Processes and Threads

Processes

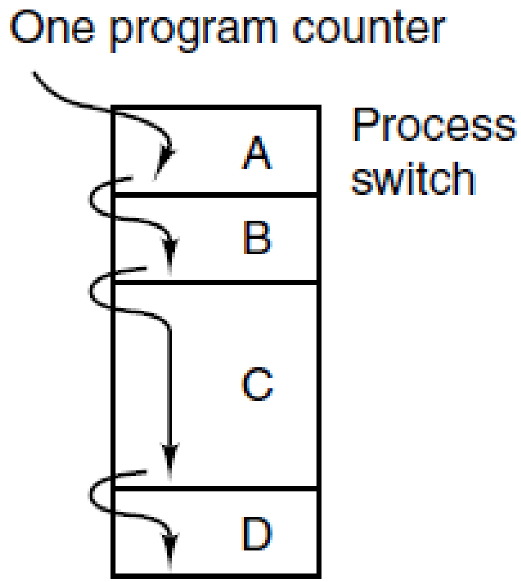
- Operating system provided ***abstraction*** to represent what is needed to run a single program.
- The unit of execution.
- Processes turn a single CPU into multiple virtual CPUs.

- Each process has its own virtual CPU.
  - A single CPU can only execute one instruction at a time.
  - However, the OS **rapidly switches** between processes, giving the illusion that multiple programs are running at the same time.
  - This switching is done so fast (thousands of times per second) that users don't notice the pauses.
  - So, each process feels like it has its own CPU, but in reality, they're all sharing the same physical CPU.
  - This concept is called **virtualization** — creating the appearance of many CPUs from one.

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

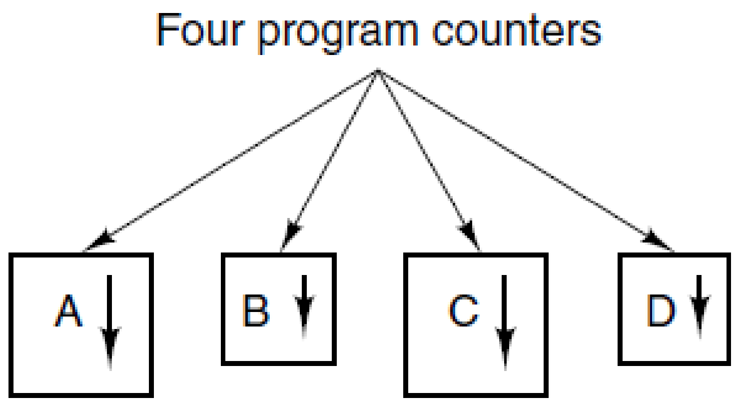
The Process Model



(a)

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

The Process Model



(b)

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

Physical Program Counter and Logical Program Counter

1. Physical Program Counter (PC)

- This is a **real hardware register** inside the CPU.
- It always points to the **next instruction** of the **currently running process**.
- There is only **one physical program counter**, because the CPU can only execute **one process at a time**.

2. Logical Program Counter

- This is a **saved copy of the program counter** for each individual process.
- When a process is **not running**, its logical PC value is stored in its **Process Control Block (PCB)**.
- During a **context switch**, the OS:
  - Saves the **physical PC** into the **logical PC** of the outgoing process.
  - Loads the **logical PC** of the next process into the **physical PC**.
- So, each process has its own logical PC value, even though there's only one physical PC.

The Process Model

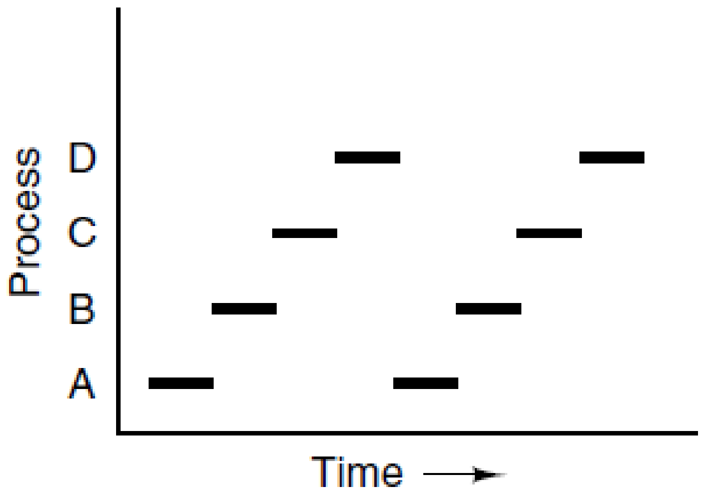


Figure 2-1. (c) Only one program is active at once. All processes have made progress, but at any given instant only one process is actually running.

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

What is a program?

