

# Processes and Threads

# What is a program?

- A Program is an **executable file** that contains:
  - **Code:** Machine Instructions
  - **Data:** Variables Stored And Manipulated In Memory
    - initialized variables (globals)
    - dynamically allocated variables (malloc, new)
    - stack variables (C automatic variables, function arguments)
- Process != Program
- Example:
  - We can run 2 instances of *Mozilla Firefox*:
    - Same program
    - Separate processes

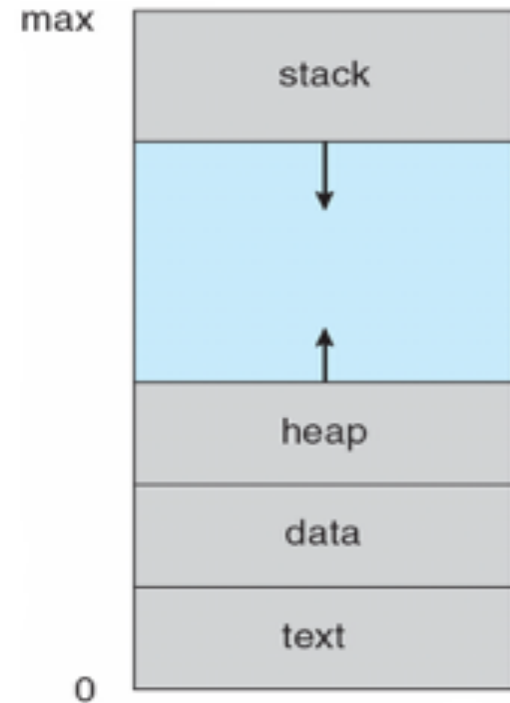
# An analogy

## Baking a Cake

- Need a cake recipe and a kitchen well stocked with all the input: flour, eggs, sugar ....
- the recipe is the **program** (i.e., an algorithm expressed in some suitable notation),
- the baker is the processor (CPU),
- and the cake ingredients are the input data.
- The **process** is the activity consisting of the baker reading the recipe, fetching the ingredients, and baking the cake.

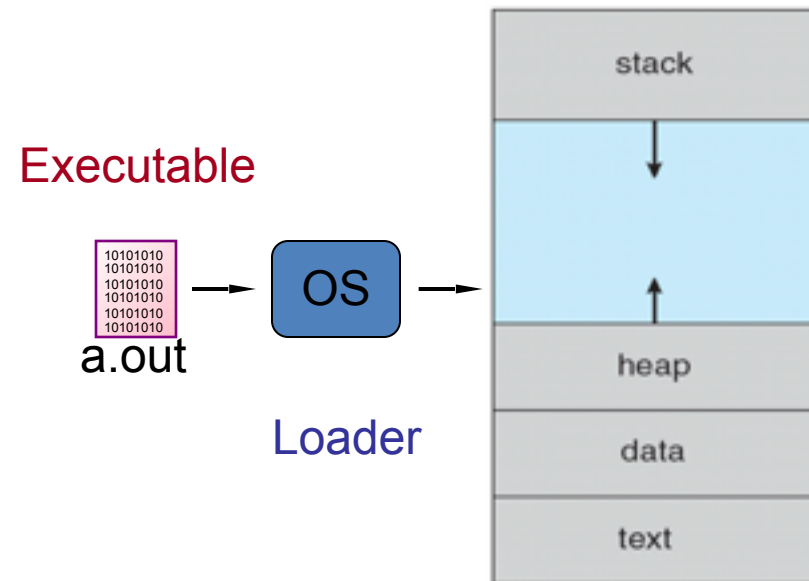
# Process in Memory

- Program becomes process when **executable** file loaded into **memory**
- Process address space
  - set of all memory addresses accessible by a process



# How Program Becomes Process

- When a program is launched
  - OS **loads executable** file of a program into memory
  - Creates **kernel data structure** for the process
  - **Initializes** data
  - Starts from an entry point (e.g., `main()`)



# Processes

- Multiple Parts
  - The Program Code, Also Called **Text Section**
  - Current **Activity** Including the current values of **PC**, Registers
  - **Stack** Containing Temporary Data
    - Function Parameters, Return Addresses, Local Variables
  - **Data Section** Containing **Global** Variables
  - **Heap** Containing Memory **Dynamically** Allocated During Run Time

# Process Data Structures

- OS **represents** a process using a Process Control Block (*PCB*)
  - Has all the details of a process
  - Context of the process
  - Also called **process table entry**

# Process Control Block (PCB)

Process management	Memory management	File management
Registers Program counter Program status word Stack pointer Process state 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	Pointer to text segment Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

Figure: **Fields** of a PCB



# Process Creation

- Four principal events for process creation:
  - **System initialization.**
  - Execution of process creation **system call** by a running process.
  - A user can request to create a new process (typing a command or double clicking an icon).
  - Initiation of a **batch job** (possibly remotely).

