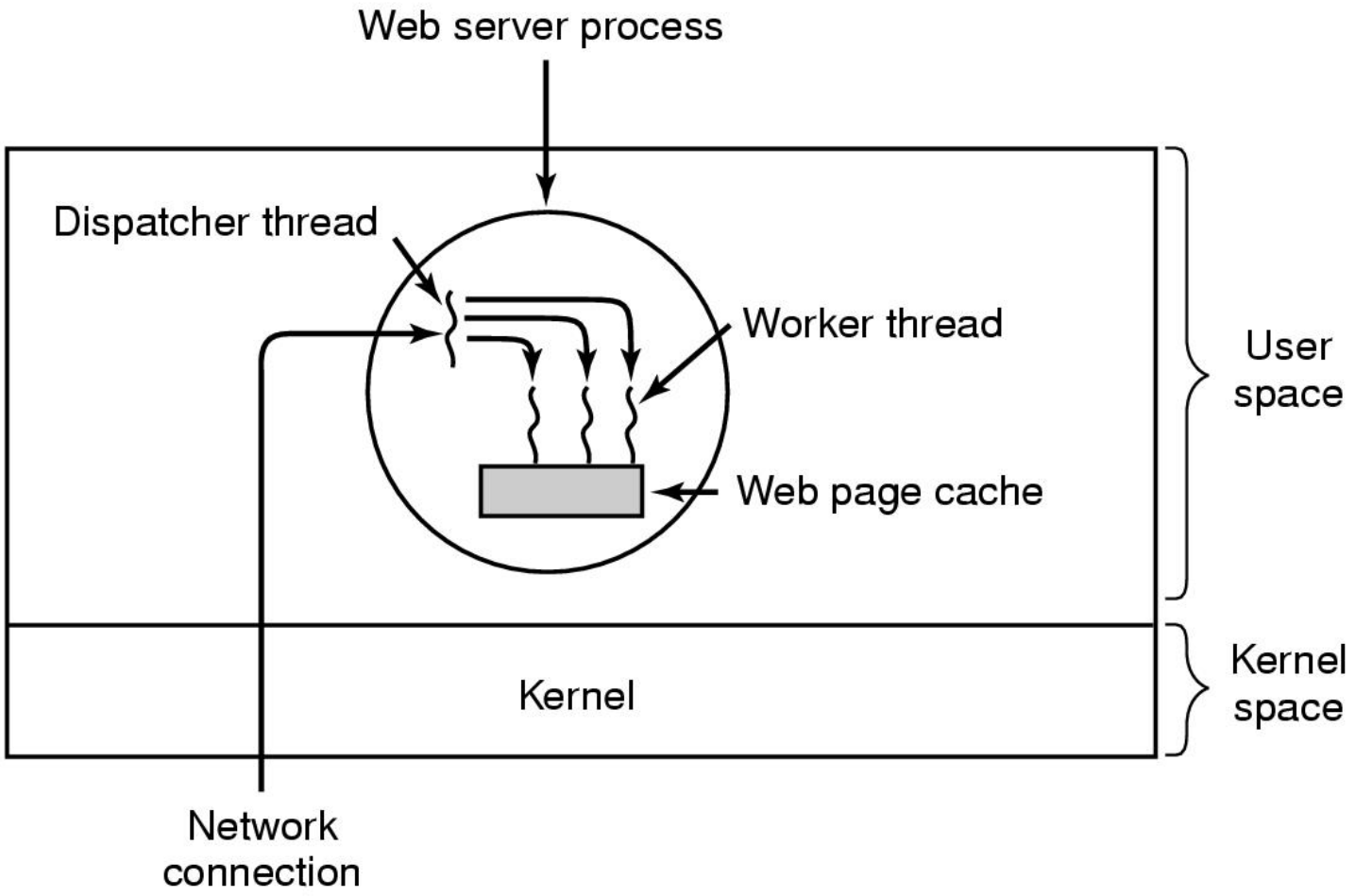


Figure 2-8. A multithreaded Web server.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Thread Usage: Word Processor