- A Program is an <u>**executable file**</u> 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

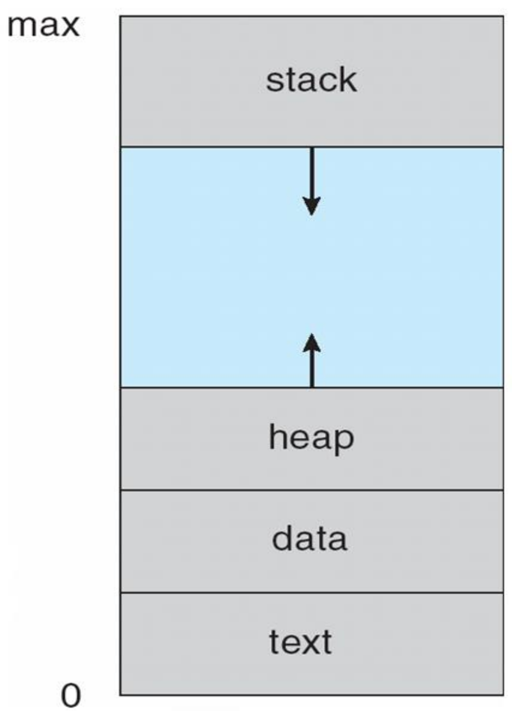
An analogy

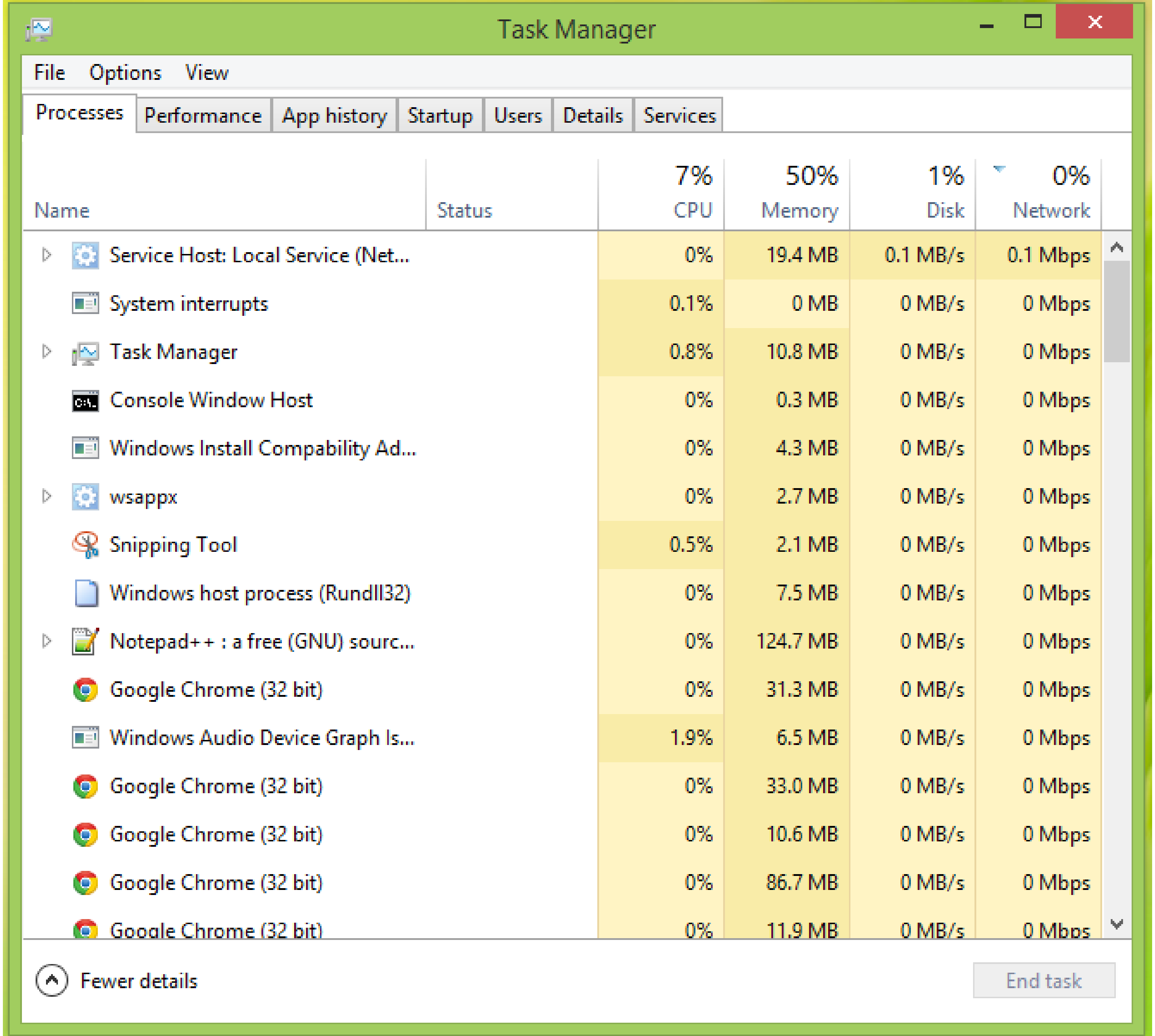
Baking a Cake

- 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

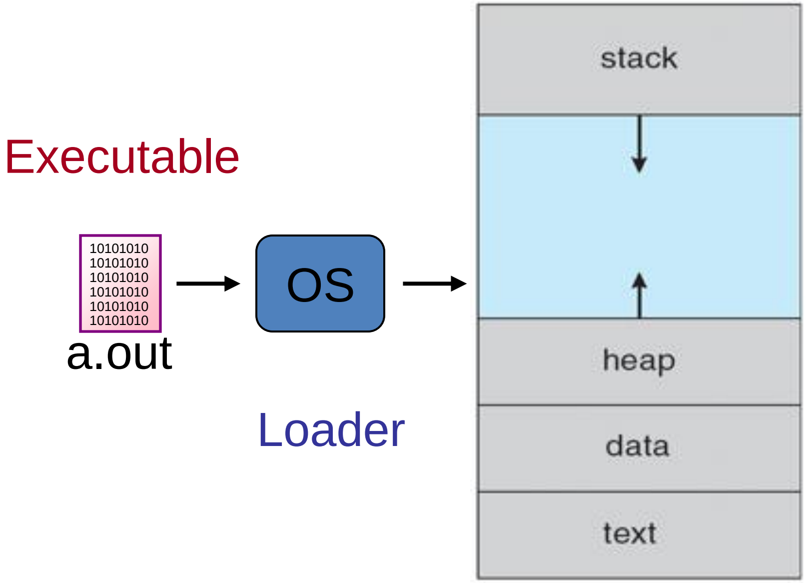
- Program becomes process when executable file loaded into memory
- Process address space
  - set of all memory addresses accessible by a process