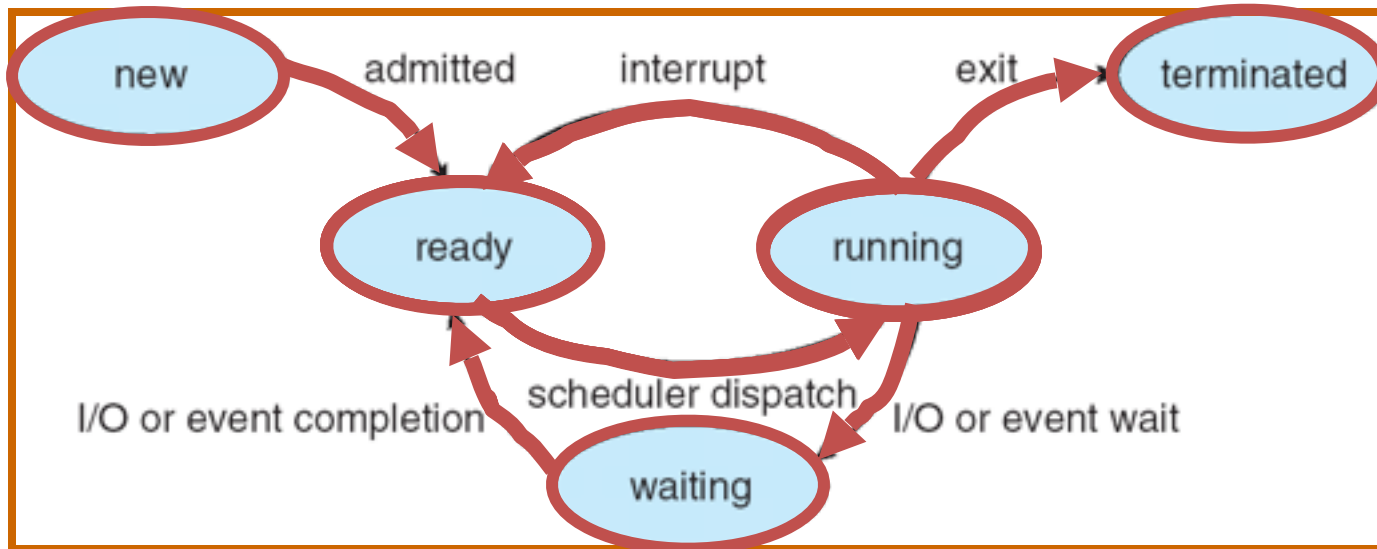
# Process Termination

- Nothing lasts forever, not even processes



- A process terminates usually due to
  - Normal exit (voluntary)
  - Error exit (voluntary)
  - Fatal error (involuntary)
  - Killed by another process (involuntary)

# Lifecycle of a Process



- As a process executes, it changes *state*
  - **new**: The process is being created
  - **ready**: The process is waiting to run
  - **running**: Instructions are being executed
  - **waiting (or, blocked)**: Process waiting for some event to occur
  - **terminated**: The process has finished execution

# The Process Model

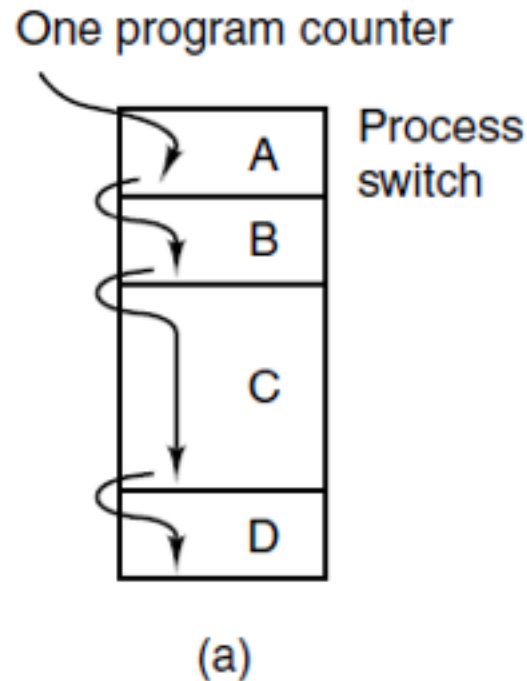


Figure 2-1. (a) Multiprogramming of four programs in memory. Only one physical program counter.

# The Process Model

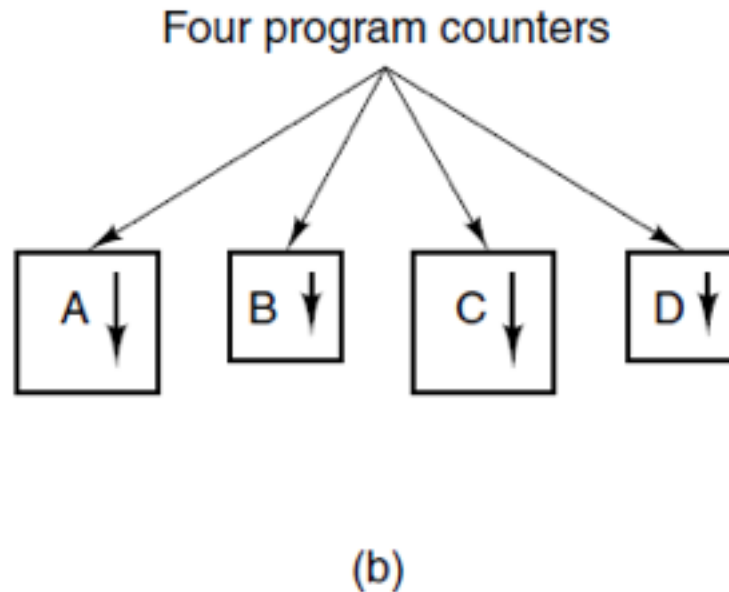


Figure 2-1. (b) Conceptual model of four independent, sequential processes. There exist only one physical program counter, but four different logical counters. When a program runs, its logical counter is loaded into the physical program counter