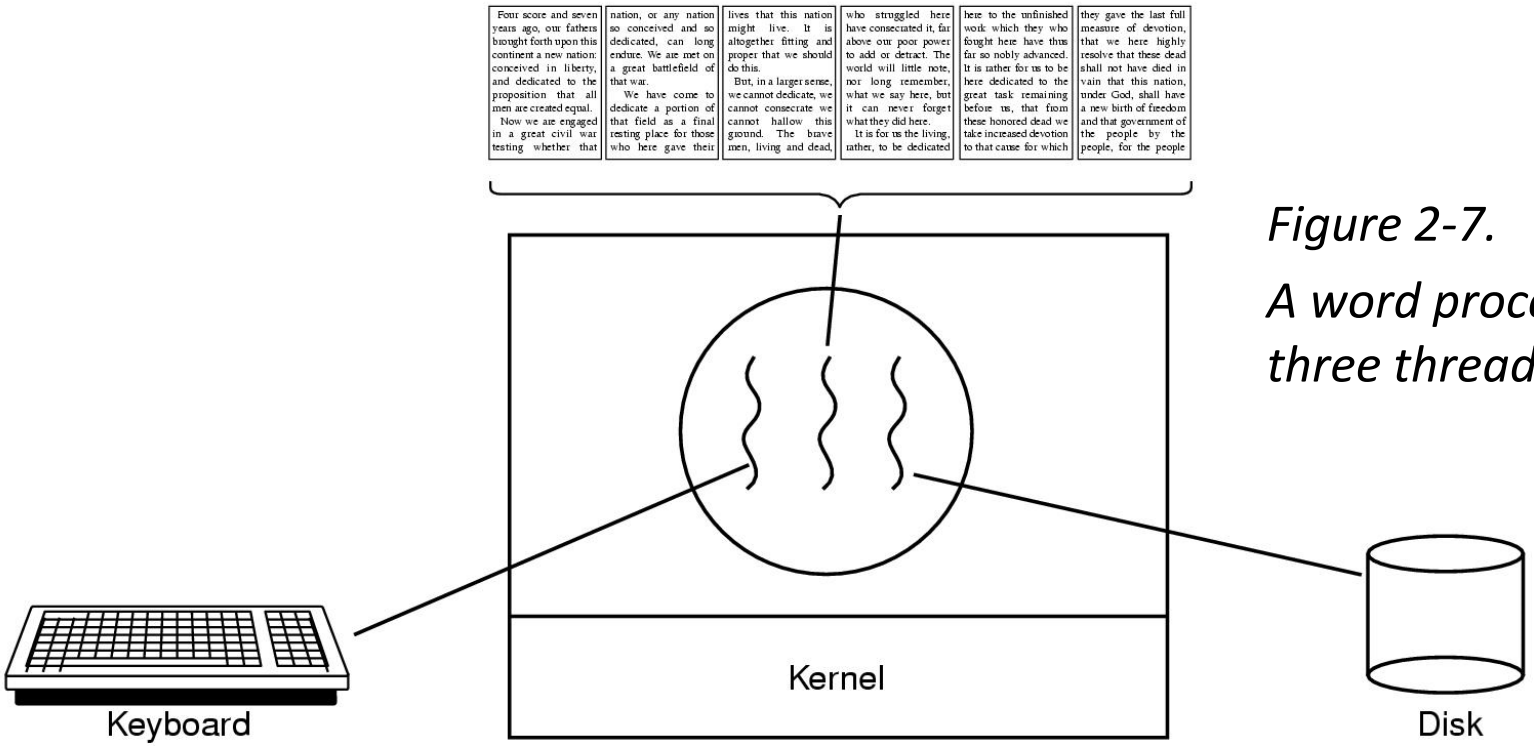


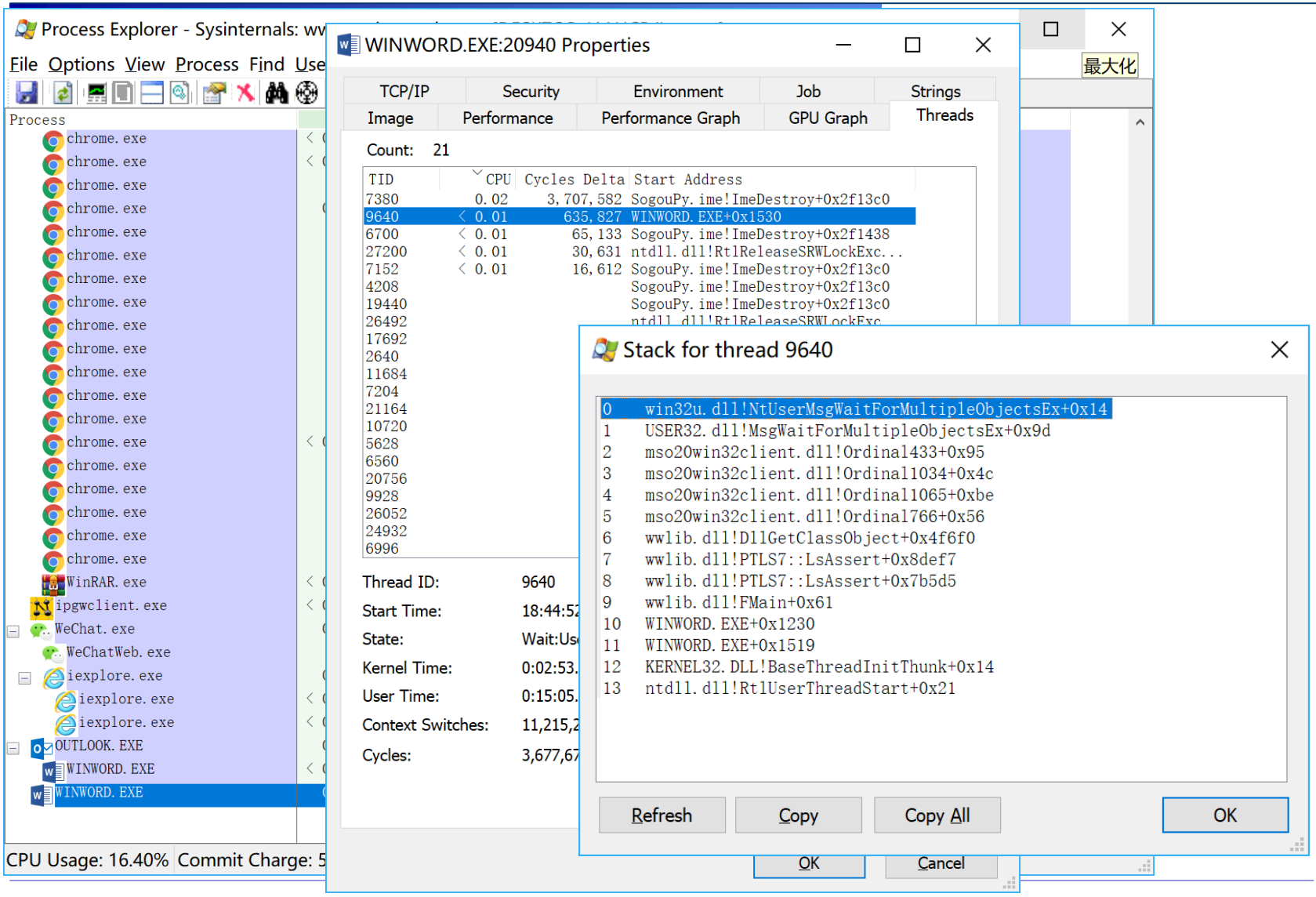
Figure 2-7.

