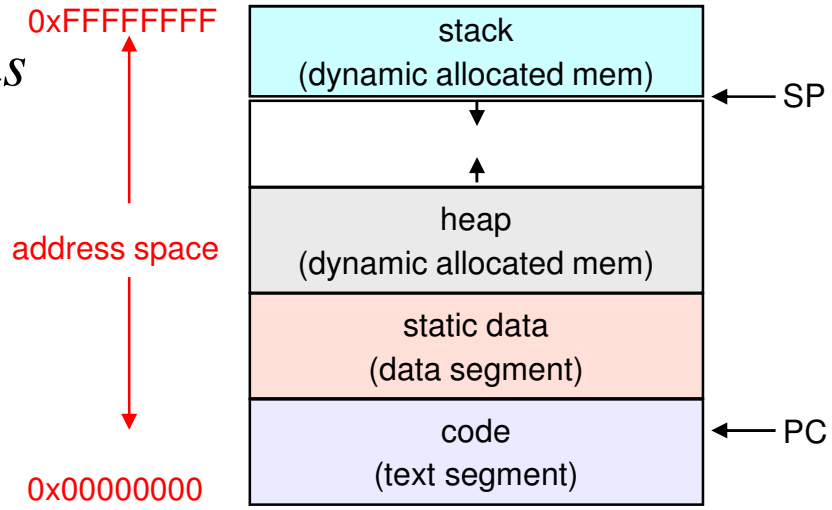
Introduction to Operating Systems

- Introduction
- Processes and Threads
- Memory Management
- File Systems
- Input / Output
- Deadlocks

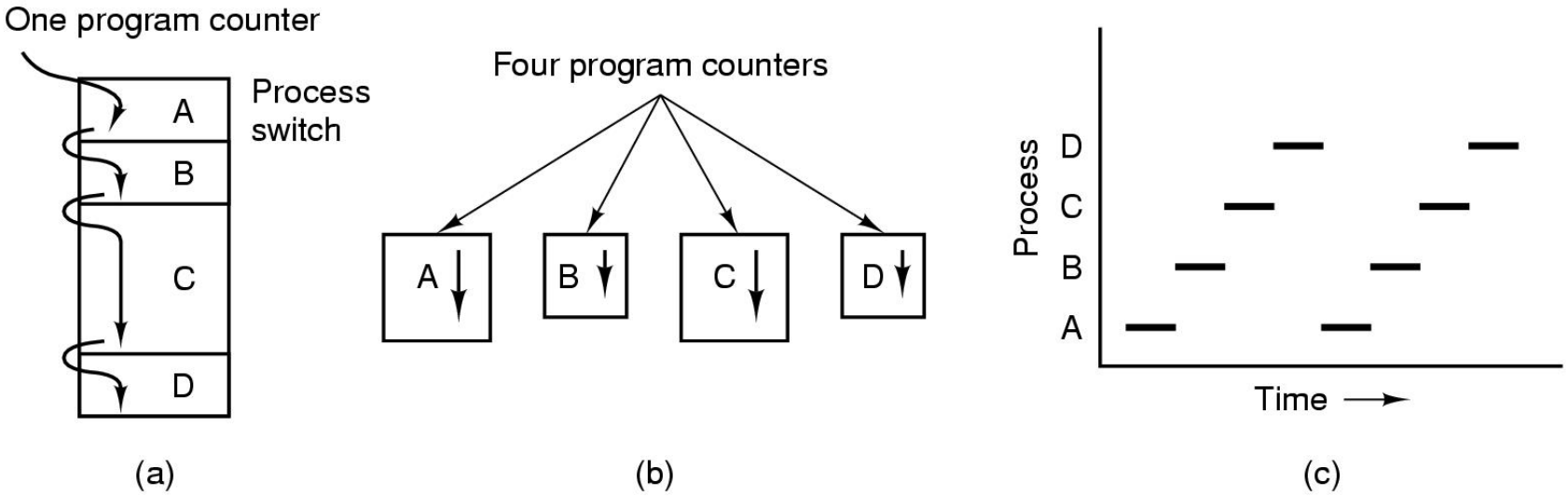
Processes

- The processes are
  - *an abstraction of a running program*
  - *is often called a job or task*
  - *a program in execution; process execution must progress in sequential fashion.*
- A process includes
  - *codes for the running programs*
  - *program counter (PC)*
  - *data for the running program*
  - *Stack for procedure calls.*



Processes

- Process Model



(a) Multiprogramming of four programs

(b) Conceptual model of 4 independent, sequential processes

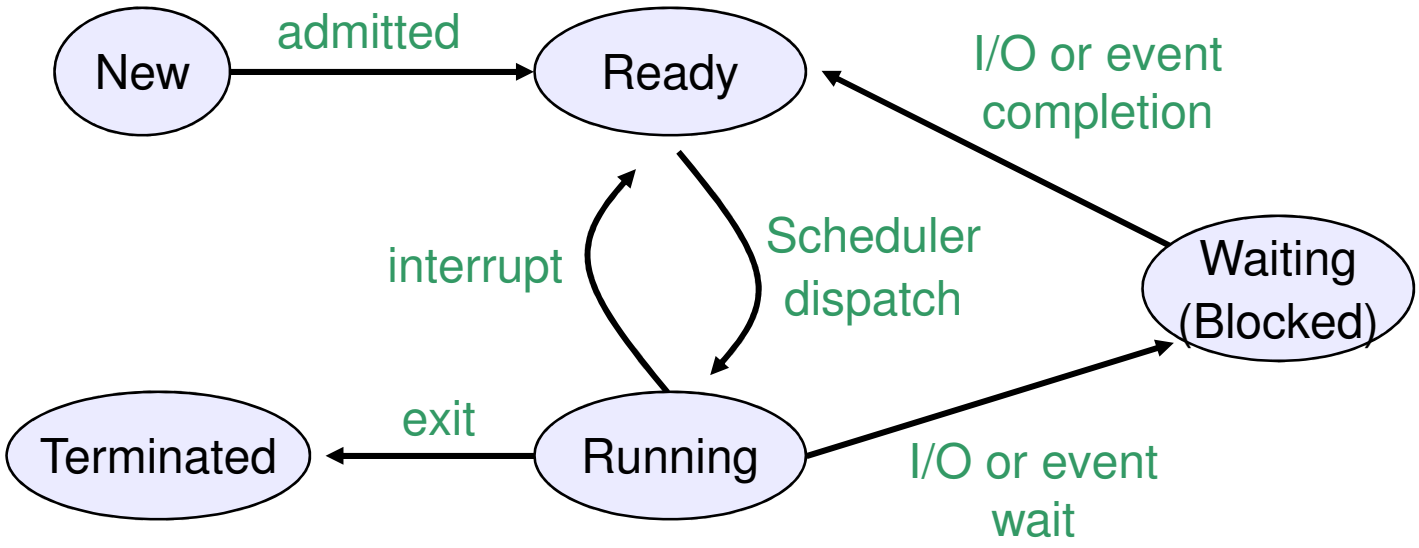
(c) Only one program active at any instant