# The Process Model

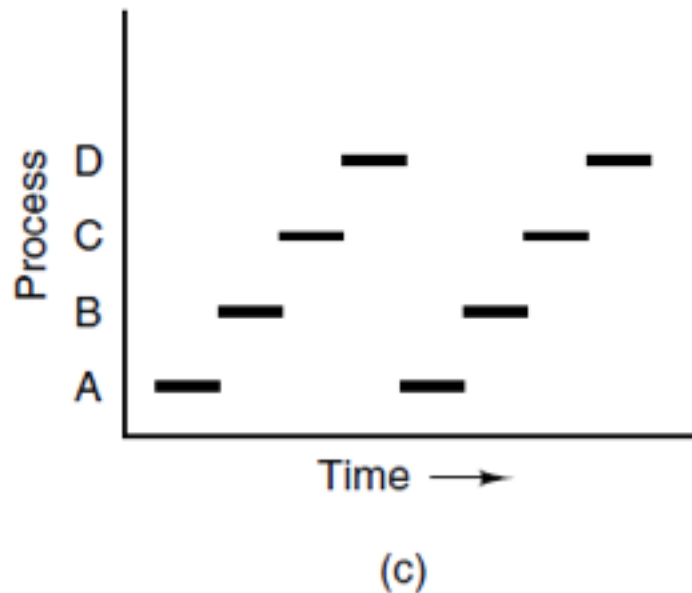
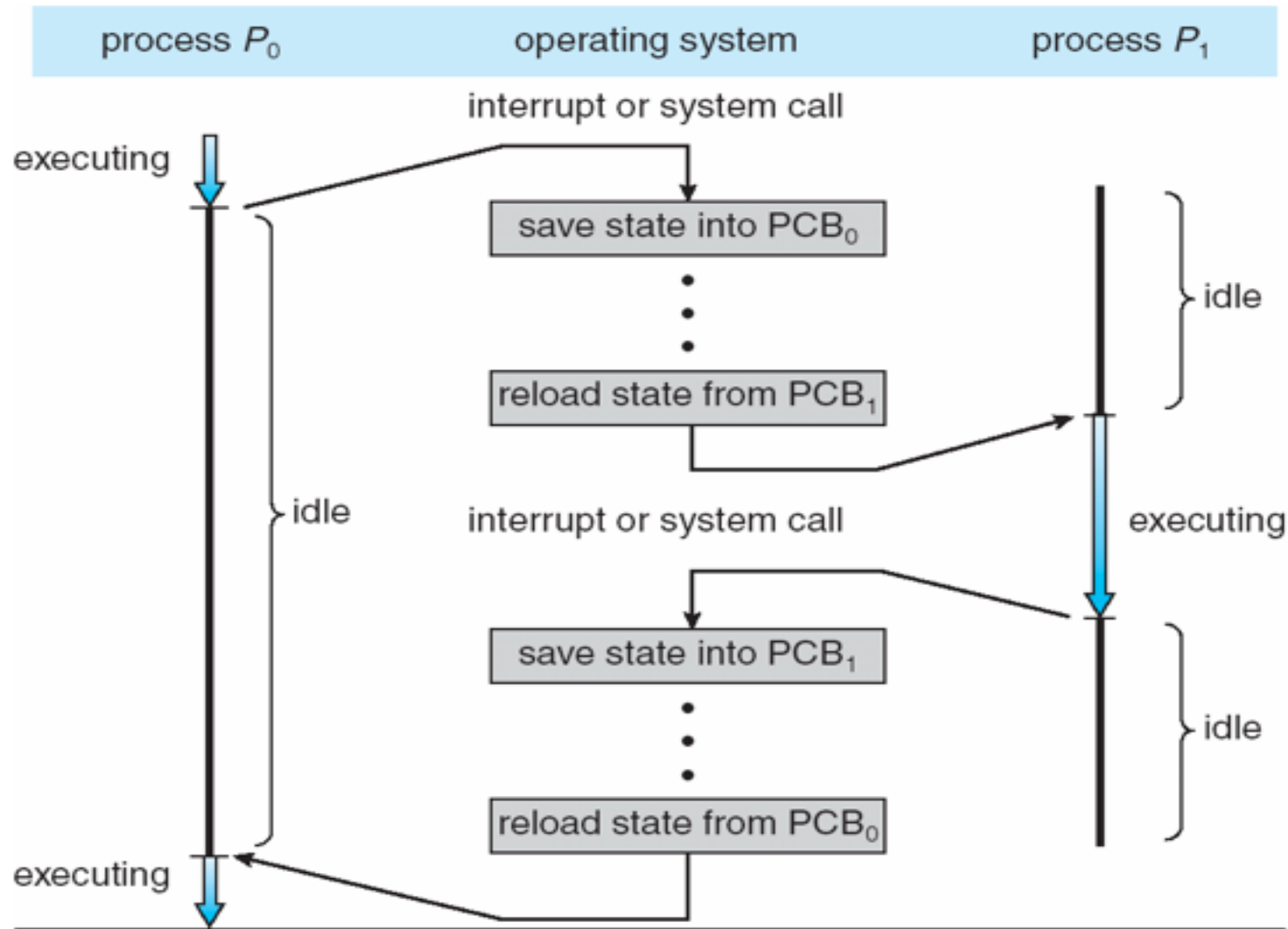


Figure 2-1. (c) Only one program is active at once. All processes have made progress, but at any given instant only one process is actually running.

# Multiprogramming

- Rapidly switching back and forth from process to process is called **multiprogramming**.
- When each process runs, its logical program counter is loaded into the real program counter.
- When it is finished (for the time being), the physical program counter is saved in the process' stored logical program counter in memory.
- From now on, we assume that there is only one CPU.

# CPU Switch From Process to Process





# Context Switch

- For a running process
  - All registers are loaded in CPU and modified
    - E.g. Program Counter, Stack Pointer, General Purpose Registers
- When process relinquishes the CPU, the OS
  - Saves register values to the PCB of that process
- To execute another process, the OS
  - Loads register values from PCB of that process

## ⇒ Context Switch

- Process of switching CPU from one process to another
- Very machine dependent for types of registers

# What does it take to create a process?

- Must construct new PCB
  - Inexpensive
- Must set up new address space
  - More expensive
- Creating a new process is costly
- Context switching is costly

Need something more lightweight!