Task Manager
File
Options
View
Processes
Performance
App history
Startup
Users
Details
Services
Name
Status
7%
CPU
50%
Memory
1%
Disk
0%
Network
Service Host: Local Service (Net... 0% 19.4 MB 0.1 MB/s 0.1 Mbps
System interrupts 0.1% 0 MB 0 MB/s 0 Mbps
Task Manager 0.8% 10.8 MB 0 MB/s 0 Mbps
Console Window Host 0% 0.3 MB 0 MB/s 0 Mbps
Windows Install Compatibility Ad... 0% 4.3 MB 0 MB/s 0 Mbps
wsappx 0% 2.7 MB 0 MB/s 0 Mbps
Snipping Tool 0.5% 2.1 MB 0 MB/s 0 Mbps
Windows host process (Rundll32) 0% 7.5 MB 0 MB/s 0 Mbps
Notepad++ : a free (GNU) sourc... 0% 124.7 MB 0 MB/s 0 Mbps
Google Chrome (32 bit) 0% 31.3 MB 0 MB/s 0 Mbps
Windows Audio Device Graph Is... 1.9% 6.5 MB 0 MB/s 0 Mbps
Google Chrome (32 bit) 0% 33.0 MB 0 MB/s 0 Mbps
Google Chrome (32 bit) 0% 10.6 MB 0 MB/s 0 Mbps
Google Chrome (32 bit) 0% 86.7 MB 0 MB/s 0 Mbps
Google Chrome (32 bit) 0% 11.9 MB 0 MB/s 0 Mbps
Fewer details
End task

How Program Becomes Process

- When a program is launched
  - OS loads program into memory
  - Creates kernel data structure for the process
  - Initializes data (global/static variables)
  - Starts from an entry point (e.g., main())



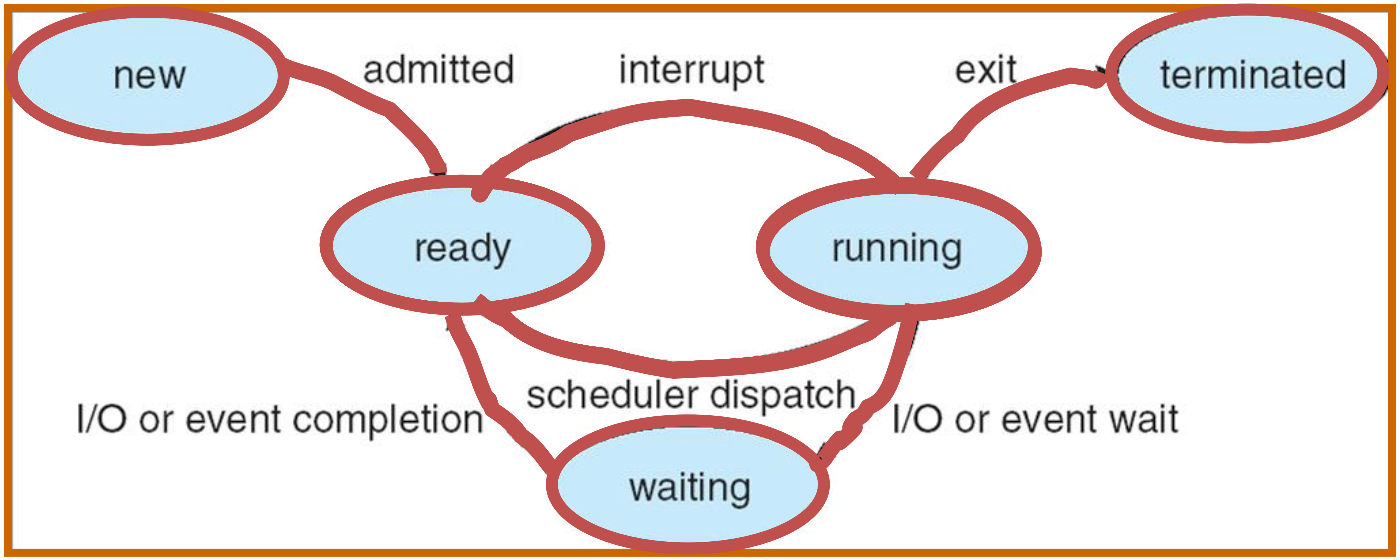
Process Creation

- 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).