Process States

- Each process has an execution state which indicates what it is currently doing
  - *ready: waiting to be assigned to CPU*
    - *could run, but another process has the CPU*
  - *running: executing on the CPU*
    - *is the process that currently uses the CPU*
  - *blocked (waiting): waiting for an event, e.g. I/O*
    - *cannot make progress until event happens*

- As a process executes, it moves from state to state
  - *Windows: Run the task manager and look up the state of all active processes*
  - *Linux/Unix: type "ps –aux" to see all processes in the system or run system monitor*

Process state transition



Implementation of process

- How does the OS represent a process in the kernel?
  - *at any time, there are many processes, each in its own particular state*
  - *the OS maintains a table, called the process control block (PCB), with one entry per process.*

- PCB contains all info about the process
  - *OS keeps all of a process' hardware execution state in the PCB when the process isn't running*
    - *PC*
    - *SP*
    - *Registers*
  - *when process is unscheduled, the state is transferred out of the hardware into the PCB*

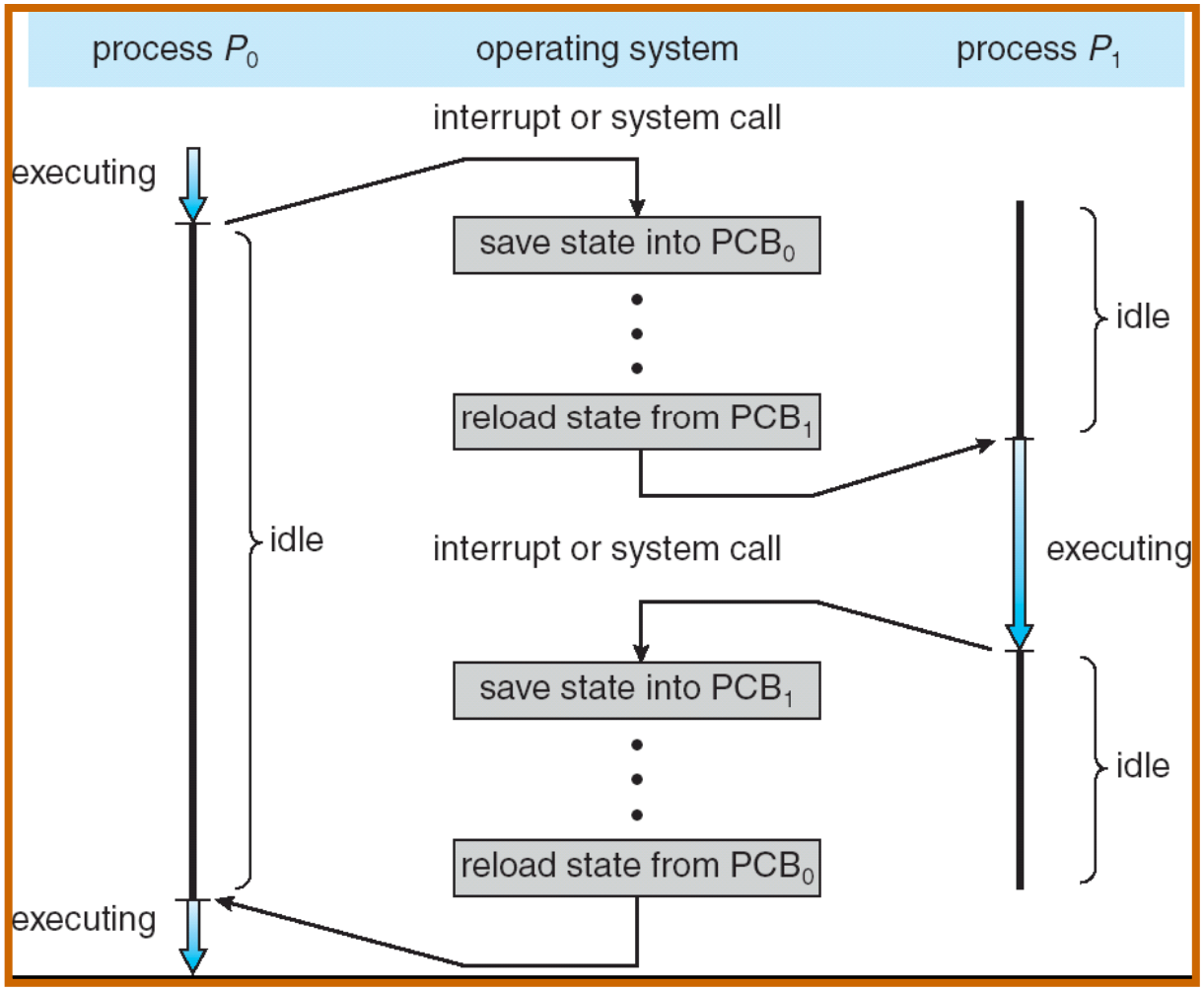
Implementation of process

Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

- Fields of a process table entry

•In Linux:
  - defined in task_struct (include/linux/sched.h)
  - over 95 fields!!!