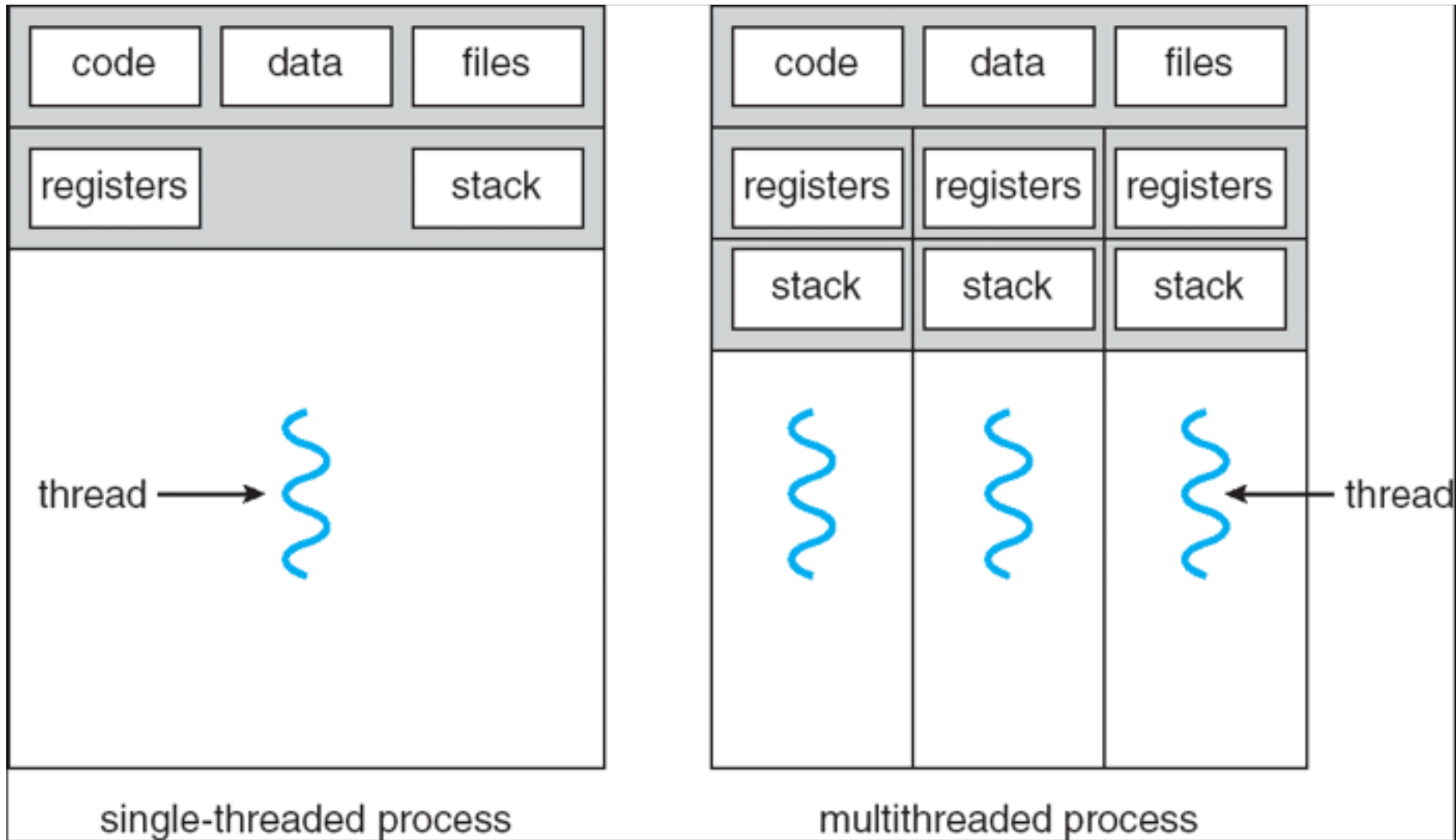
# Threads and Processes

- Most operating systems therefore support two entities:
  - the process,
    - which defines the address space and **general** process attributes
  - the thread,
    - which defines a **sequential** execution stream **within** a process
    - Like a miniprocess within a process
- A thread is bound to a single process.
  - For each process, however, there may be many threads.
- Threads are the unit of scheduling
- Processes are **containers** in which threads execute

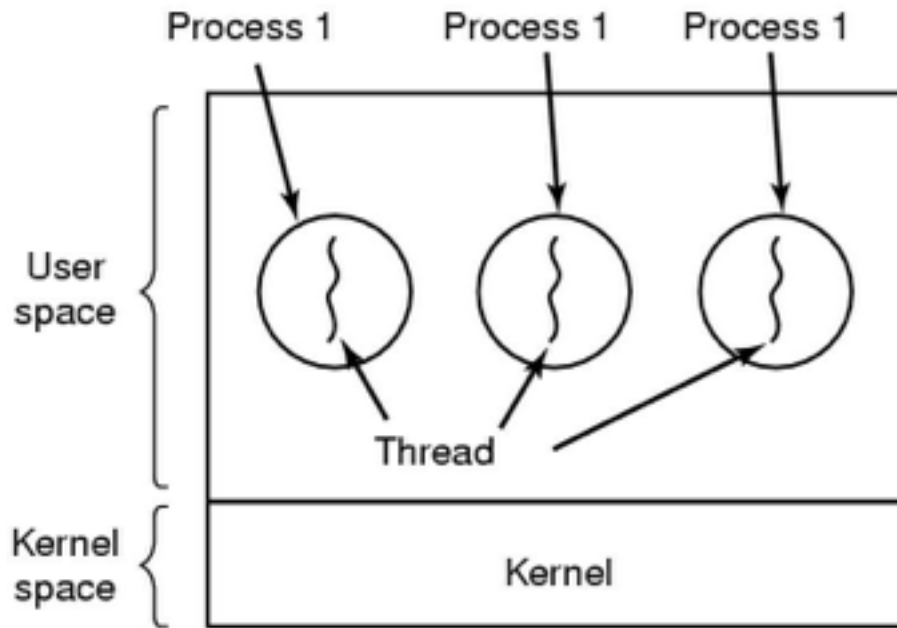
# Threads and Processes

- Thread within the same process needs a new ability:
  - Share an address space and all of it's data among themselves.
- Remember, processes does not share their address spaces.