A word processor with three threads.

- A thread can wait for I/O, while the other threads can still running
- What if it is single-threaded?
- Can we use three processes instead?

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Windows Threads from Process Explorer



Thread Information on Linux

- Process information:
  - Read **/proc/[PID]/stat** file
- Thread information:
  - Read **/proc/[PID]/task/[thread ID]/stat**

This Lecture

Threads

What is a thread?

Kernel-level and user-level threads

Implementing threads

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

Kernel-Level Thread Limitations

- Kernel-level threads make concurrency much cheaper than processes
  - Much fewer states 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 more fine-grained concurrency, need even "cheaper" threads

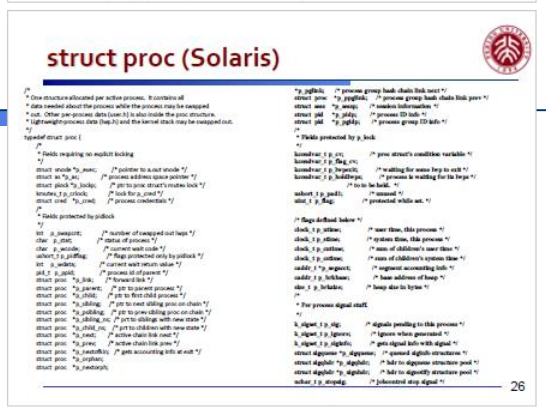
User-Level Threads

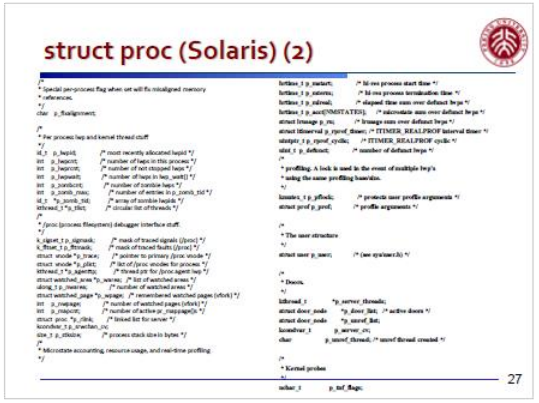
- To make threads cheap and fast, they need to be implemented at user level
  - Kernel-level threads are managed by the OS
  - User-level threads are managed entirely by the run-time system (user-level library)
- User-level threads are small and fast
  - A thread is simply represented by a PC, registers, stack, and small **thread control block** (TCB)
  - Creating a new thread, switching between threads, and synchronizing threads are done via procedure call
    - User-level thread operations are 100x faster than kernel-level threads
  - pthreads: PTHREAD_SCOPE_PROCESS

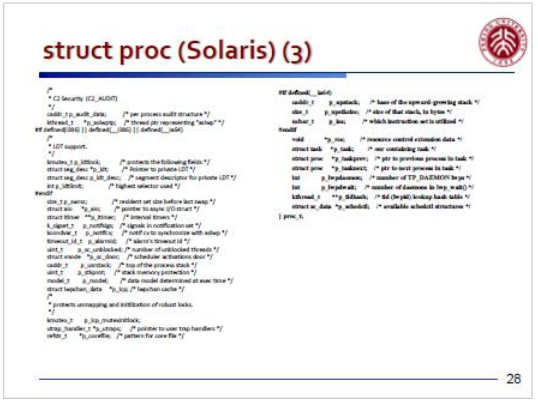
Small but Fast

- Nachos thread class

```
public class KThread{
    int status;
    String name;
    Runnable target;
    TCB tcb;
    int id;
    <Methods>
};
```







User-Level Thread Limitations

- But, user-level threads are not a perfect solution
  - As with everything else, there is 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
  - …
- Solving this requires communication between the kernel and the user-level thread manager