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

Process Data Structures

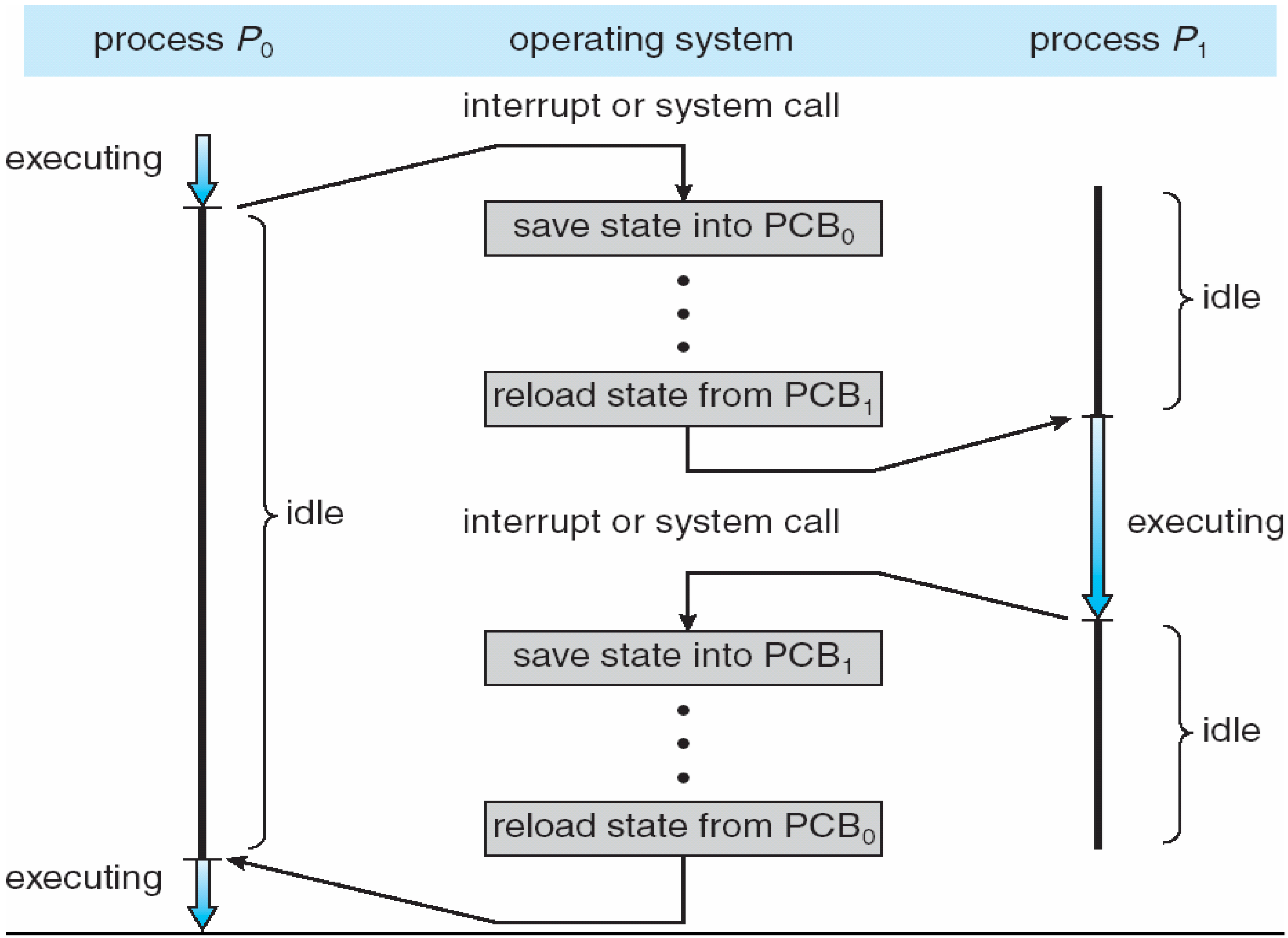
- OS represents a process using a Process Control Block (*PCB)*
  - Has all the details of a process
  - Context of the process
  - Also called **process table entry**

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

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!

