OPERATING SYSTEM

Process Management

Chapter 2

Processes and Threads

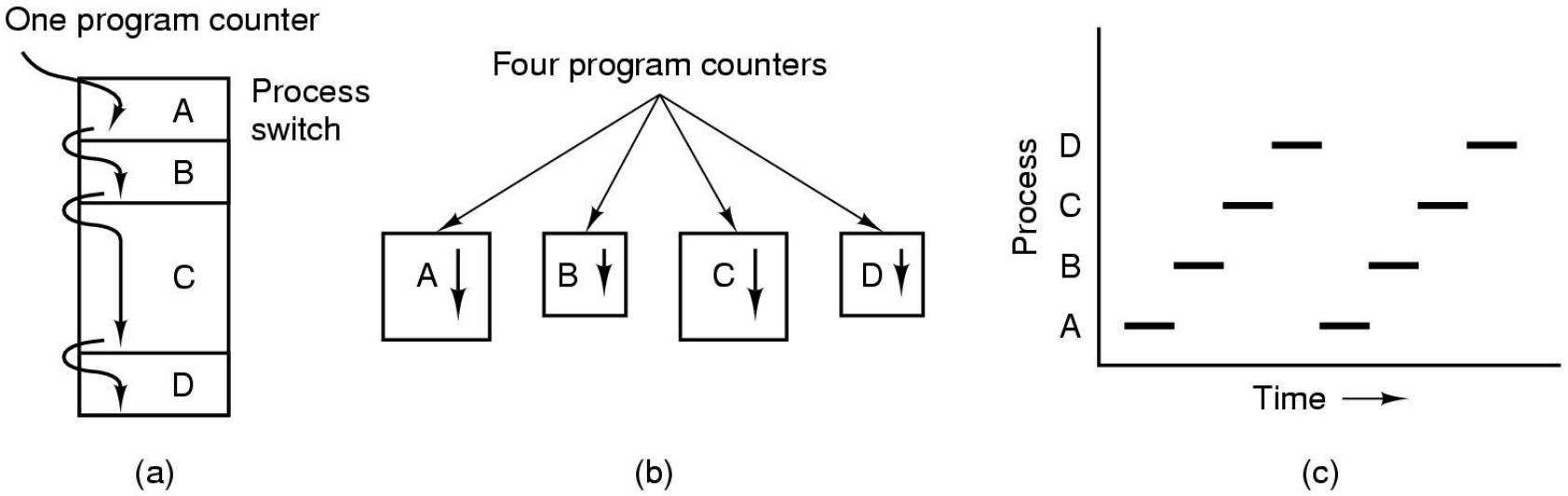
2.1 Processes
2.2 Threads

Processes

- it's the active instance of a program running on the system, along with all the resources the OS allocates to it.
- When you run a program, the OS creates a process that includes: Program Code (Text), Process ID (PID), Memory Space, Registers & Program Counter, Open Files & I/O Handles, Process State, Security Attributes
- In process model, all the runnable software on the computer, sometimes including the operating system, is organized into a number of sequential processes, or just processes for short.
-

Processes

The Process Model



- (a) Multiprogramming of four programs
- (b) Conceptual model of 4 independent, sequential processes
- © Only one program active at any instant

Process Creation

Principal events that cause process creation

1. System initialization

- Execution of a process creation system

1. User request to create a new process
2. Initiation of a batch job

Process Termination

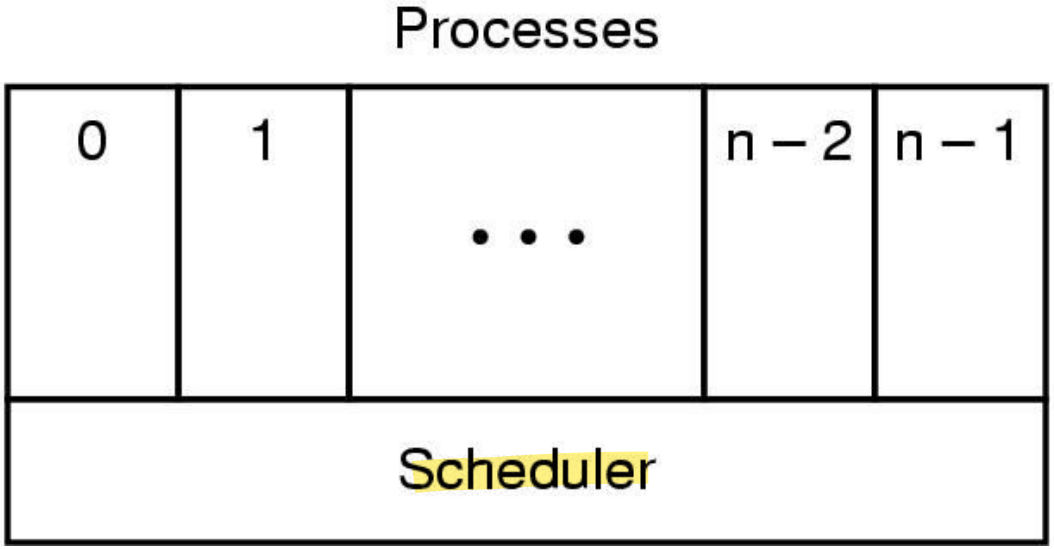
Conditions which terminate processes

1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (involuntary)
4. Killed by another process (involuntary)

Process Hierarchies

- Parent creates a child process, child processes can create its own process
- Forms a hierarchy
  - UNIX calls this a "process group"
- Windows has no concept of process hierarchy
  - all processes are created equal

Process States (2)