Kernel vs. User Threads

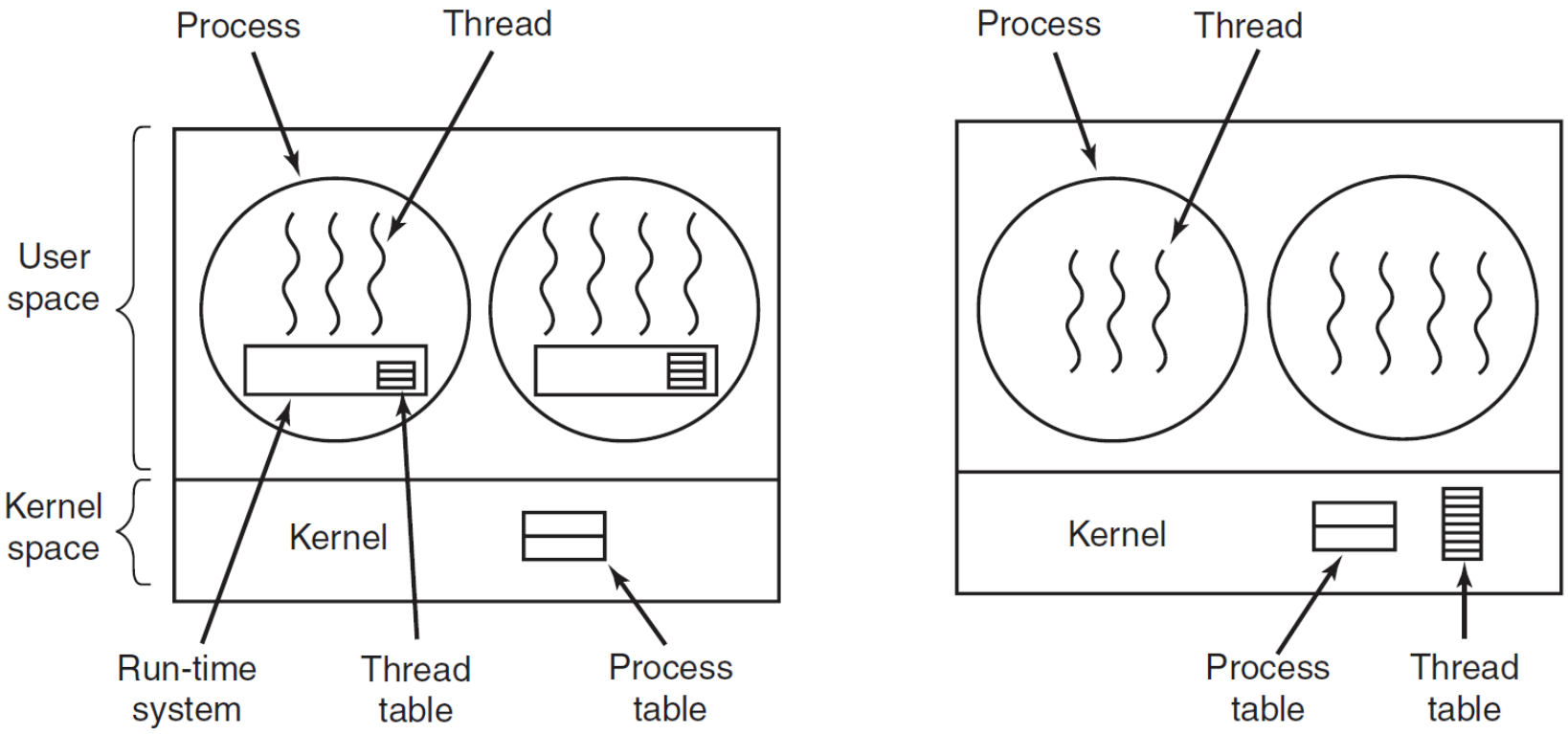


Figure 2-16. (a) A user-level threads package. (b) A threads package managed by the kernel.

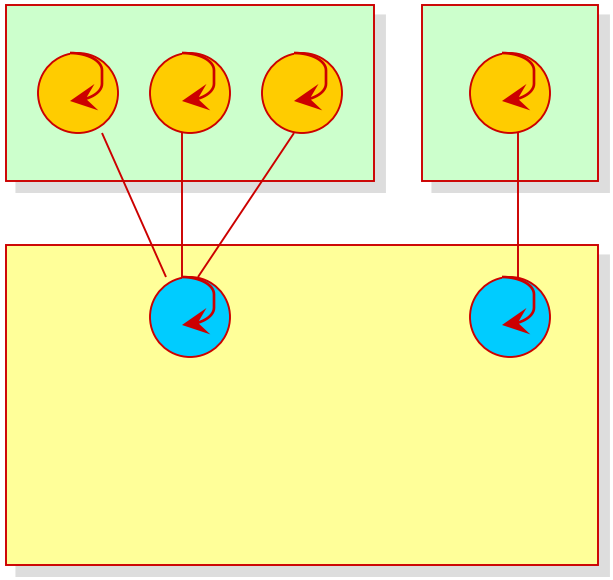
Kernel vs. User Threads

- 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 (especially for performance)

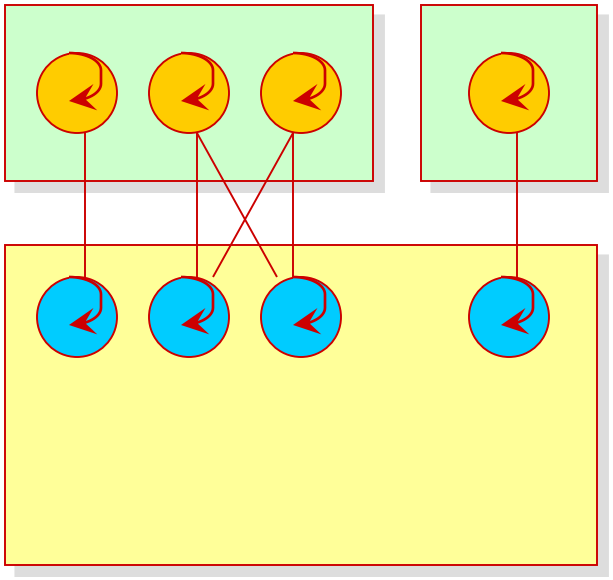
Kernel and User Threads

- 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 C#, others)
  - Java threads are user-level threads
  - On older Unix, only one “kernel-level thread” per process
    - Multiplex all Java threads on this one kernel-level thread
  - On modern OSes
    - Can multiplex Java threads on multiple kernel-level threads
    - Can have more Java threads than kernel-level threads