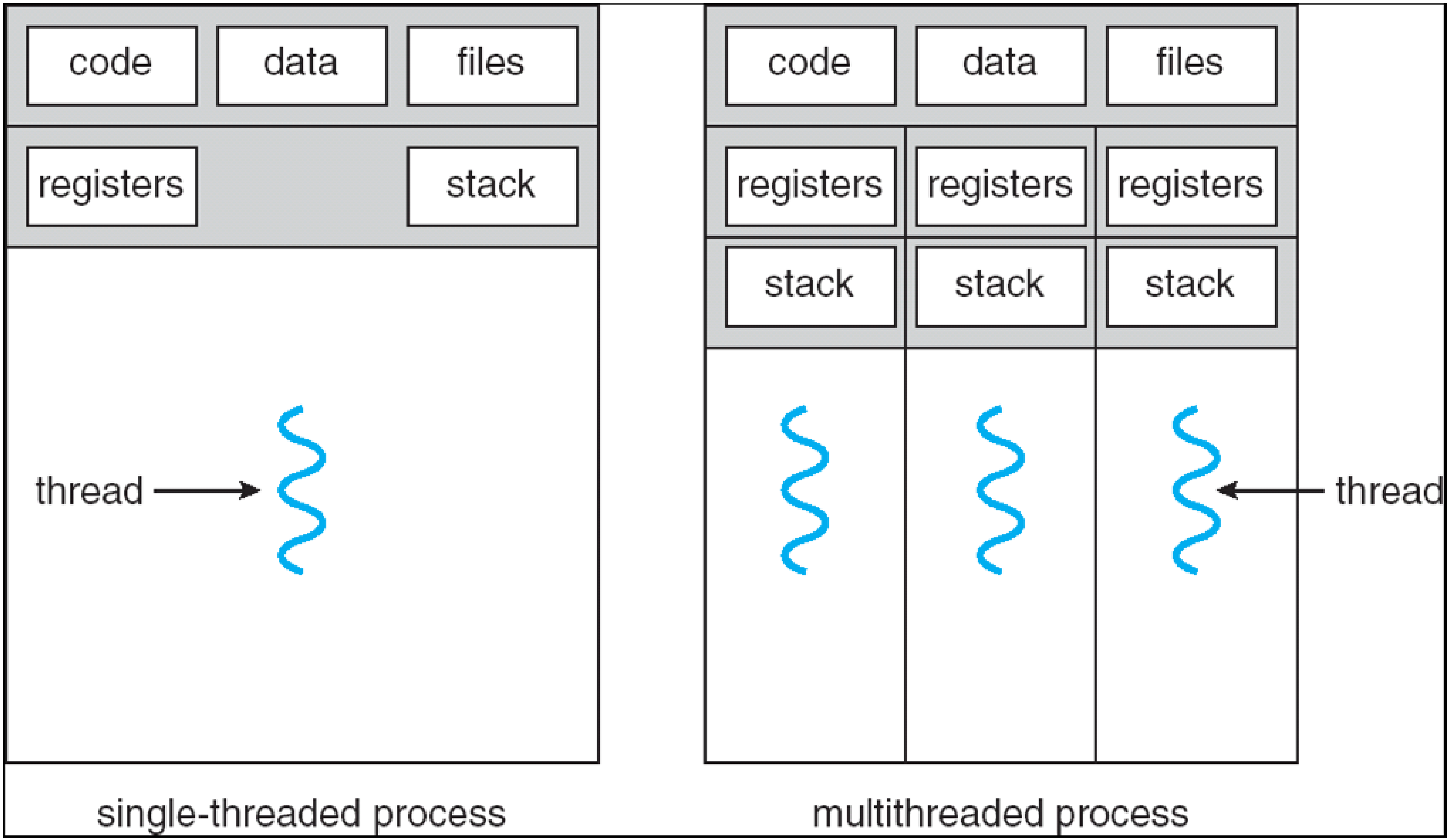
Threads and Processes

- 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

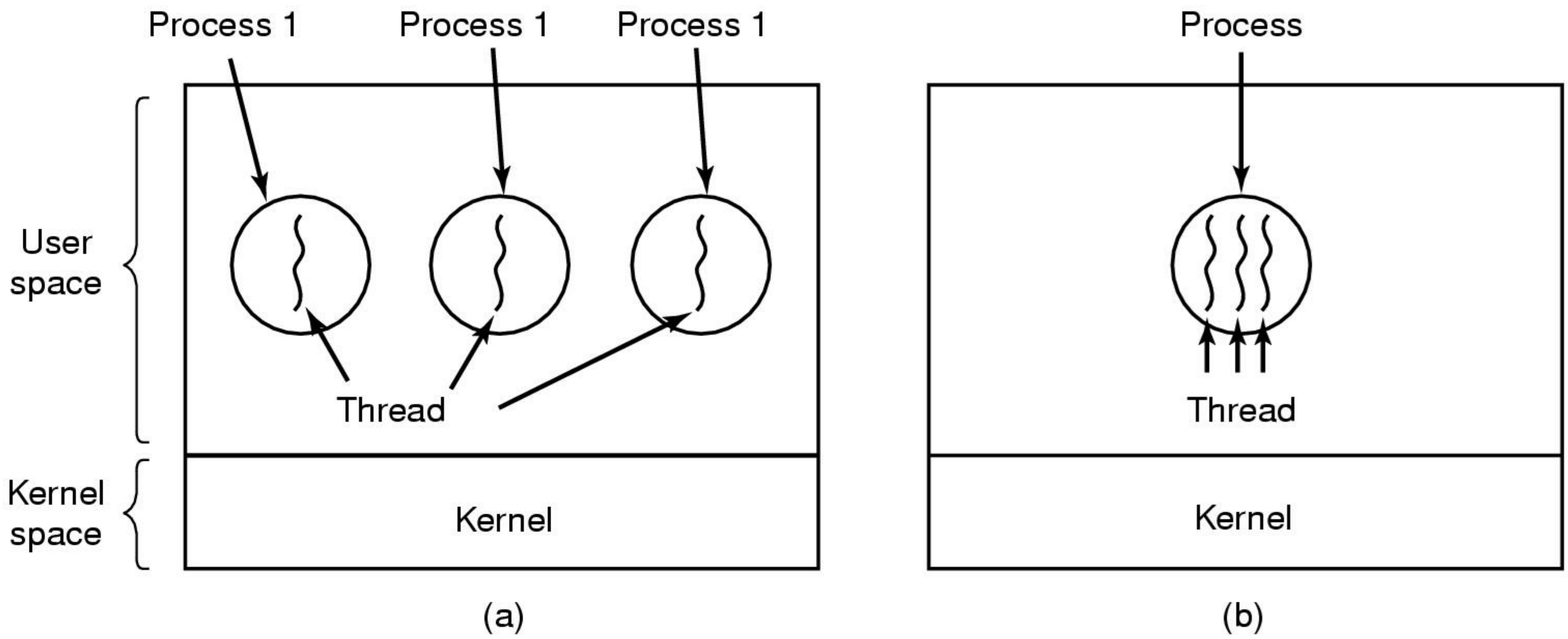
Threads and Processes

- 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

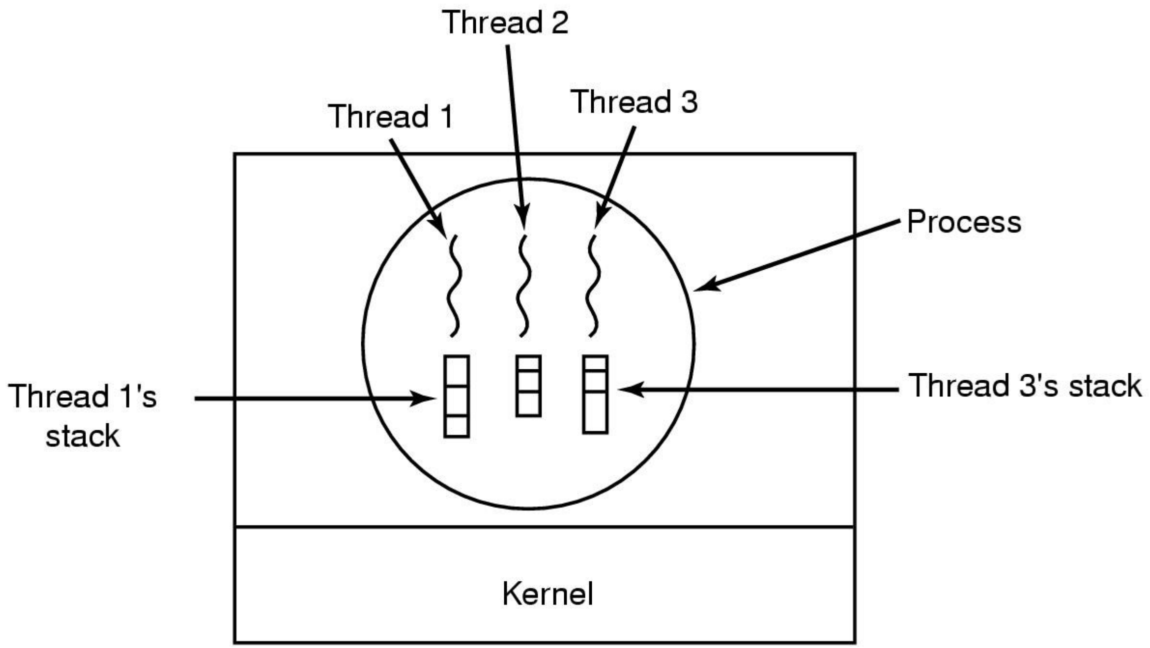


(a) Three processes each with one thread
(b) One process with three threads

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?

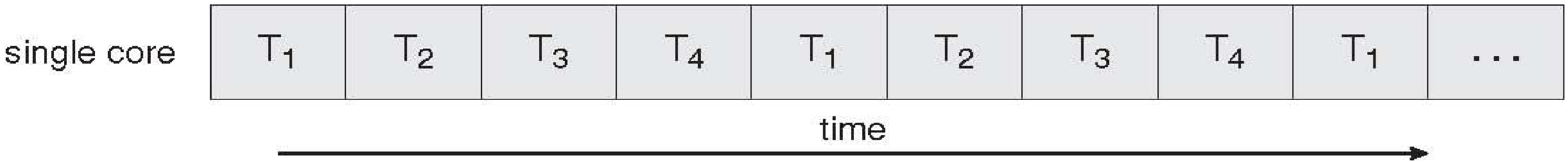


- What will happen if they share one stack?
  - Each thread call different procedures and each has a different execution history.