Implementation of process



Implementation of process

- Context switch
  - *When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process*
  - *Context-switch time is overhead; the system does no useful work while switching*
  - *Time dependent on hardware support*

Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Resource sharing
  - *Parent and children share all resources*
  - *Children share subset of parent's resources*
  - *Parent and child share no resources*
- Execution
  - *Parent and children execute concurrently*
  - *Parent waits until children terminate*

Process Creation

- Address space
  - *Child duplicate of parent*
  - *Child has a program loaded into it*
- what creates the first process, and when?
  - *In Unix, the first process is called "init", which creates a login process per terminal. When a user logs in, the login process creates a shell process, which is used to enter user commands*

Unix Process Creation

- UNIX process creation through fork() system call
  - *creates and initializes a new PCB*
  - *creates a new address space*
  - *initializes new address space with a copy of the entire contents of the address space of the parent*
  - *initializes kernel resources of new process with resources of parent (e.g. open files)*
  - *places new PCB on the ready queue*
- the fork() system call returns twice
  - *once into the parent, and once into the child*
  - *returns the child's PID to the parent*
  - *returns 0 to the child*

Fork()

```
int main(int argc, char **argv)
{
        char *name = argv[0];
        int child_pid = fork();

        if (child_pid == 0) {
                printf("Child of %s is %d\n", name, child_pid);
                return 0;
        } else {
                printf("My child is %d\n", child_pid);
                return 0;
        } //end-else

} //end-main
```

Output

```
% gcc -o testparent testparent.c
% ./testparent
My child is 486
Child of testparent is 0

% ./testparent
Child of testparent is 0
My child is 488
```

Fork & Exec

- So how do we start a new program, instead of just forking the old program?
  - *the exec() system call!*
  - *int execv(const char *path, char *const argv[]), or execl, execlp, execle, execvp.*

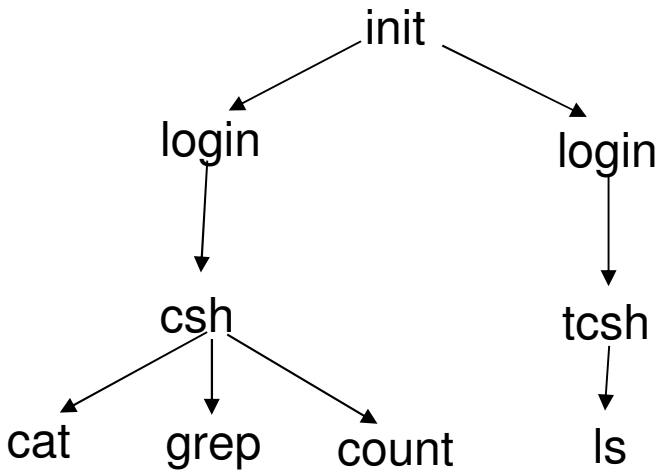
- exec()
  - *stops the current process*
  - *loads program 'path' into the address space*
  - *initializes hardware context, args for new program*
  - *places PCB onto ready queue*
  - *note: does not create a new process!*

Unix Shells

```
int main(int argc, char **argv)
{
    while (1) {
        char *cmd = get_next_command();
        int child_pid = fork();
        if (child_pid == 0) {
            manipulate STDIN/STDOUT/STDERR fd's
            exec(cmd);
            panic("exec failed!");
        } else {
            wait(child_pid);
        }
    }
}
```

Process Hierarchies

- In some systems, when a process creates another process, the parent process and the child process are associated in some ways
  - *A process and its descendants form a process group*
  - *When a signal is delivered (ctrl^c) to kill a process, the signal is delivered to all processes in the group. Each process can catch the signal, ignore the signal, or take default action, which is to be killed by the signal.*



Unix/Linux Process Hierarchy

Process termination

- A process will terminate, usually due to one of the following conditions:
  - *Normal exit (voluntary)*
    - *The process have done their work. e.g. Process calls a "exit" system call.*
  - *Error exit (voluntary)*
    - *The process discovers a fatal error. For example, a user types the command to compile a program and no such file exists.*
  - *Fatal error (involuntary)*
    - *The process causes an error, often due to a program bug. For example, executing an illegal instruction, referreng nonexistent memory, or dividing by zero.*
  - *Killed by another process (involuntary)*
    - *A process executes a system call telling the OS to kill some other process.In unix, this call is kill.*