User and Kernel Threads



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

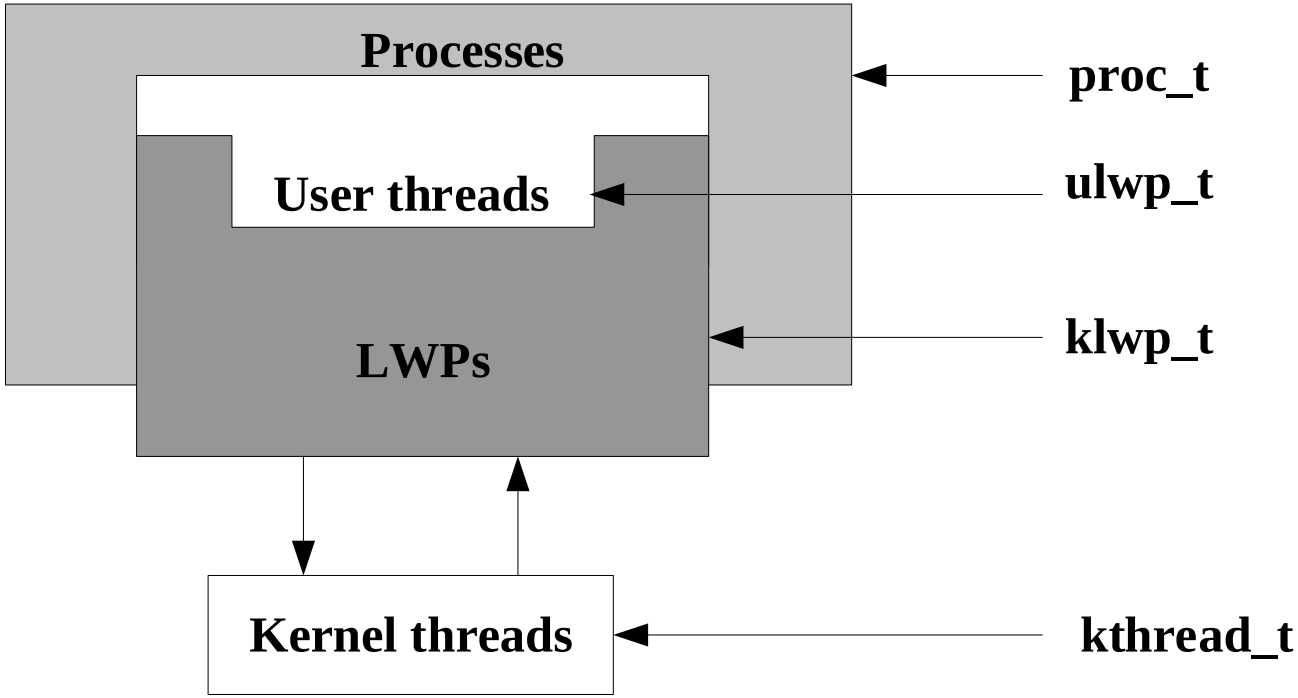


Multiplexing user-level threads on multiple kernel-level threads for each process

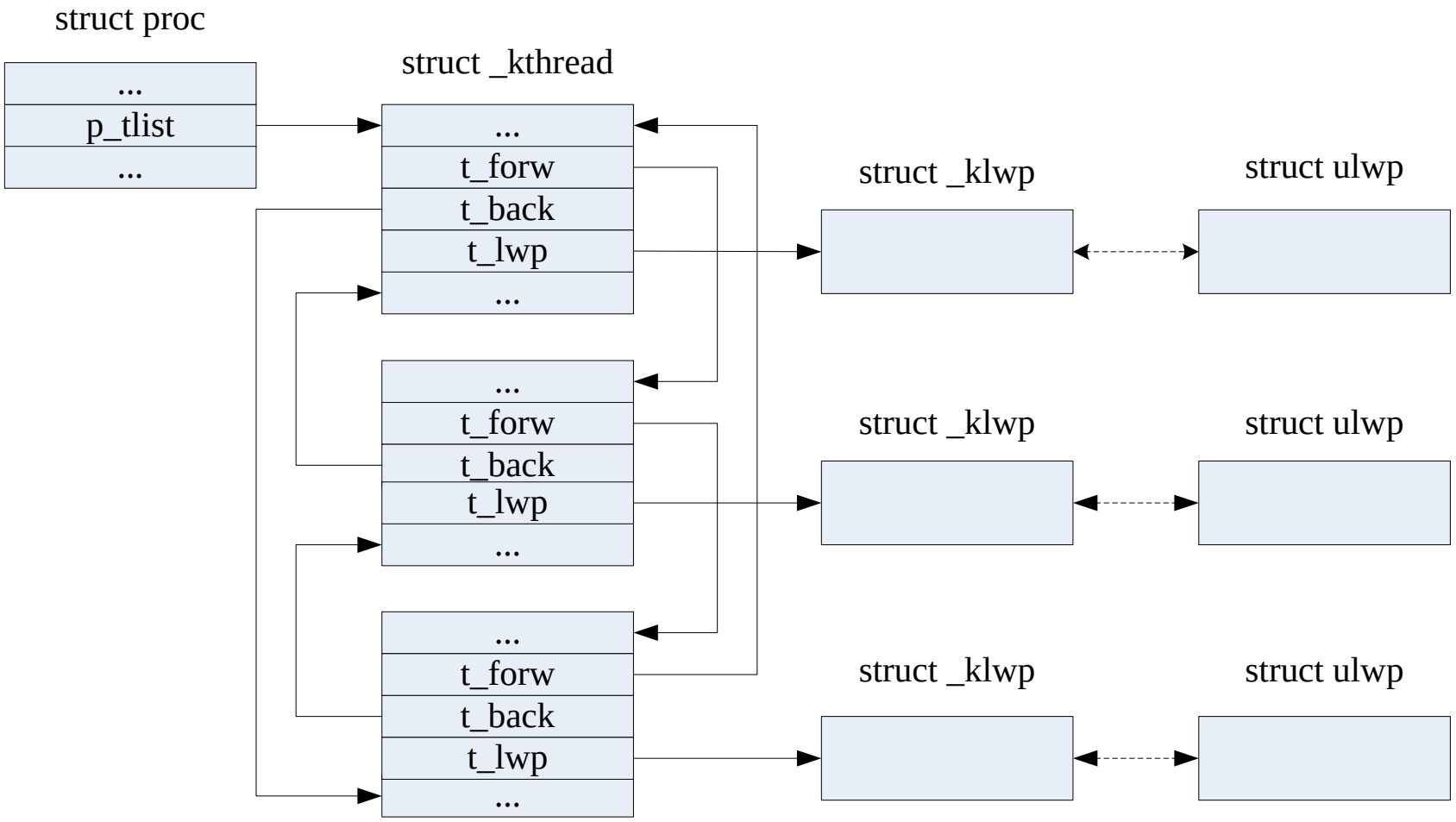
Case Study: Solaris Processes/Threads

- Multiprogramming mechanism in Solaris
  - Processes
    - Resource management
  - Kernel threads
    - Kernel scheduling
  - User threads
    - Abstraction of program in user mode
  - Light-weight processes (LWPs)
    - Binding user threads and kernel threads together