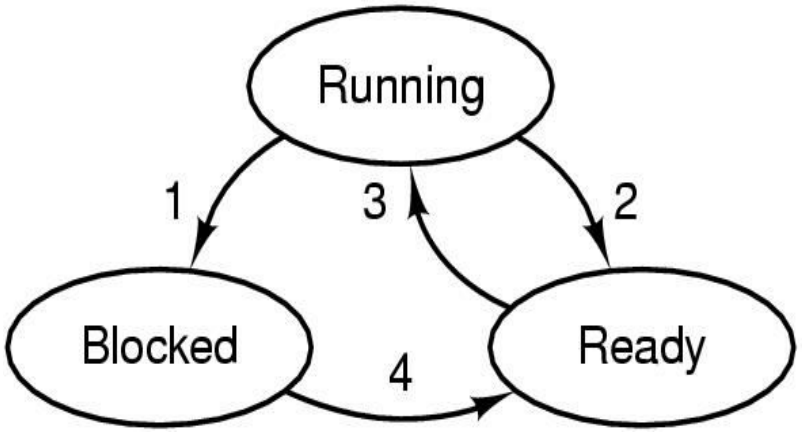


- Lowest layer of process-structured OS
  - handles interrupts, scheduling
- Above that layer are sequential processes

Process States (1)

- each process is an independent entity,
- often need to interact with other processes.
  - cat chapter1 chapter2 chapter3 | grep tree
- Possible process states
  - Running (actually using the CPU at that instant).
  - Ready (runnable; temporarily stopped to let another process run).
  - Blocked (unable to run until some external event happens).

Process States (1)



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Transition between states

Implementation of Processes (1)

- To implement the process model, the operating system maintains a table (an array of structures), called the process table, with one entry per process.
- Information about the process that must be saved when the process is switched from running to ready or blocked state so that it can be restarted later as if it had never been stopped.

Implementation of Processes (1)

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

Implementation of Processes (2)

Interrupt handling with one CPU

Interrupt: A signal sent by hardware (or software) to the CPU for the interrupt. Example: You press a key → keyboard sends an interrupt.

Interrupt Vector:

A table in memory that tells the CPU which ISR to call

ISR (Interrupt Service Routine):

A special function in the OS kernel that handles the specific interrupt

Implementation of Processes (2)

Every type of hardware interrupt (keyboard, disk, timer, etc.) has a predefined slot in a special table in memory (the interrupt vector table).

That slot stores the memory address of the OS function (ISR) that knows how to handle that specific device.

When the device interrupts the CPU, the CPU looks up the address in the table and jumps to the right handler—without needing to guess what caused the interrupt

Modeling multiprogramming

- Keep the CPU Busy, don't wait for I/O interrupt.
- Instead of running just one program, load several programs into memory at the same time.
- When one program is waiting for I/O, the CPU can switch to another program that's ready to run.

Imagine: Each program is waiting 80% of the time → so it's only using the CPU 20% of the time. If you load 5 programs, you might think:

20% × 5 = 100% → CPU is always busy!

But this is too optimistic, because all 5 programs might be waiting at the same time (e.g., all reading files).

Modeling multiprogramming

Let: p = fraction of time a process waits for I/O (e.g., p = 0.8 for 80% wait), and n = number of programs in memory

Then: Probability that all n programs are waiting = p^n

So, CPU is idle p^n of the time

Therefore, CPU utilization = $1 - p^n$

Modeling multiprogramming

Figure 2-6 shows, for different values of p (or "I/O wait"), the CPU utilization as a function of n, which is called the **degree of multiprogramming**.

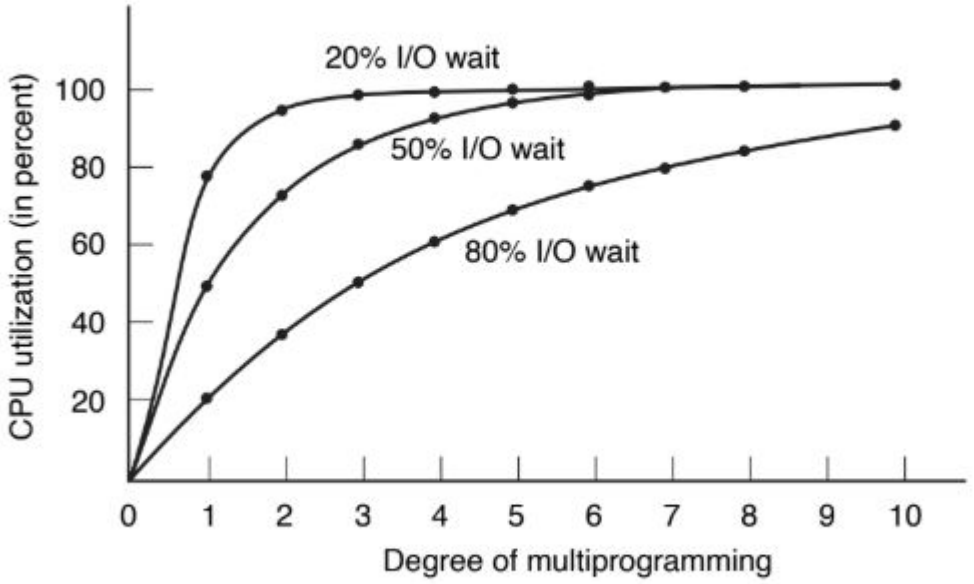


Figure 2-6. CPU utilization as a function of the number of processes in memory.