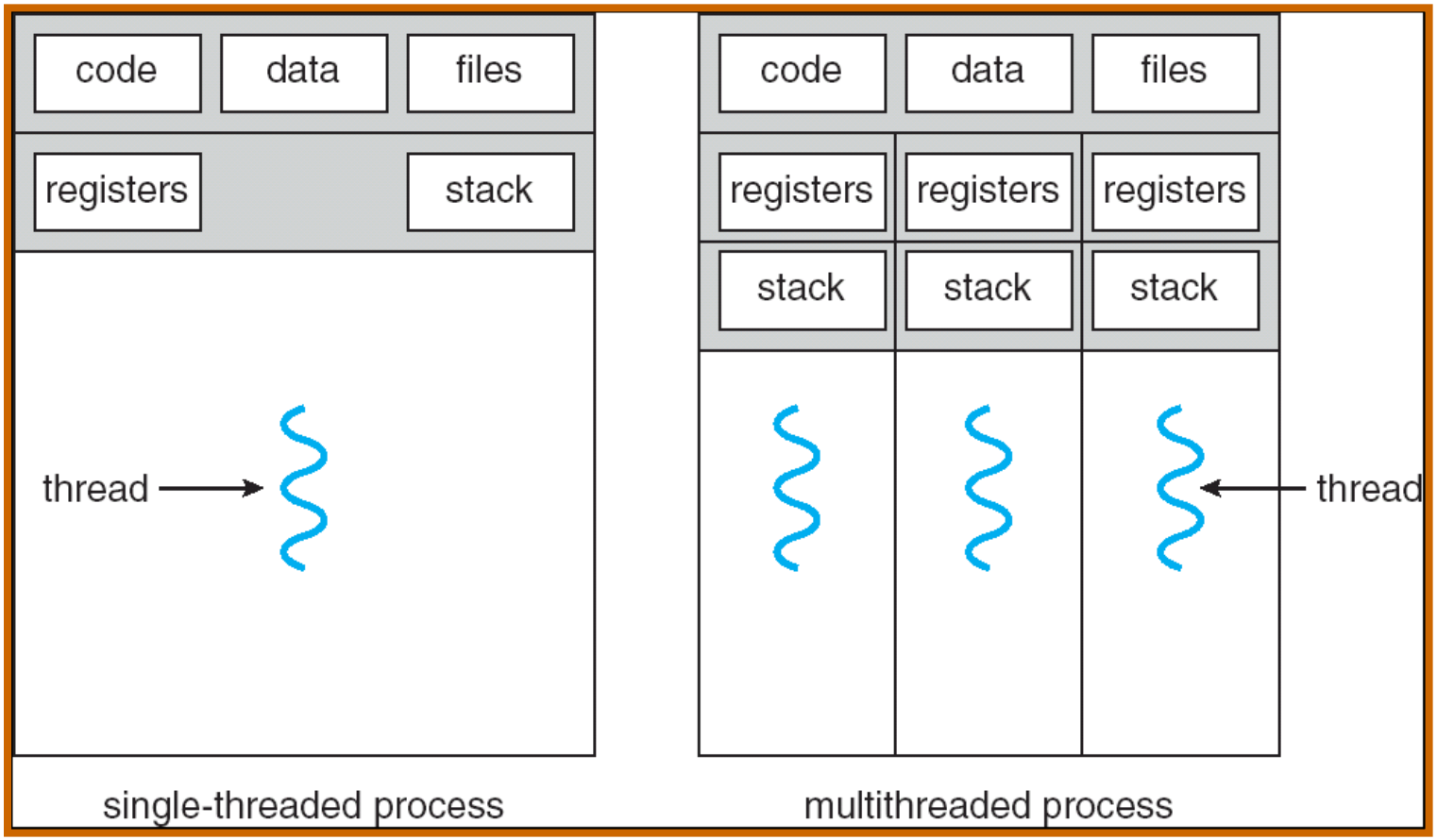
Threads

- [process] Each process has an address space and a single thread of control, that almost the definition of a process.
- [thread] Nevertheless, to multiple threads of control in the same address space running in quasi-parallel, as though they were almost separate processes (except for the shared address space).
- Why would anyone want to have a kind of process within a process?
  - *Easier programming model:* *Instead of thinking about interrupts, timers, and context switches, thinking about parallel process.*
  - *Lighter weight than processes:* *easier (faster) to create and destroy than processes.*

Threads

- *Performance argument:* *yield no performance gain when all of them CPU bound, but when there is substantial computing and also substantial I/O, having threads allows these activities to overlap, thus speeding up the application.*
- *Real parallelism:* *with multiple CPUs, real parallelism is possible.*