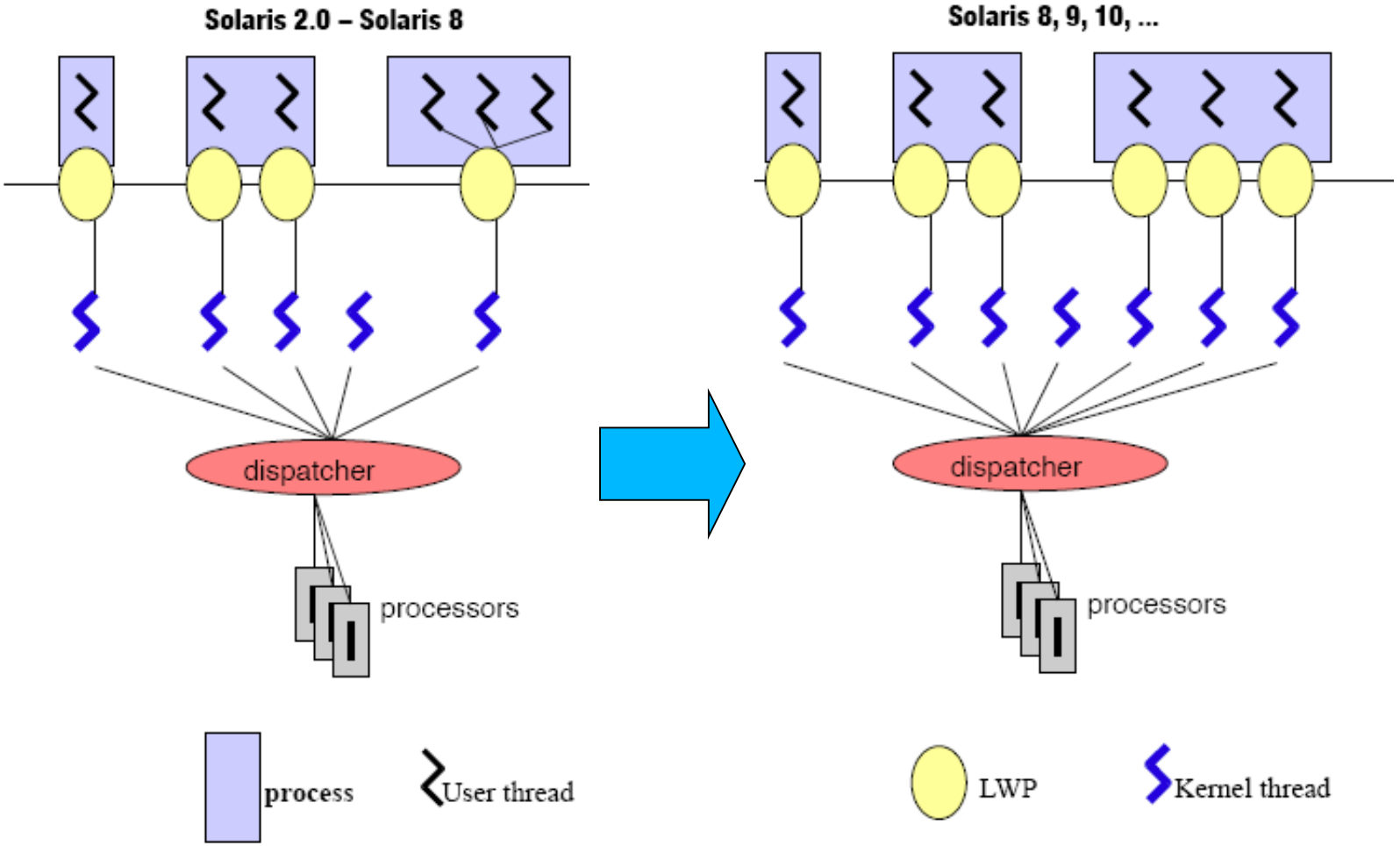
Process/Thread implementation in Solaris



Processes, threads, LWPs



Process/thread Evolution in Solaris



This Lecture

Threads

What is a thread?

Kernel-level and user-level threads

Implementing threads

Implementing Threads

- Implementing threads has a number of issues
  - Interface
  - Scheduling
  - Non-preemptive vs. preemptive scheduling
  - Scheduling
  - Synchronization
  - …
- Focus on user-level threads now
  - Kernel-level threads are similar to original process management and implementation in the OS

Sample Thread Interface

- thread_fork(procedure_t)
  - Create a new thread of control
  - Also thread_create(), thread_setstate()
- thread_exit()
  - Terminate the calling thread; also thread_destroy
- 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 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()?

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 the 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 uses involuntary context switch
  - Need to regain control of processor asynchronously
  - How?
    - Use timer interrupt
  - Timer interrupt handler forces current thread to "call" thread_yield
    - How do you do this?

Other Thread Schemes

- POSIX threads
- Scheduler activations
- Pop-up threads

POSIX Threads

- IEEE standard 1003.1c for threads
  - The threads package it defines is called Pthreads
  - Defines over 60 function calls, supported by most UNIX systems

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Figure 2-14. Some of the Pthreads function calls.