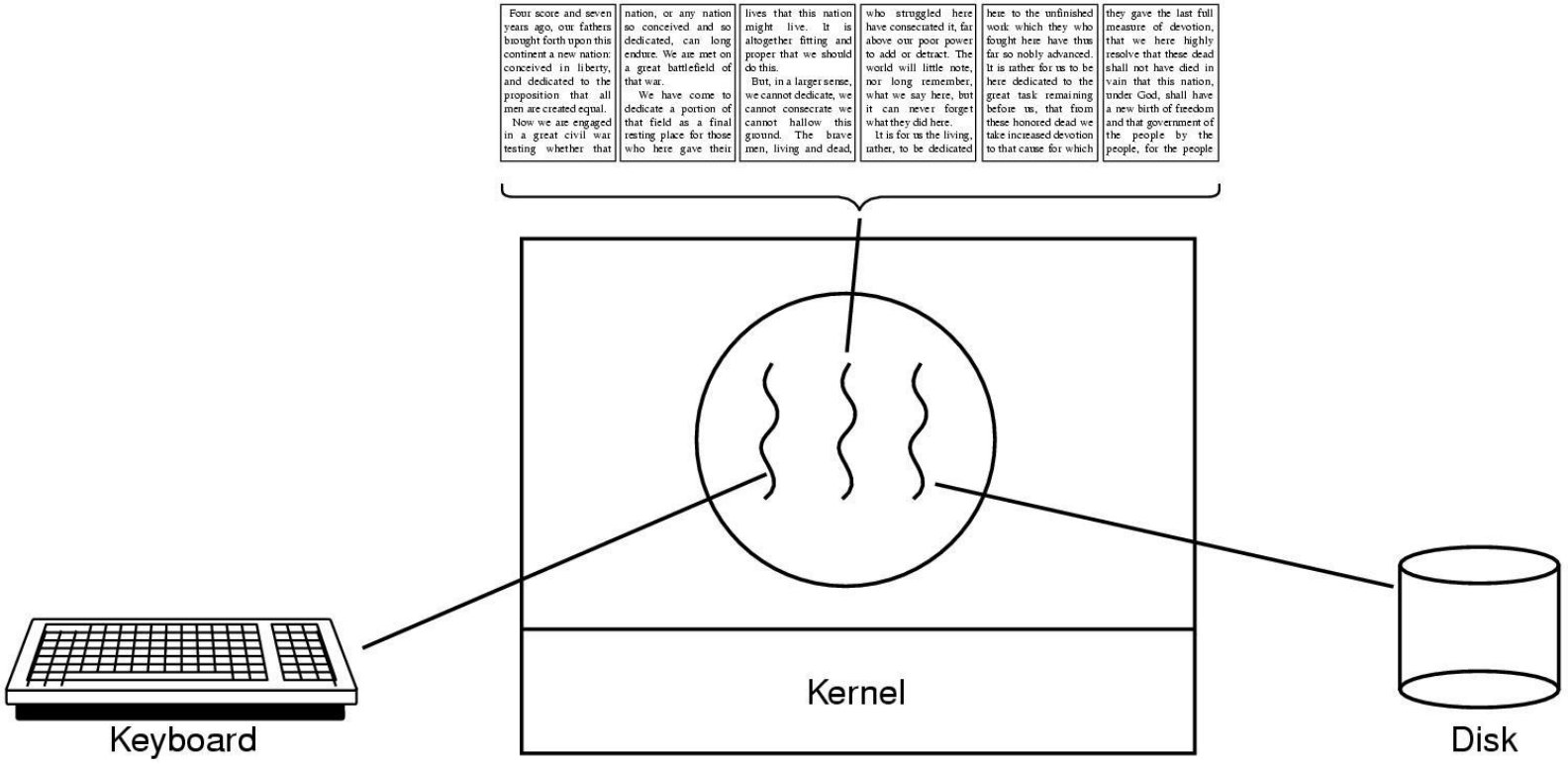
from the above figure, if processes spend 80% of their time waiting for I/O, at least 10 processes must be in memory at once to get the CPU waste below 10%.

Threads

The Thread Model (1)

- Kind of process within a process, or miniprocess
- In many applications, multiple activities are going on at once.

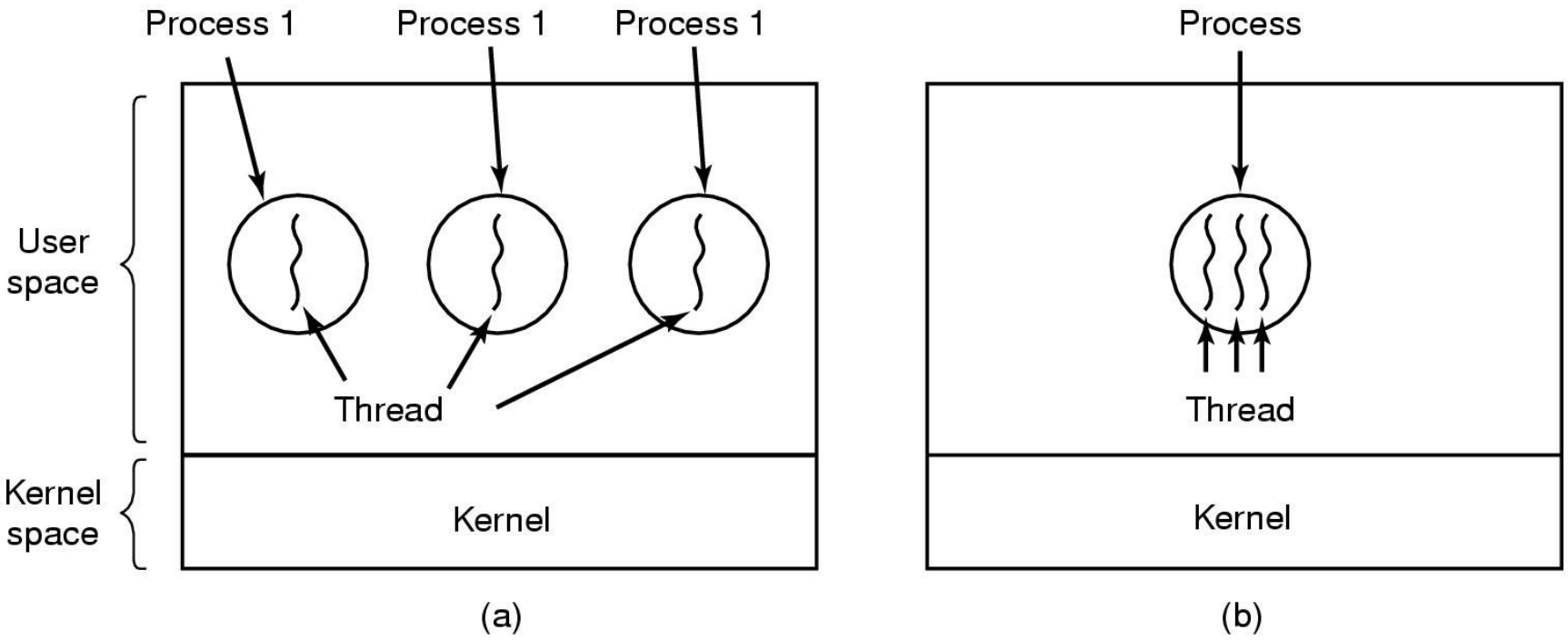
Thread Usage (1)