Single and Multithreaded Processes

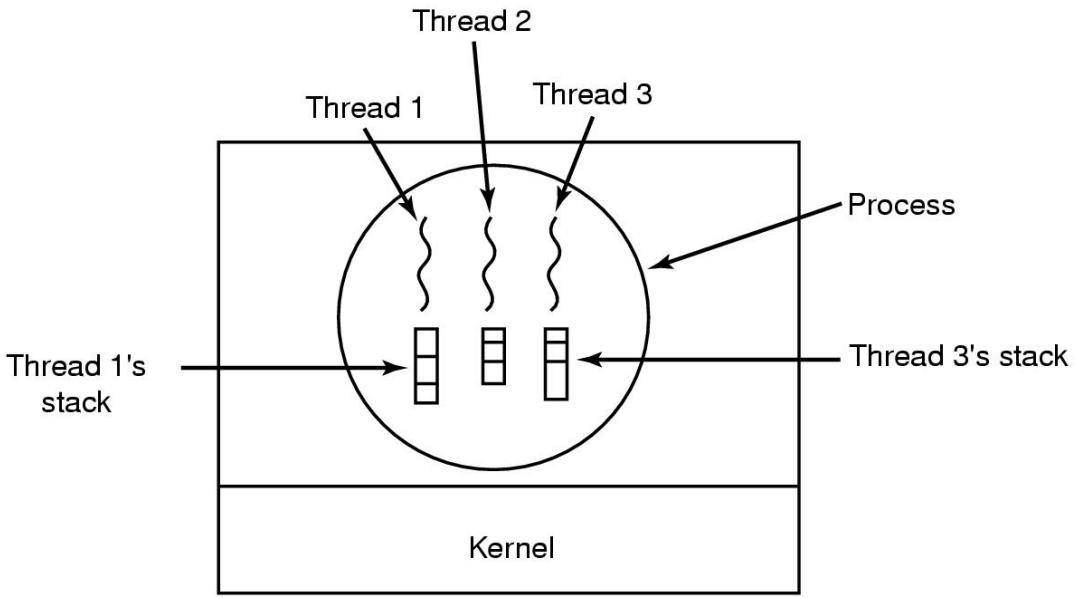


The Thread Model

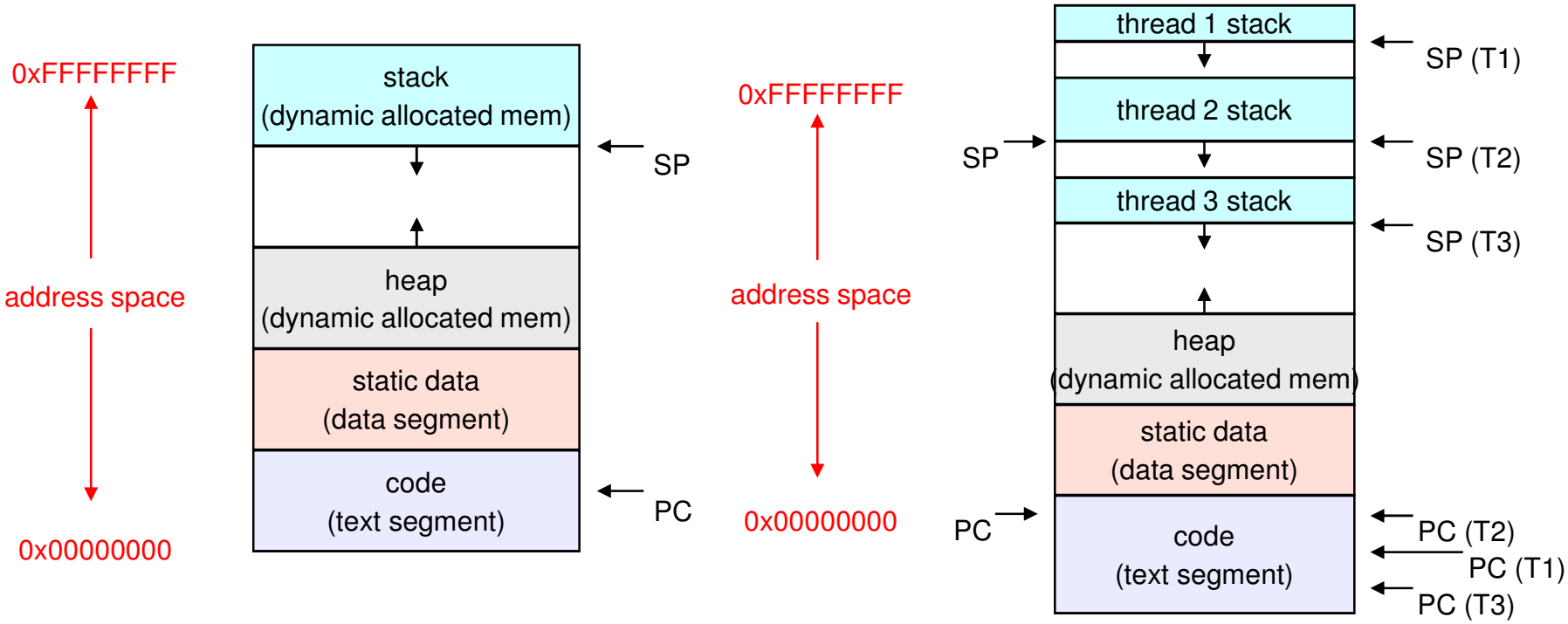
Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

(a) (b)

- (a) Items shared by all threads in a process
- (b) Items private to each thread