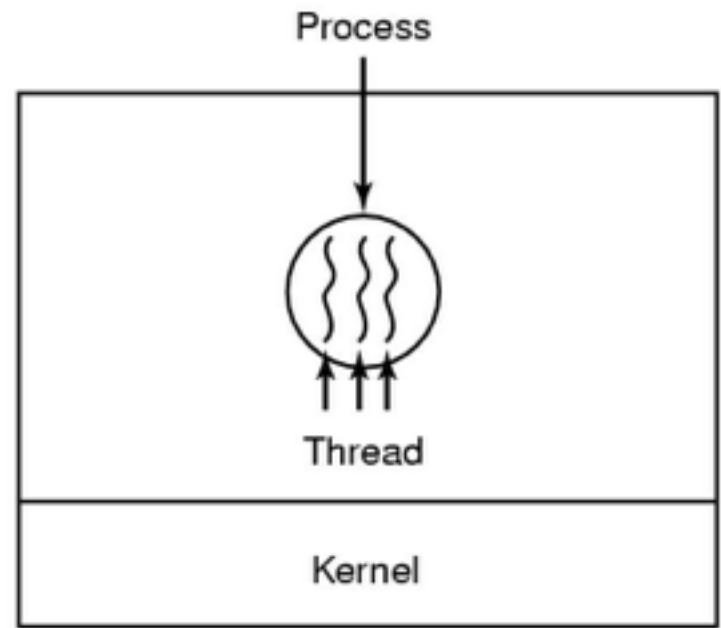
# Multithreaded Processes



# The Classical Thread Model



(a)



(b)

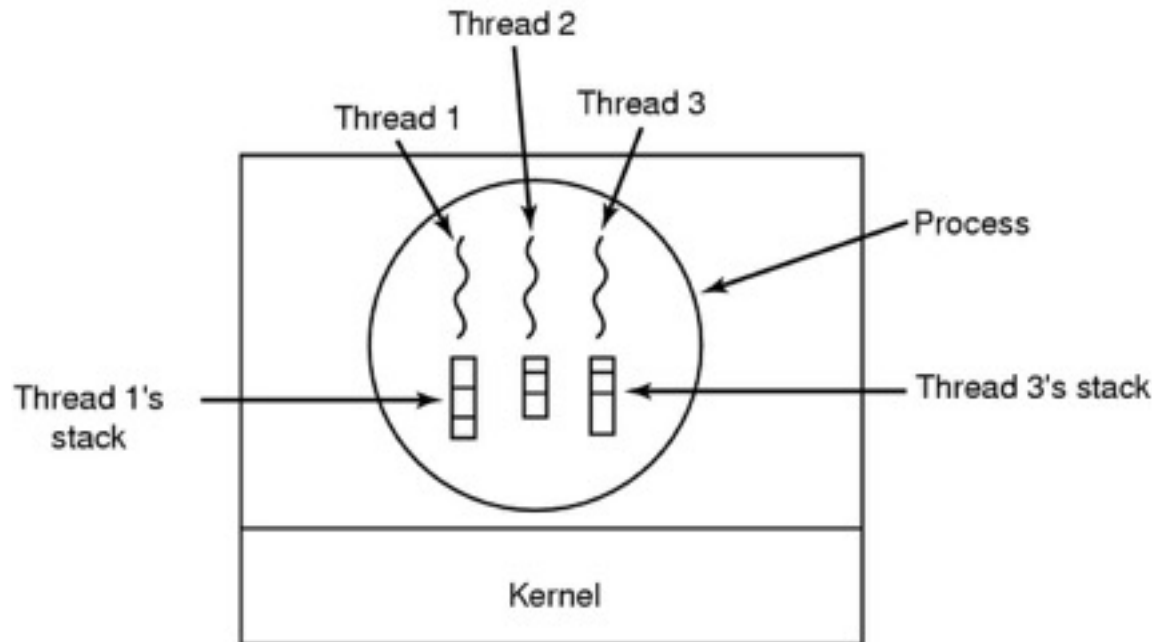
(a) Three processes each with one thread

(b) One process with three threads

# The Classical Thread Model

- Shared information
  - Address space: text, data structures, etc.
  - I/O and file: comm. ports, directories and file descriptors, etc.
  - Global variables and child processes.
  - Accounting info: stats
- Private state
  - State (ready, running and blocked)
  - Registers
  - Program counter
  - Execution stack
- Each thread execute separately

# Why each thread has its own stack?



- What will happen if they share one stack?
  - Each thread call different **procedures** and each has a different **execution history**.