A word processor with three threads

Threads

The Thread Model (1)



(a) Three processes each with one thread

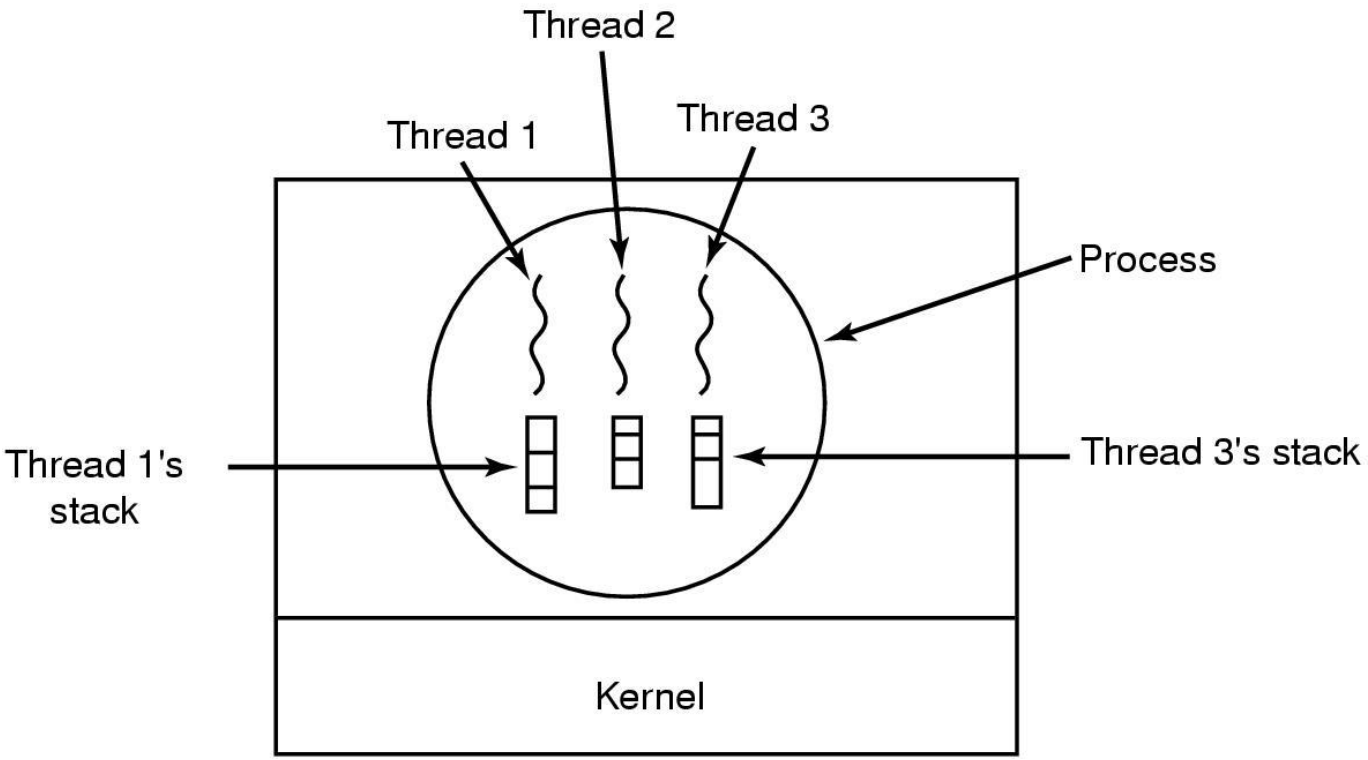
(b) One process with three threads

The Thread Model (2)