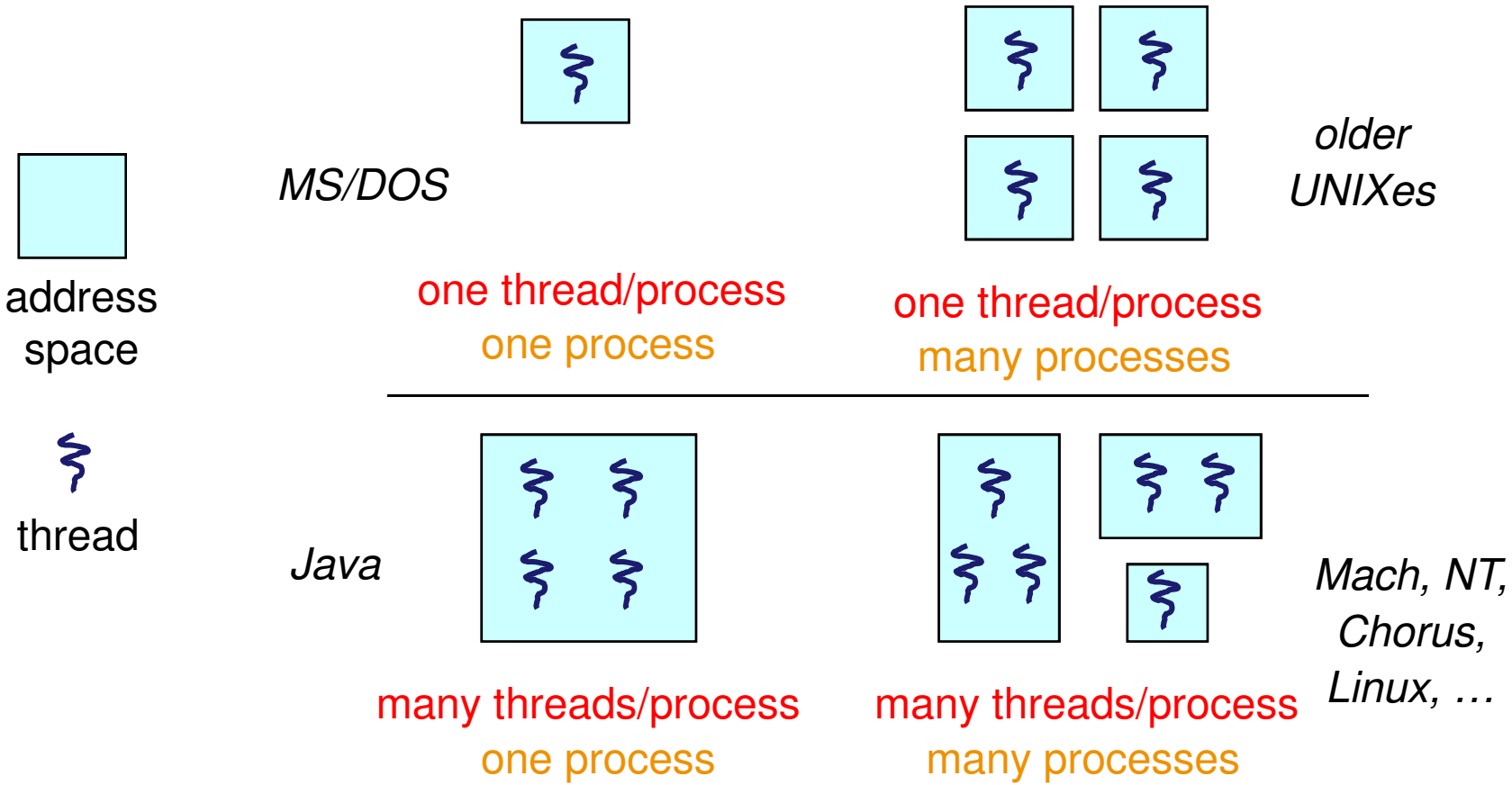


Process with and without threads

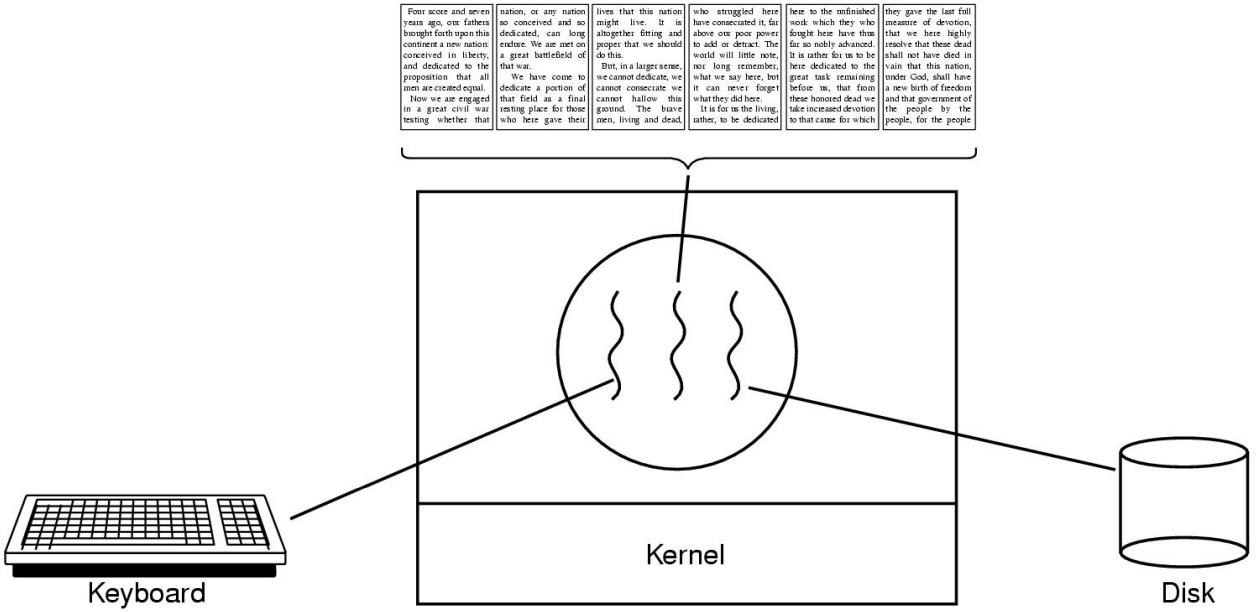


- Recall a process's address space
- A process's address space with threads

Thread Design Space



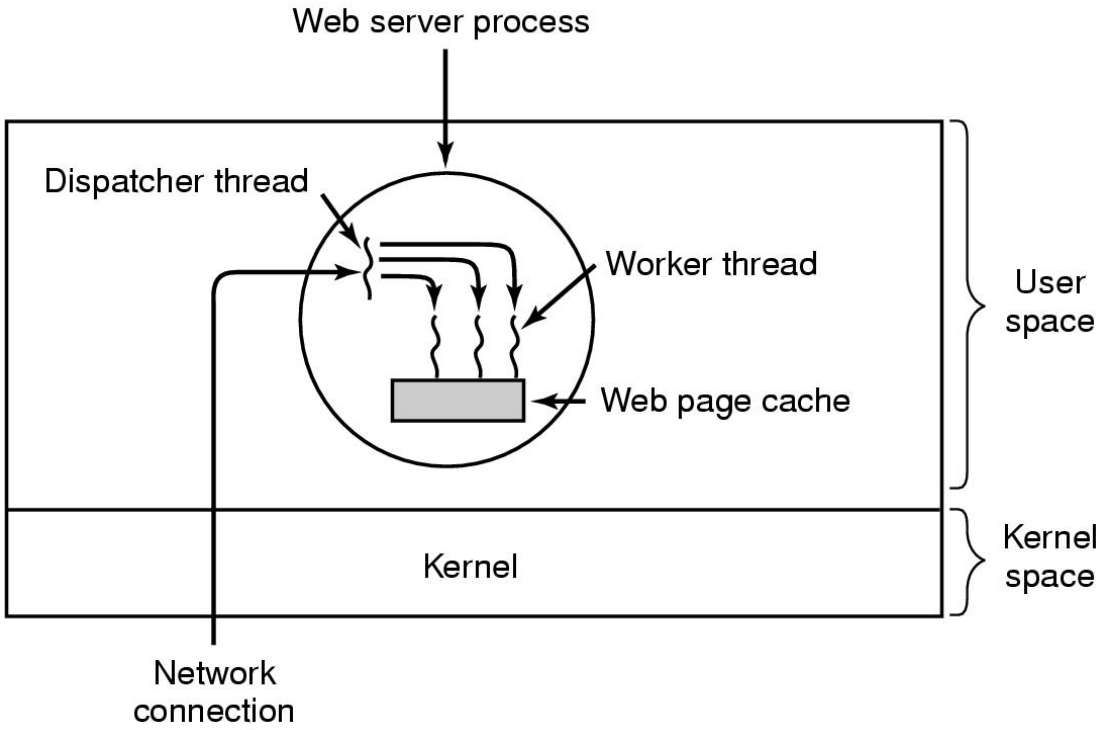
Thread Usage



- A word processor with three threads

- One thread interacts with the user
- The second thread reformats the entire pages.
- The third thread can handle the disk backup.