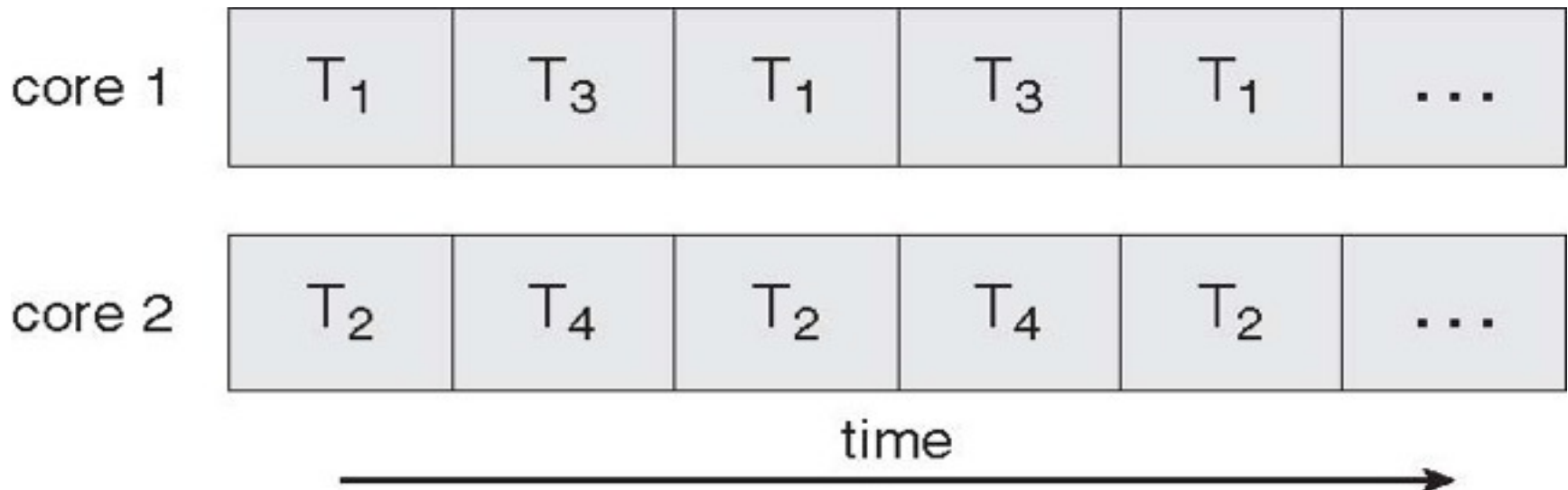
# Thread Context Switch

- Multiplex multiple threads on single CPU
- Similar to process context switch, but less expensive
  - Still needs to switch register set
  - But no memory management related work!!!

# Concurrent Execution on a Single-core System



# Parallel Execution on a Multicore System



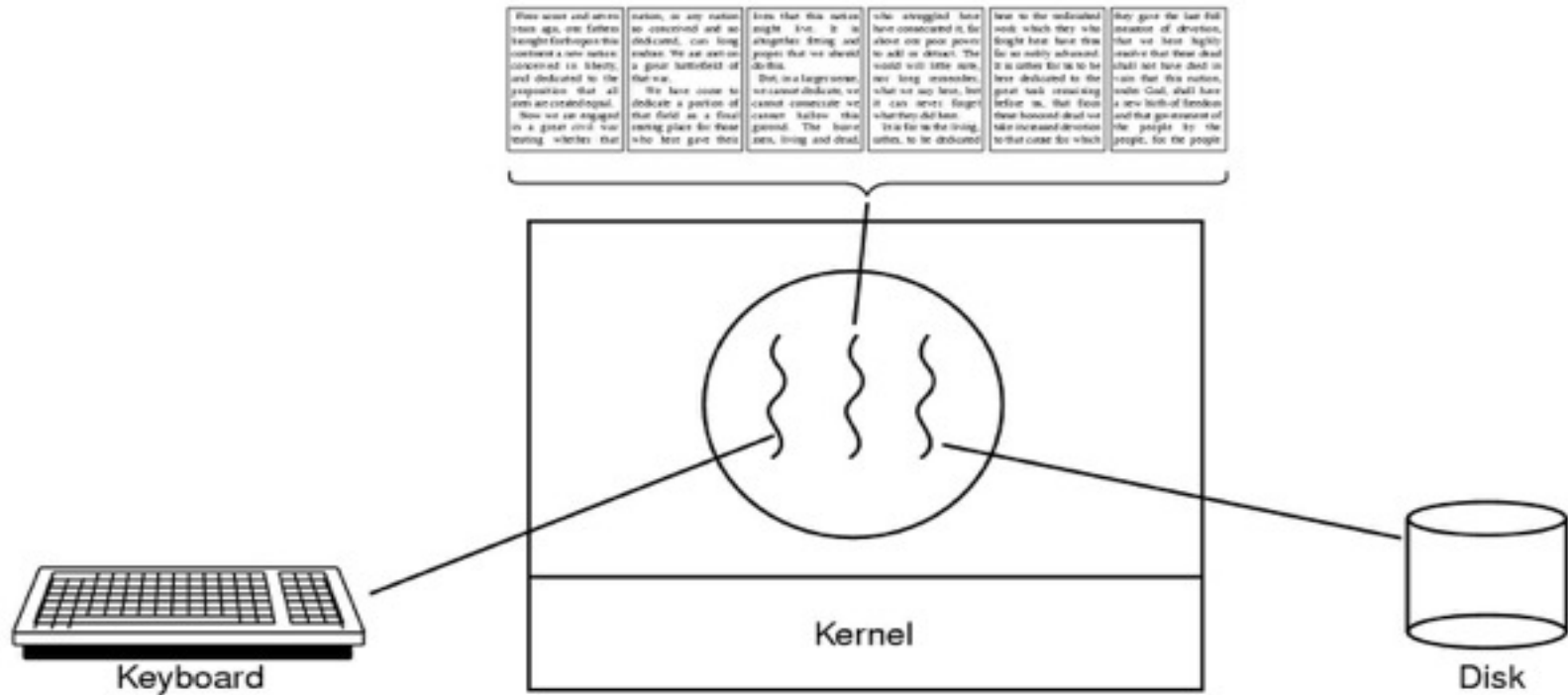
# Thread Dynamics

- Threads are dynamically created/terminated
- Thread is the unit of scheduling.  
Multiple threads need to be scheduled
  - Ready
  - Blocked
  - Running
  - Terminated
- Threads share CPU and on single processor machine only one thread can run at a time