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	

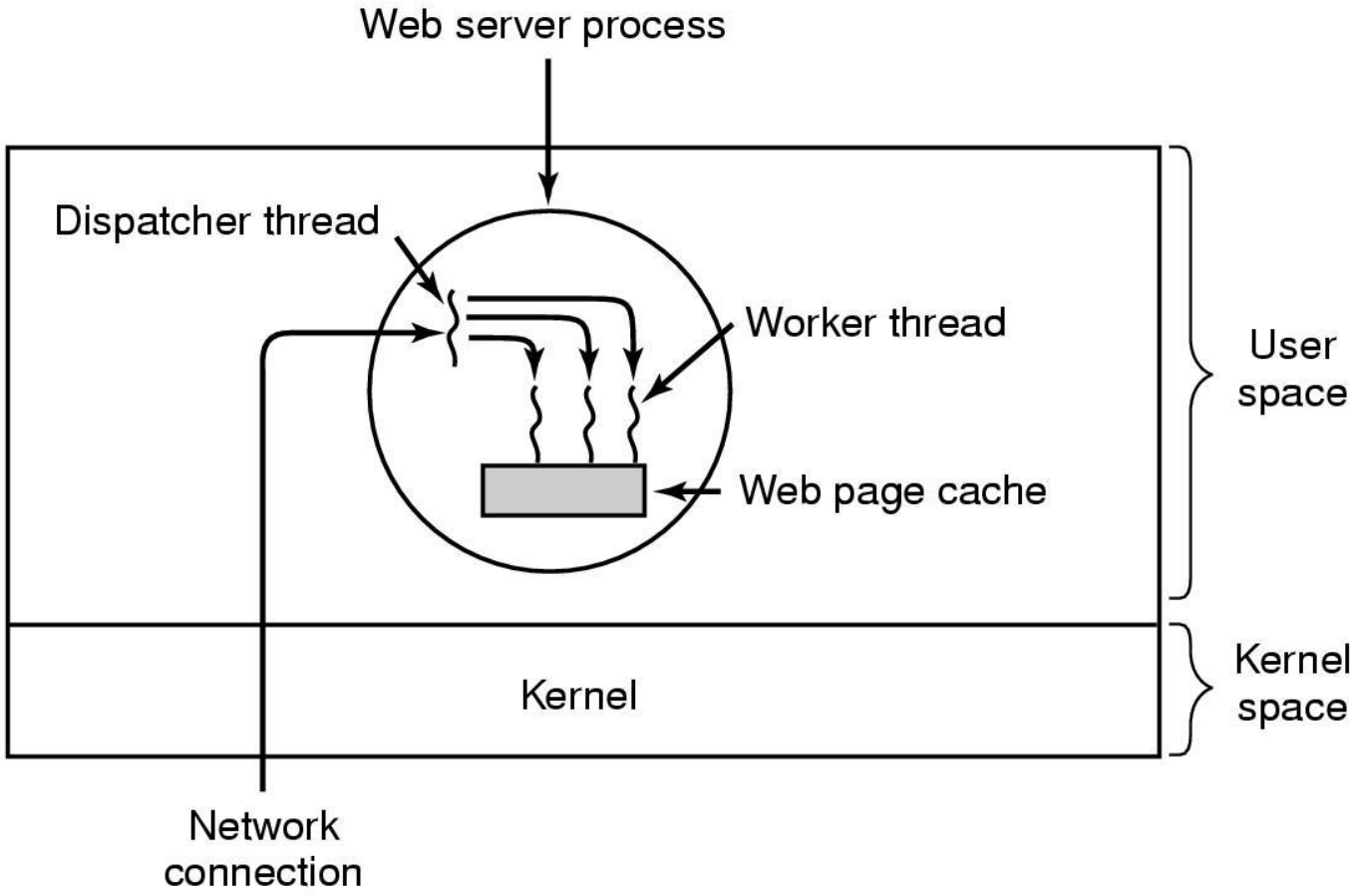
- Items shared by all threads in a process
- Items private to each thread

The Thread Model (3)



Each thread has its own stack

Thread Usage (2)