Scheduler Activations

- Goal: Get the best of both worlds — the efficiency of user-level threads and the non-blocking ability of kernel threads
- Relies on *upcalls*
  - Normally, user programs call functions in the kernel
  - Sometimes, though, the kernel calls a user-level process to report an event
  - Sometimes considered unclean — violates usual layering

ANDERSON, T.E., BERSHAD, B.N., LAZOWSKA, E.D., and LEVY, H.M.: "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism," ACM Trans. on Computer Systems, vol. 10, pp. 53–79, Feb. 1992.

Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism

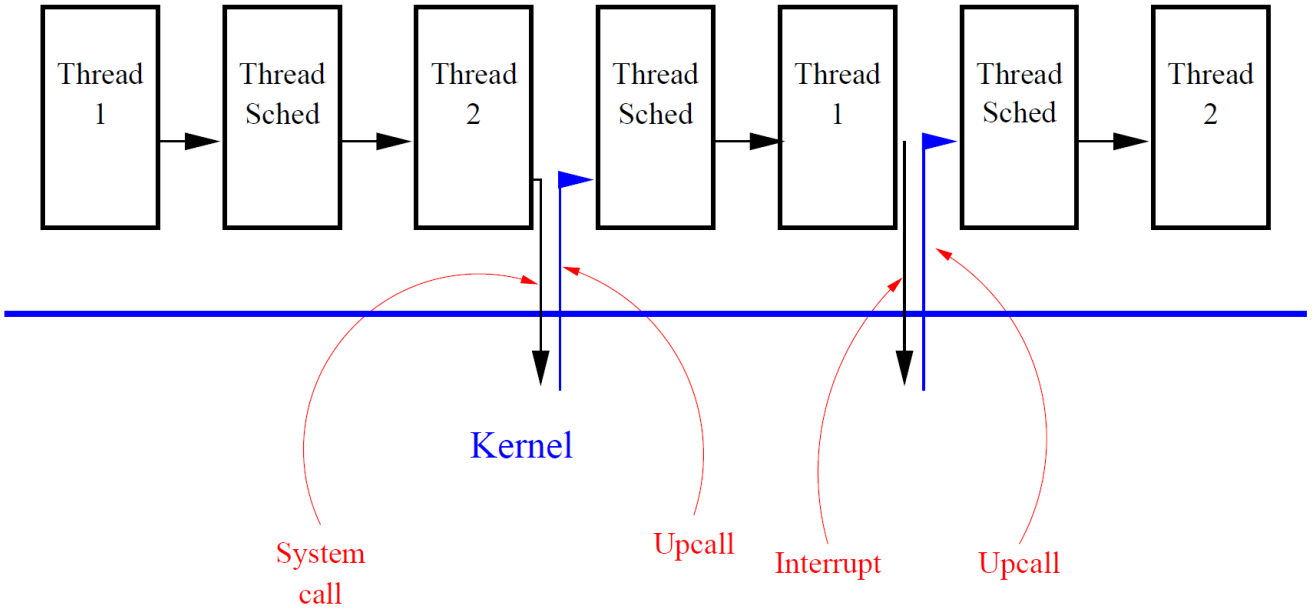
THOMAS E. ANDERSON, BRIAN N. BERSHAD, EDWARD D. LAZOWSKA, and HENRY M. LEVY
University of Washington

Threads are the vehicle for concurrency in many approaches to parallel programming. Threads can be supported either by the operating system kernel or by user-level library code in the application address space, but neither approach has been fully satisfactory.

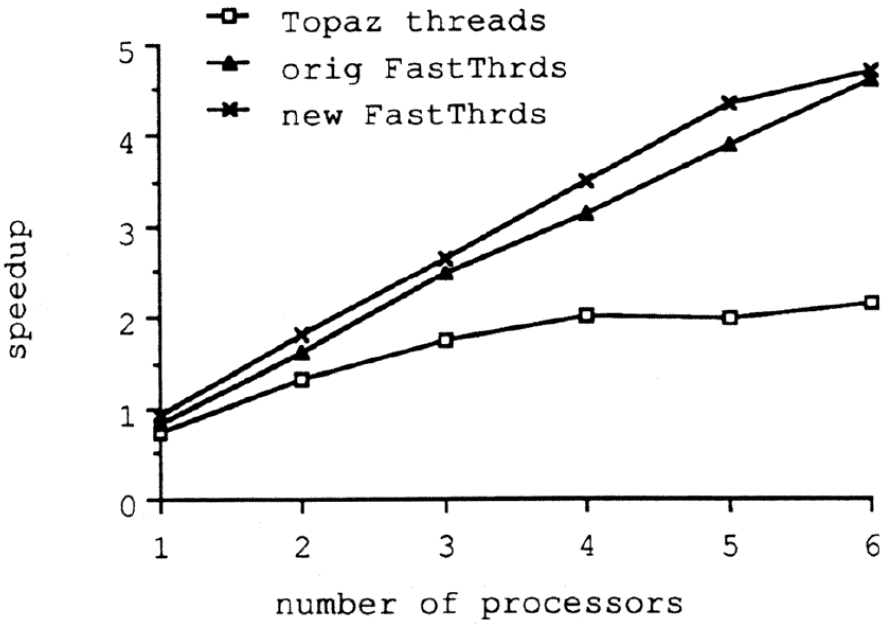
This paper addresses this dilemma. First, we argue that the performance of kernel threads is *inherently* worse than that of user-level threads, rather than this being an artifact of existing implementations; managing parallelism at the user level is essential to high-performance parallel computing. Next, we argue that the problems encountered in integrating user-level threads with other system services is a consequence of the lack of kernel support for user-level threads provided by contemporary multiprocessor operating systems; kernel threads are the *wrong abstraction* on which to support user-level management of parallelism. Finally, we describe the design, implementation, and performance of a new kernel interface and user-level thread package that together provide the same functionality as kernel threads without compromising the performance and flexibility advantages of user-level management of parallelism.