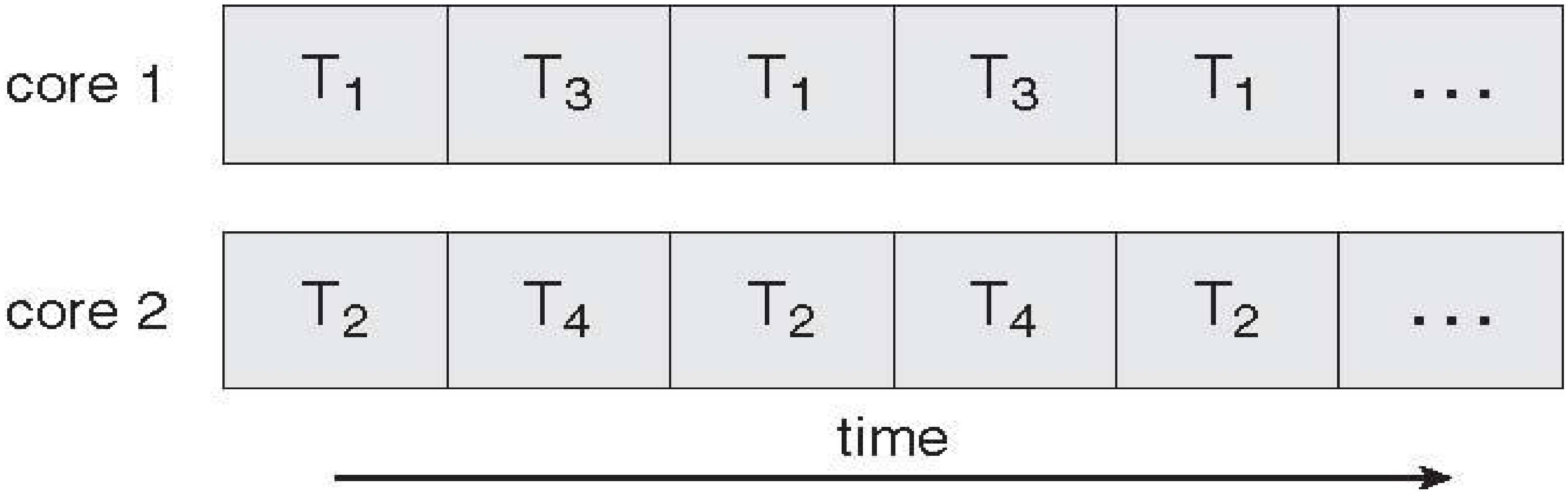
Thread Context Switch

- 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

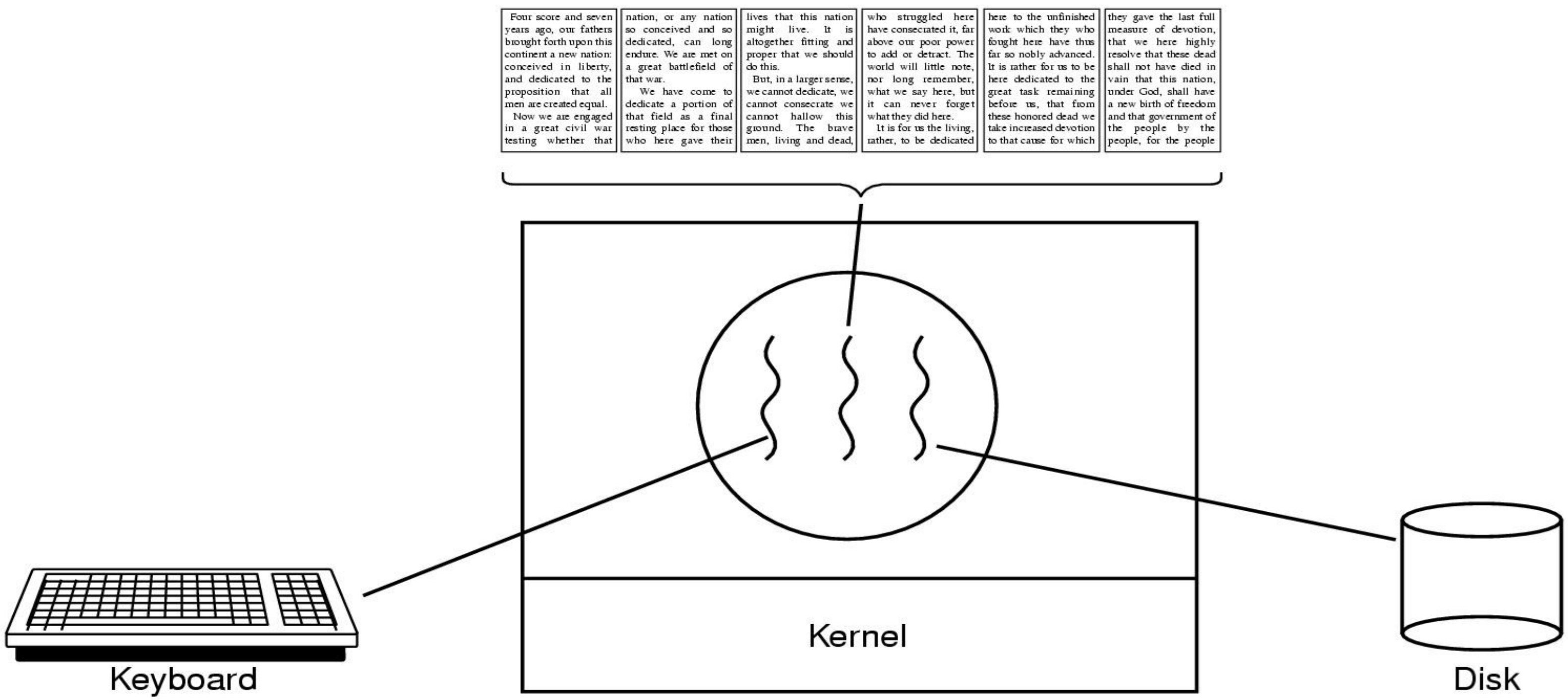
- Threads are dynamically created/terminated
- Thread is the unit of scheduling.