Thread Usage



- A multithreaded Web server

- One thread, the Dispatcher, reads incoming requests for work from the network. Then, it chooses an idle (blocked) worker thread and hands it the request.
- The second thread, worker thread, it checks to see if the request can be satisfied from cache. If not, it starts a read operation from disk to get the page

Thread Usage

```
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
  wait_for_work(&buf)
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page)
      read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

(b)

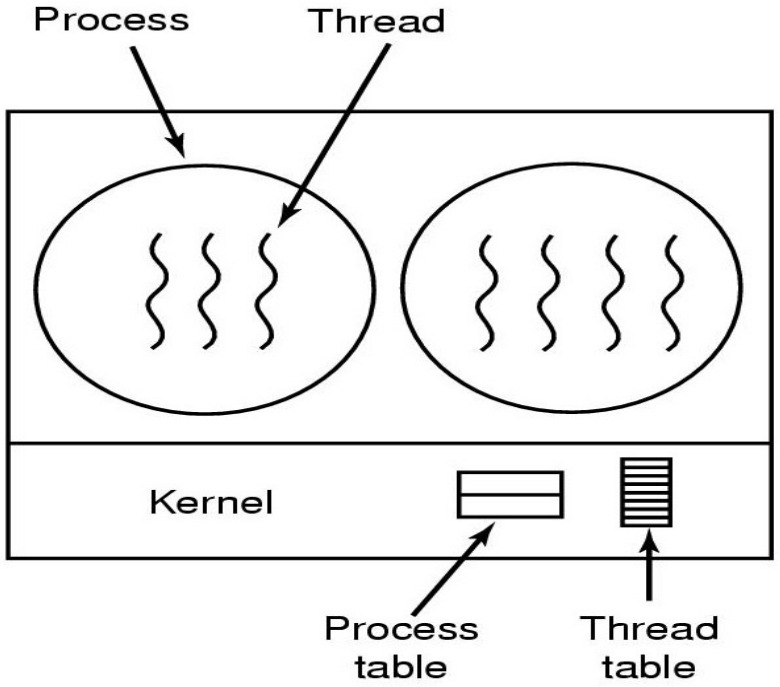
- Rough outline of code for previous slide
  - *(a) Dispatcher thread*
  - *(b) Worker thread*

Kernel Thread and User-Level Threads

- Who is responsible for creating/managing threads?
- Two answers, in general:
  - *the OS (kernel threads)*
    - *thread creation and management requires system calls*
  - *the user-level process (user-level threads)*
    - *a library linked into the program manages the threads*

Kernel Threads

- OS now manages threads *and* processes
  - *all thread operations are implemented in the kernel*
  - *OS schedules all of the threads in a system*
    - *if one thread in a process blocks (e.g. on I/O), the OS knows about it, and can run other threads from that process*
    - *possible to overlap I/O and computation inside a process*



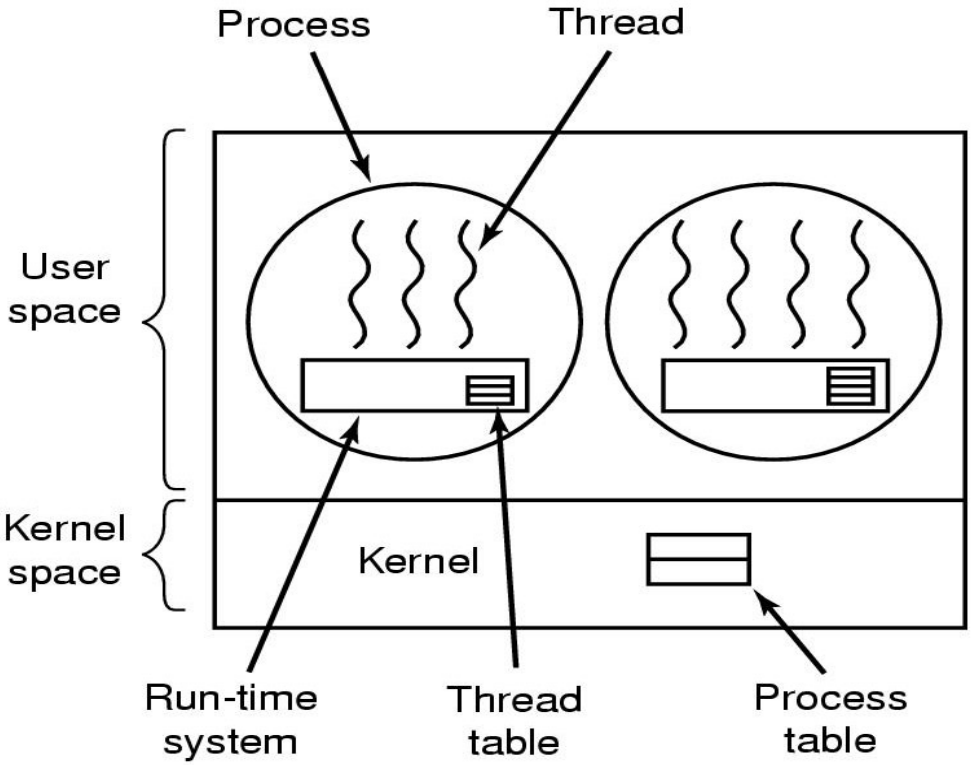
Kernel Threads

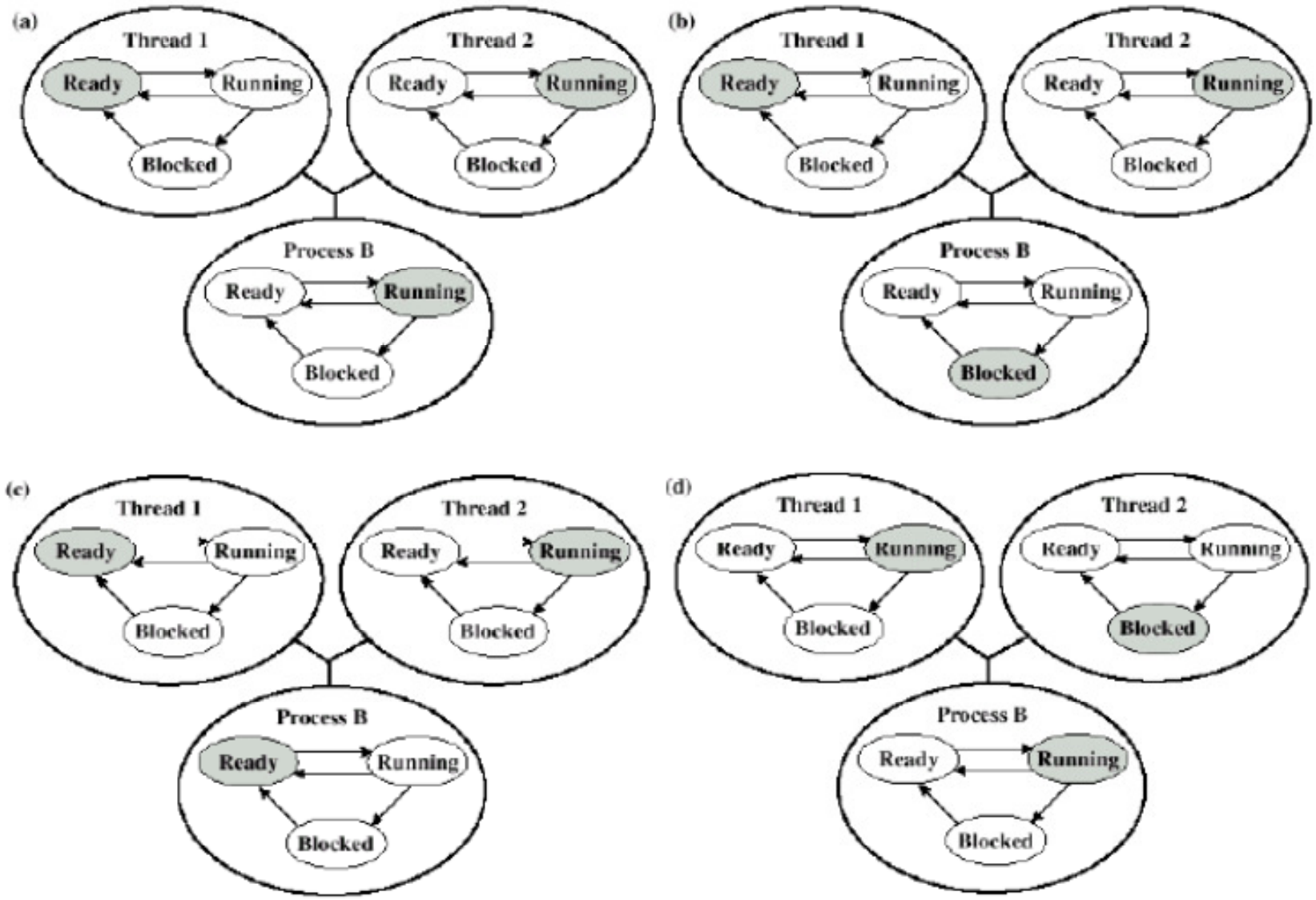
- Kernel threads are cheaper than processes
  - *less state to allocate and initialize*

- But, they can still be too expensive
  - *thread operations are all system calls*
  - *OS must keep state for each thread*
  - *Created by "clone" system call in Linux*
    - *Similar to fork, but threads share the same address space*

User-Level Threads

- To make threads cheap and fast, they need to be implemented at the user level
  - *managed entirely by user-level library, e.g. libpthreads.a*
  - *Each process keeps track of its own threads in a thread table*





- **Examples of the relationship between User-level thread states and process states**

User-Level Threads