Scheduler Activations

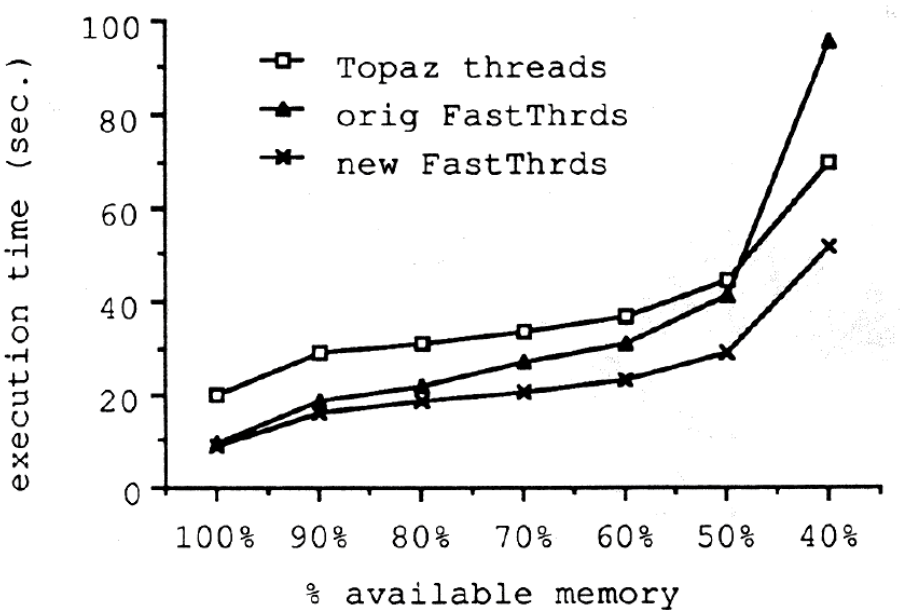
- Thread creation and scheduling is done at user level
  - When a system call from a thread blocks, the kernel does an upcall to the thread manager
  - The thread manager marks that thread as blocked, and starts running another thread
  - When a kernel interrupt occurs that's relevant to the thread, another upcall is done to unblock it



Results



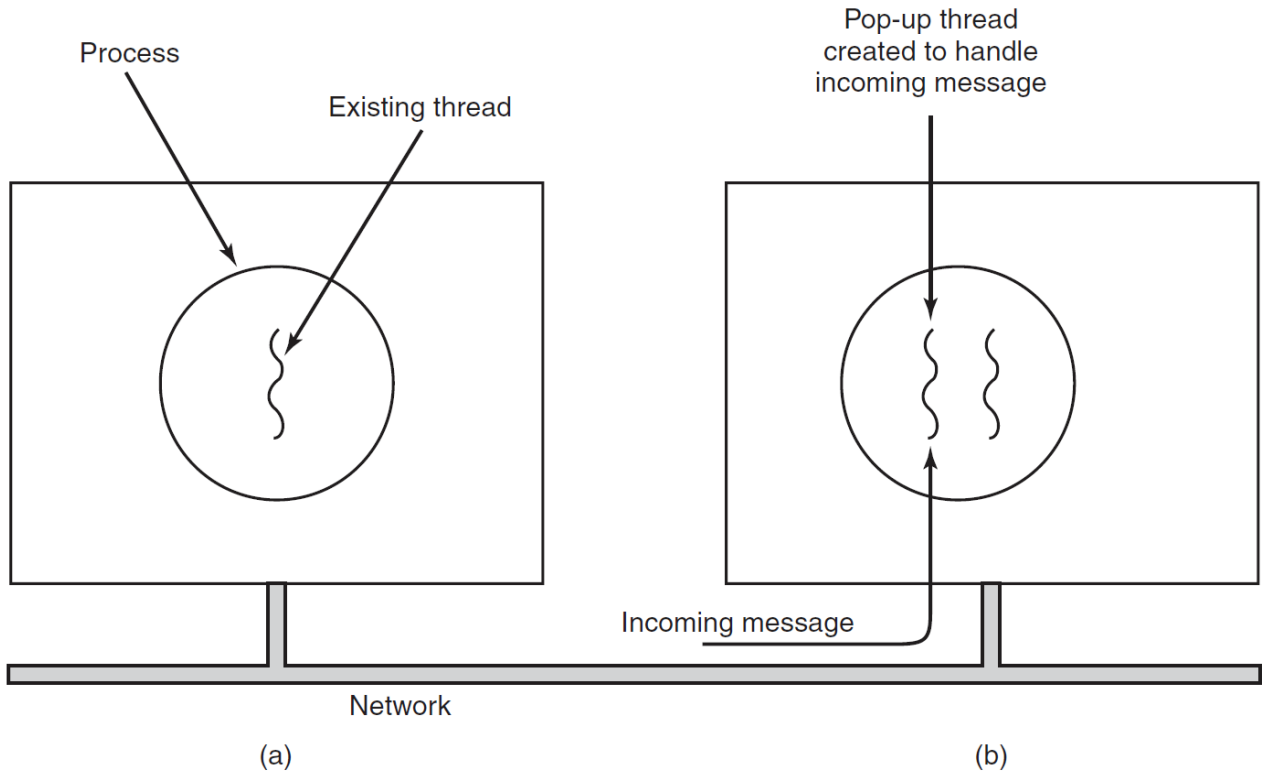
Speedup of N-Body application versus number of processors, 100% of memory available.



Execution time of N-Body application versus amount of available memory, 6 processors.

Pop-up Threads

- When a message arrives, the kernel creates a new thread
- Sometimes, the thread can run entirely in the kernel
- Messy and delicate — what process should own the thread, what sorts of events result in thread creation, etc.



Summary

- Threads
  - What is a thread
  - Kernel-level and user-level threads
  - Implementing threads

- Next lecture: Memory Management