# Thread Usage

- Why need threads?
  - Simplify coding
  - Concurrent activities within a process
    - Better CPU utilization
    - Better responsiveness
  - Less costly to create & switch
  - Utilizing parallelism of multi-processor systems

# Thread Usage: word processor



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

# Thread Implementation

- In user space
  - Kernel unaware of multiple threads
  - User level runtime system does scheduling
- In kernel space
  - Kernel supports threads (lightweight process)

# User-Level Threads

- The thread scheduler is part of a *user-level library*
- Each thread is represented simply by:
  - PC
  - Registers
  - Stack
  - Small control block
- All thread operations are at the user-level:
  - Creating a new thread
  - switching between threads
  - synchronizing between threads



# User-Level vs. Kernel Threads

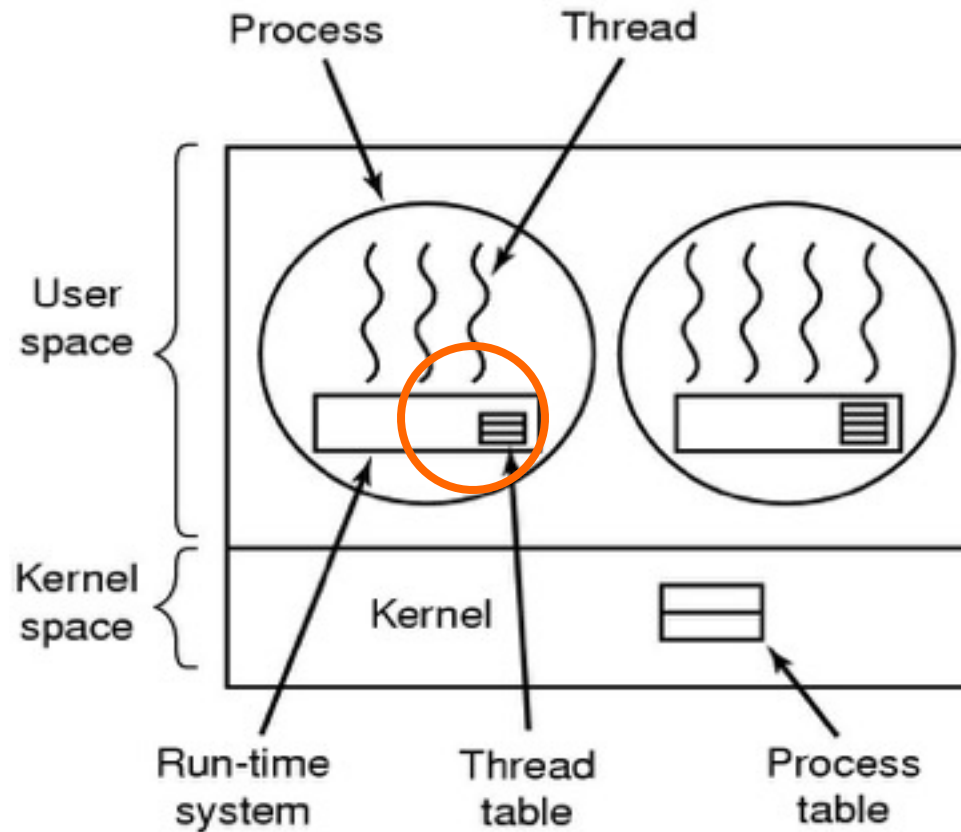
## User-Level

- Managed by application
- Kernel not aware of thread
- Context switching cheap
- Create as many as needed
- Must be used with care

## Kernel-Level

- Managed by kernel
- Consumes kernel resources
- Context switching expensive
- Number limited by kernel resources
- Simpler to use