- Why is user-level thread management possible?
  - *threads share the same address space*
  - *therefore the thread manager doesn't need to manipulate address spaces*
  - *threads only differ in hardware contexts (roughly)*
    - *PC, SP, registers*
    - *these can be manipulated by the user-level process itself!*

User-Level Threads

- User-level threads are small and fast
  - *each thread is represented simply by a PC, registers, a stack, and a small thread control block (TBC)*
  - *creating a thread, switching between threads, and synchronizing threads are done via procedure calls*
    - *no kernel involvement is necessary!*
  - *user-level thread operations can be 10-100x faster than kernel threads as a result*

Performance Example

- On a 700MHz Pentium running Linux 2.2.16:
  - *Processes*
    - *fork/exit: 251 ms*
  - *Kernel threads*
    - *pthread_create()/pthread_join(): 94 ms*
  - *User-level threads*
    - *pthread_create()/pthread_join: 4.5 ms*

User-Level Thread Limitations

- But, user-level threads aren't perfect
  - *tradeoff, as with everything else*
- User-level threads are invisible to the OS
  - *there is no integration with the OS*
- As a result, the OS can make poor decisions
  - *blocking a process whose thread initiated I/O, even though the process has other threads that are ready to run*

Thread Context Switch

- Very simple:
  - *save context of currently running thread*
    - *push machine state onto thread stack*
  - *restore context of the next thread*
    - *pop machine state from next thread's stack*
  - *return to caller as the new thread*
    - *execution resumes at PC of next thread*

- This is all done by assembly language

Thread Interface

- This is taken from the POSIX pthreads API:
  - *t = pthread_create(thread pointer,attributes, start_procedure, start_procedure_arguments)*
    - *creates a new thread of control*
    - *new thread begins executing at start_procedure*
  - *pthread_exit()*
    - *terminates the calling thread*
  - *pthread_join(t)*
    - *waits for the named thread to terminate*

- Windows
  - *CreateThread to start a thread*
  - *WaitForSingleObject(handle) to wait for thread termination*