A multithreaded Web server

Thread Usage (3)

```
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, reads incoming requests for work from the network.

  (b) Worker thread, can have multiple worker thread

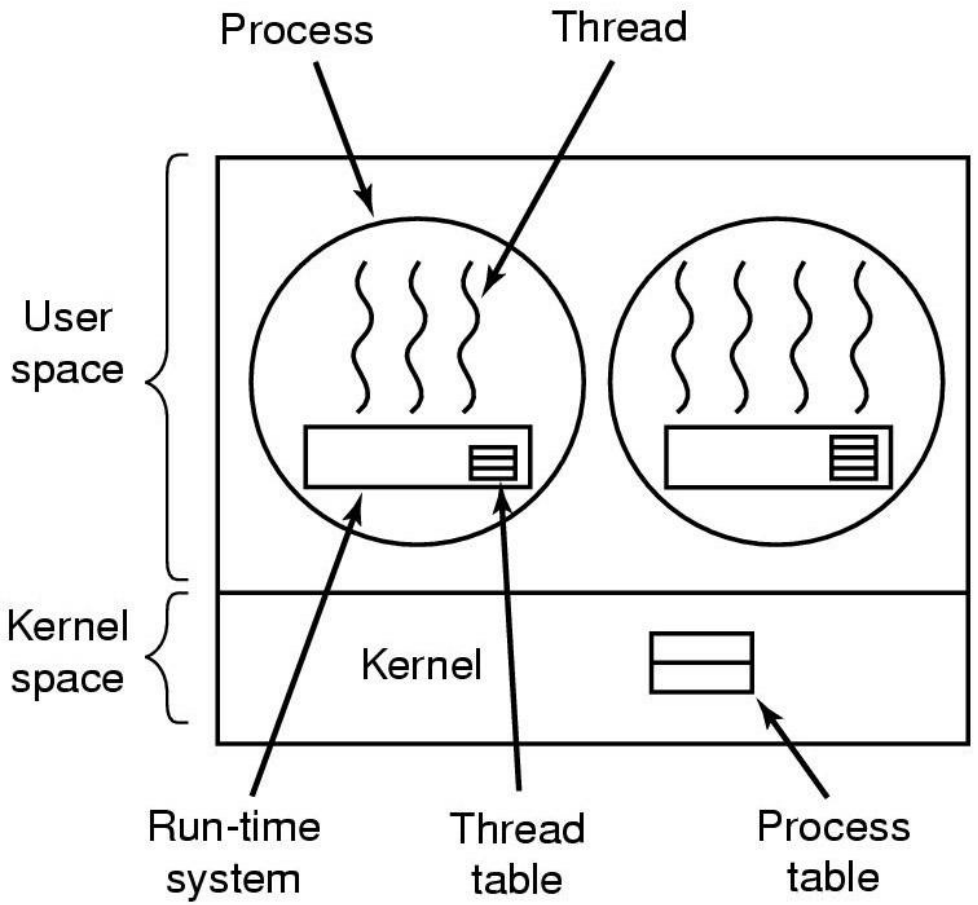
Implementing Threads

- User space
- kernel space
- hybrid

User Level Threads: Implementation

- put the threads package entirely in user space.
- The kernel knows nothing about them. As far as the kernel is concerned, it is managing ordinary, single-threaded processes.
- All thread management (creating, switching, scheduling) is done by your code or a user-level library—not the OS kernel.

Implementing Threads in User Space



A user-level threads package

User Level Threads:Advantage

- Can be implemented on an non-thread OS
- Thread switching can be implemented to be at least an order of magnitude faster than trapping to kernel.
  - Run time system procedures are used
- Similarly scheduling can be very fast because procedure called for scheduling are local , no context switch is needed, no memory cache need to be flushed.
- Allow each process to have its own customized scheduling algorithm.
- Scale better. Because kernel threads would require some table and stack space in kernel and if no. of threads increases… PROBLEMS

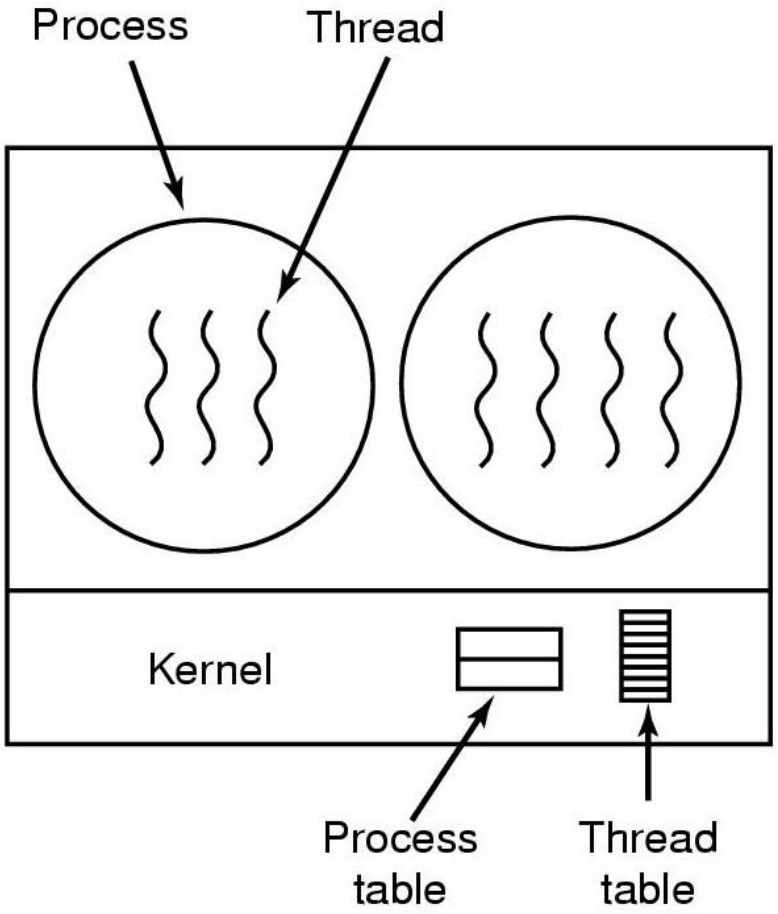
User Level Threads:Problems(1)

- Blocking system calls implementation issue
  - If a thread's call blocks the whole process will be blocked!!!
- During "page fault" the entire process is blocked since kernel only knows the process
  - Even though other threads might be runnable
- If a thread starts running , no other thread within the process will run unless the first thread voluntary give up the cpu