# Implementing Threads in User Space



A user-level threads package

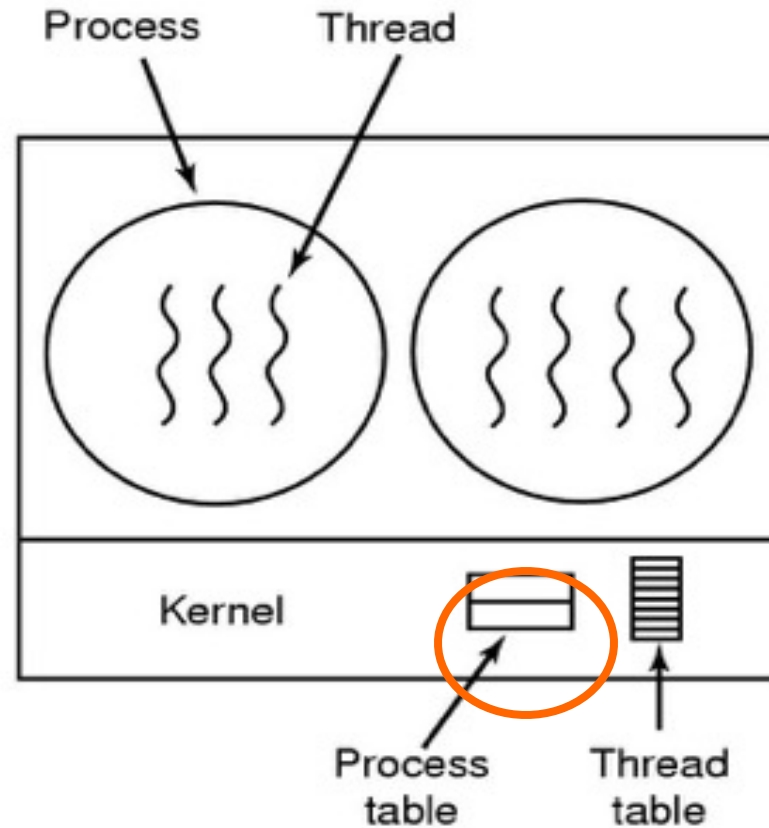
# User-level Threads

- Advantages
  - Fast Context Switching:
    - Switching entirely in user mode - local procedures.
    - No need to trap to kernel, no memory flush;
  - Customized Scheduling
- Disadvantages
  - Blocking
    - Any user-level thread can **block** the entire task executing a single system call (page fault is similar case).
  - No protection, threads are expected to be polite to share CPU.
    - Uncooperative/buggy threads may monopolize CPU.

# Kernel Threads

- Kernel threads may not be as heavy weight as processes, but they still suffer from performance problems:
  - Any thread operation still requires a system call.
  - The kernel doesn't trust the user
    - there must be lots of checking on kernel calls

# Implementing Threads in the Kernel

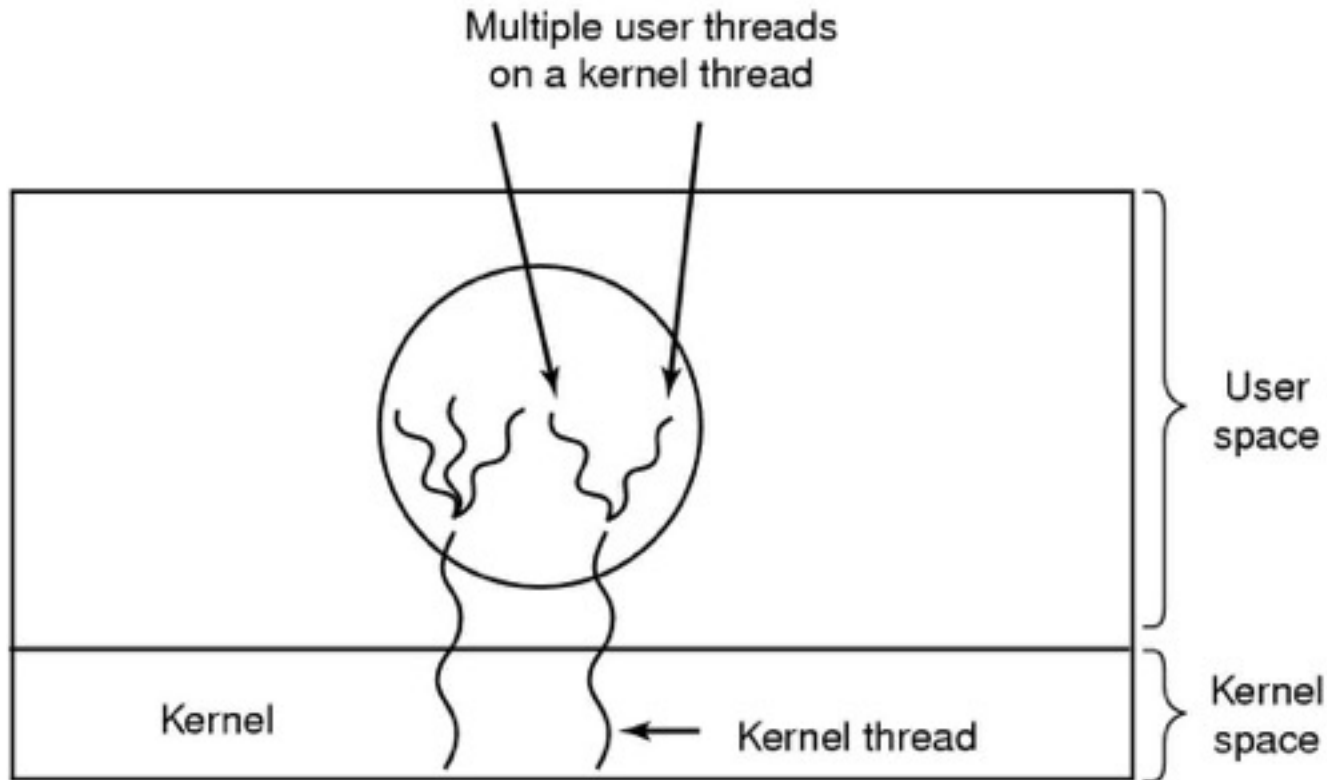


A thread package managed by the kernel

# Kernel-Level Threads

- Advantages:
  - Kernel aware of threads, if one thread blocks, can schedule another thread in the process.
- Disadvantages:
  - Context switch is more expensive.

# Hybrid Implementations



Multiplexing user-level threads onto kernel-level threads

# Hybrid Implementations

- Combining the advantages of the 2 methods
- the kernel is aware of only the kernel-level threads and schedules those.
- Each kernel-level thread has some set of user-level threads that take turns using it.
- These user-level threads are created, destroyed, and scheduled just like user-level threads in a process



# Process Context Switching vs Thread Context Switching

- We consider user process and user thread here

PCS	TCS
Requires user mode to kernel mode switching	Does not requires user mode to kernel mode switching
Changing address space is required	Address space remains the same
All the memory addresses that the processor accounts get flushed.	All the memory addresses that the processor accounts remain saved.
In addition to switching of PC, registers and stack pointers, memory management related tasks such as flushing and reloading cache, TLB, segmentation buffer are performed.	No memory management related task. TCS involves switching of PC, registers and stack pointers.

Thanks 😊