  - within a single process, there are no clock interrupt

User Level Threads:Problems(2)

- Most Devastating Arguments (Against):
  - Threads are needed for applications blocking often by system calls. Once a trap has occurred to kernel for the system call it is hardly more work for the kernel to switch threads

Implementing Threads in the Kernel

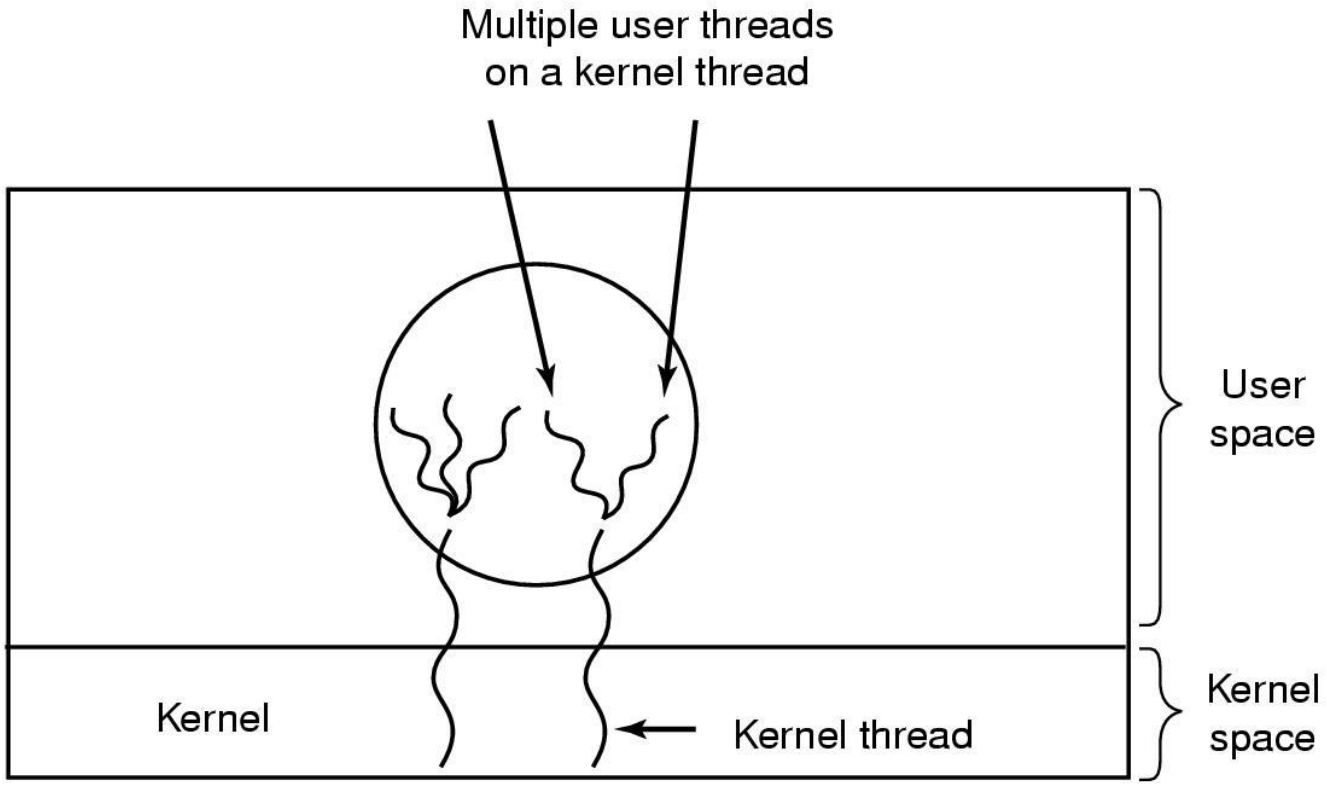


A threads package managed by the kernel

Kernel Threads

- Advantages
  - No need for non-blocking system calls
  - In case of a page fault since the kernel knows about the threads it can do thread (or if needed process) switch
- Disadvantages
  - Cost of system call is substantial as opposed to run time system procedure

Hybrid Implementations



Multiplexing user-level threads onto kernel-level threads

Hybrid Implementations

- Combining the advantages of the 2 methods
- Now the programmer can decide how many kernel threads to use and how many user-level threads to multiplex on each one.
- Gives the ultimate of flexibility
- Kernel is aware of only the kernel level threads and schedule those
- Each kernel level thread has some set of user level thread that take turns using it.

Hybrid implementation examples: Scheduler Activations

- Goal
  - mimic functionality of kernel threads
  - gain performance of user space threads
- Avoids unnecessary user/kernel transitions
  - If a thread blocks for another thread, why involve kernel?

Scheduler Activations(2)

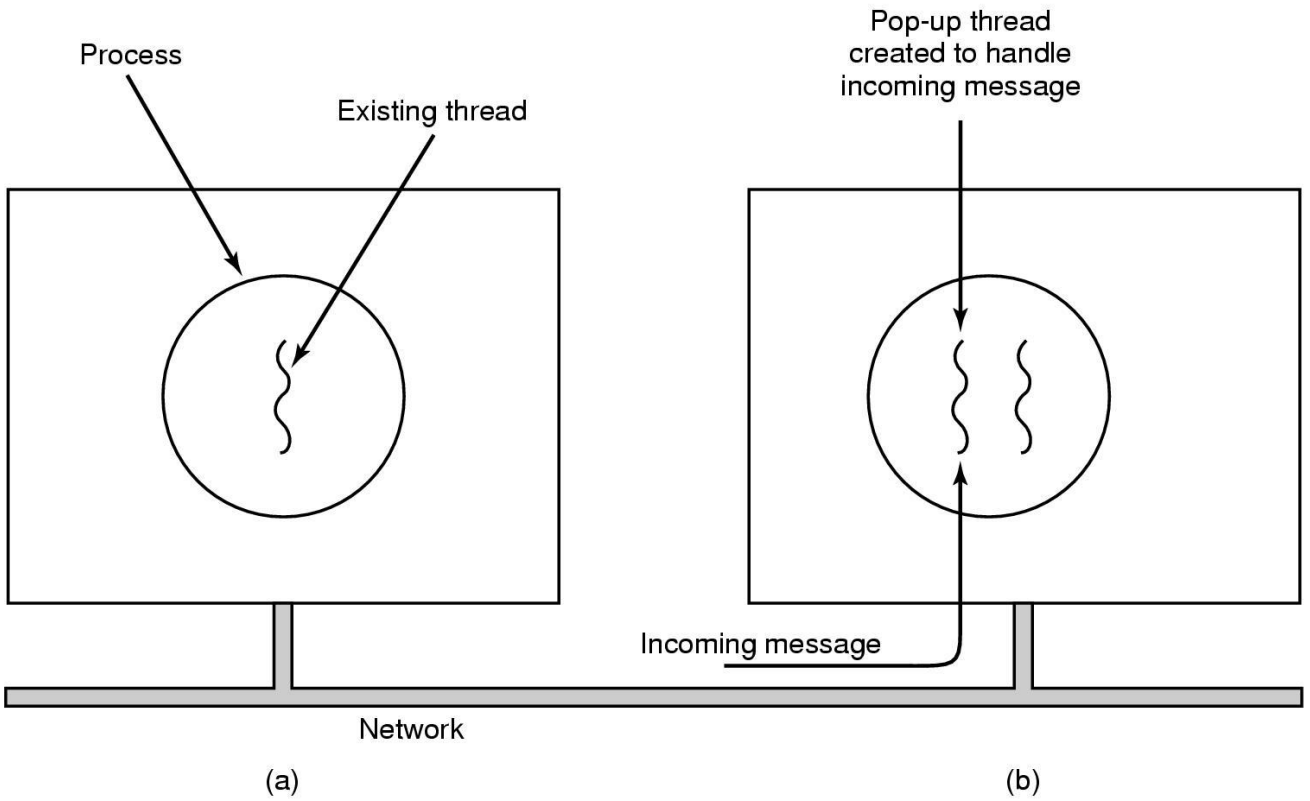
- Basic Idea (upcall):
  - When a thread is blocked, kernel **notifies** the process' run-time system passing parameters:
    - Number of the thread
    - The description of the event that occurred
  - Notification:
    - Kernel activates the run-time system at a known starting address.
  - Run-time systems then can do **scheduling**
  - As soon as the blocked thread is ready, kernel performs another upcall to inform run-time system.
  - It is upto run-time system to schedule.

Scheduler Activations(3)

- Problems:
  - Fundamental reliance on upcalls which doesn't follow a fundamental principle of layered systems

Layer *n* (lower layer for e.g. kernel) offers certain services that layer *n* + 1 (upper layer for e.g. user space) can call on, but layer *n* may not call procedures in layer *n* + 1

Pop-Up Threads



- Used in a distributed system .
- Creation of a new thread when message arrives

  (a) before message arrives

  (b) after message arrives

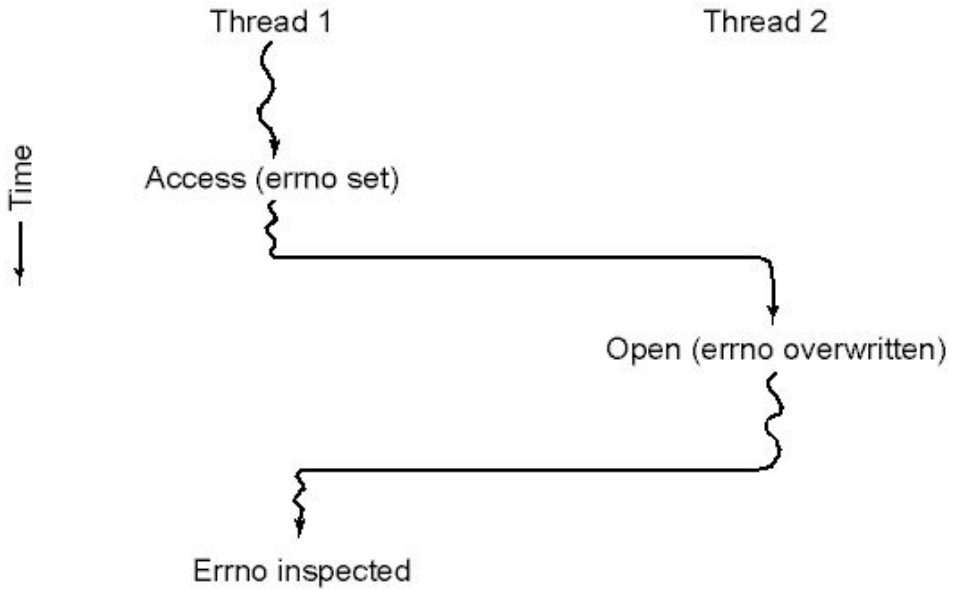
Pop-Up Threads: Advantages

- Pop-up threads do not have any history-registers, stack etc to restore
- Each one starts out fresh and is identical to all others
- So Creation of thread is very quick
- The new thread is given the incoming message to process
- Result: Latency between the message arrival and the start of the process is very short

Having pop-up thread in kernel space:

- Usually easier and faster then putting in user space
- Access all the kernel's tables and I/O devices which may be needed for interrupt processing
- A buggy kernel thread can do more damage then a buggy user thread

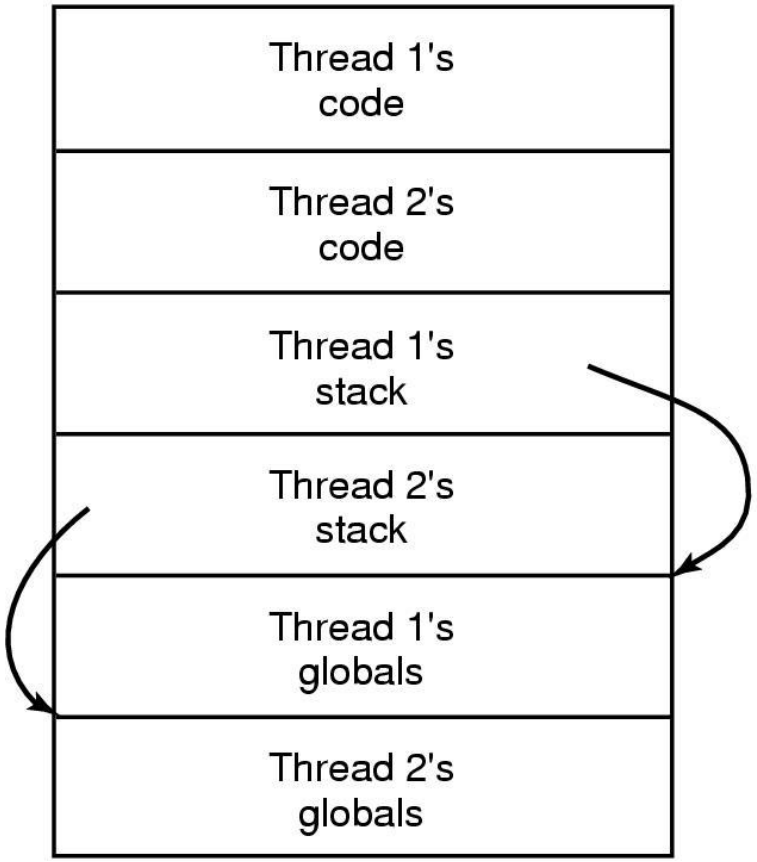
Making Single-Threaded Code Multithreaded (1)



Many existing programs were written for single-threaded processes. Converting these to multithreading is much trickier than it may at first appear.

Conflicts between threads over the use of a global variable

Making Single-Threaded Code Multithreaded (2)



Threads can have private global variables

Quiz 1 syllabus ends here!!!!