  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

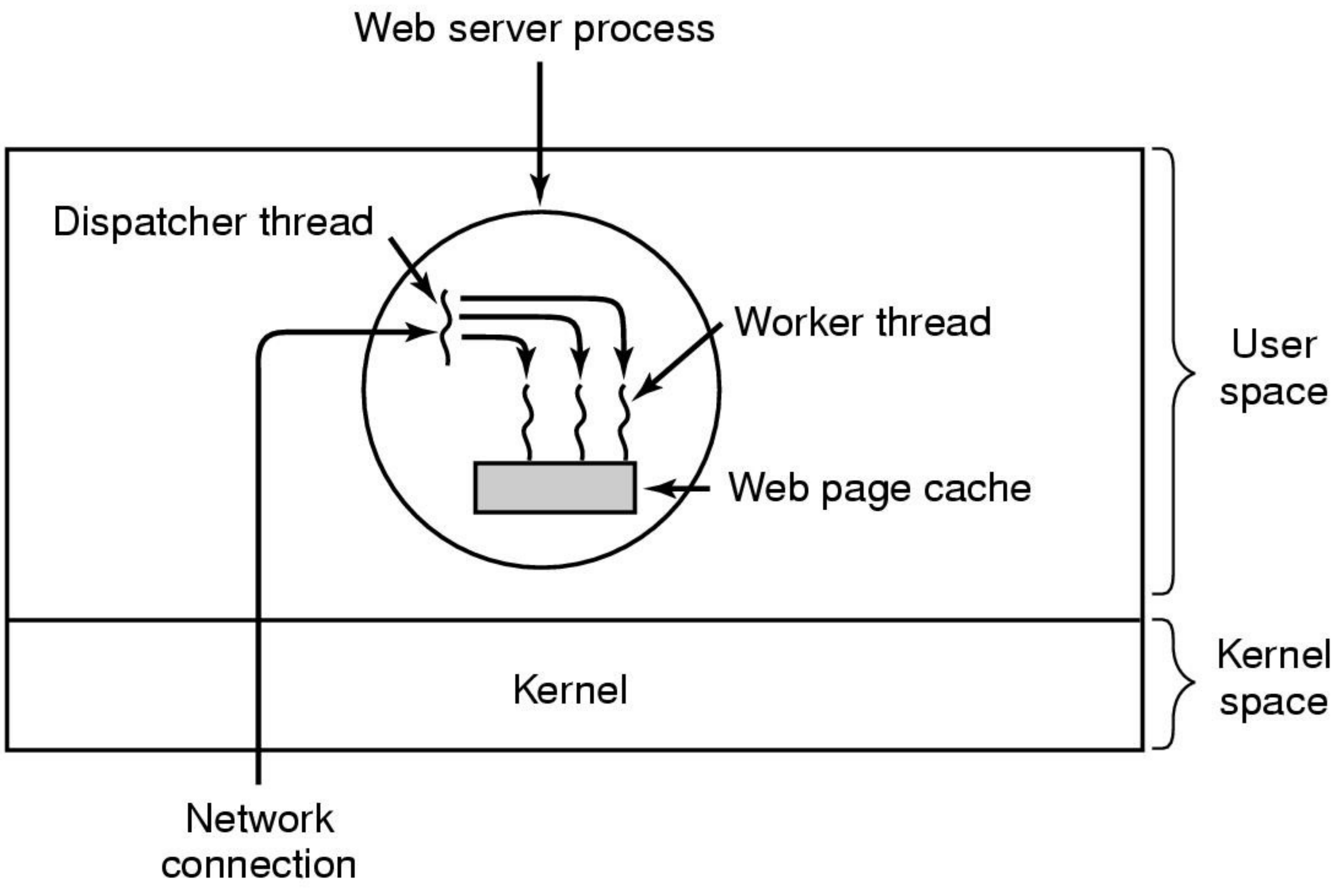
- Why need threads?
  - Simplify coding
  - Concurrent activities within a process
    - Better CPU utilization
    - Better responsiveness
  - Less costly to create & switch
  - Utilizing parallelism of multi-processor systems

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 Usage: Web Server



Blocking System Calls

- 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 to implement

Thread Implementation

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

Step-by-Step Process: From Creating User Threads to Ending the Process

1. A Process is Started

- ➔ A program is loaded into memory by the **OS loader**.
- ➔ The kernel creates a process control block (PCB) for it.
- ➔ The process starts with **one kernel thread**, which begins executing in user mode.
- ➔ The OS schedules this process on a CPU core.

2. User-Level Thread Library is Initialized

The running program includes a **user-level thread library** (e.g., GNU Pth, libucontext, custom coroutine framework).

- ➔ This library lives entirely in user space.
- ➔ It sets up:
  - ◆ Data structures to manage threads (e.g., thread table)
  - ◆ Thread metadata: Program Counter (PC), stack, register set, and small control block

3.User Threads Are Created

- ➔ The application calls something like create_thread() — not a system call.
- ➔ The user-level library:
  - ◆ Allocates a **stack** for the new thread
  - ◆ Initializes **PC** to the thread function
  - ◆ Stores thread state in its thread **control block**
  - ◆ Adds the thread to a ready queue
  - ◆ No system call is made — this is all in user space.

4. Thread Scheduling Happens (User-Level)

- ➔ The user-level scheduler:
- ➔ Picks a thread from the ready queue
- ➔ Performs a context switc h by:
  - ◆ Saving current thread's registers and stack pointer
  - ◆ Loading new thread's registers and stack pointer
  - ◆ Jumps to the new thread's PC
- ➔ Still **no kernel involvement** — very fast.

5. Kernel Sees Only One Thread

- ➔ The kernel still sees this as **one process,** regardless of how many user threads are created.
- ➔ Only the currently selected user thread is actively running.
- ➔ The process gets scheduled on a CPU core by the kernel scheduler.

6. User Thread Yields or Blocks (Voluntarily)

- ➔ When a thread finishes or yields (yield()), the user-level scheduler picks the next thread and switches context again.
- ➔ If a thread performs blocking I/O using a system call, the entire process blocks (because kernel doesn’t know other threads exist).

This is a drawback: one thread blocks → whole process is paused.

7. Kernel Performs Context Switch Between Processes

- ➔ A context switch between processes occurs when:
- ➔ A process blocks (e.g., I/O, waiting)
- ➔ A process is preempted (its time slice ends)
- ➔ A higher-priority process is scheduled

8. Execution Resumes

➔ Now the new process starts executing on the CPU.

➔ If that process also uses user-level threads, its thread scheduler picks a thread and runs it.

Process-level context switch → kernel-level

Thread-level context switch → user-level

9. Process Termination

➔ Eventually, the main thread (and all user threads) complete.

➔ The process calls exit() or finishes main().

➔ A system call is made to terminate.

User-Level Threads

The thread scheduler is part of a *user-level library.* The thread scheduler is not part of the operating system (OS) kernel. It's a software component in a user-level library that handles scheduling between threads within the same process.

- Each thread is represented simply by:
  - **PC** (Program Counter): Tracks where the thread is in its execution.
  - **Registers:** Stores temporary data during execution.
  - **Stack:** Holds function calls, local variables, and return addresses.
  - **Control Block:** A small data structure that keeps metadata like thread ID or state.
- All thread operations are at the user-level:
  - Creating a new thread
  - switching between threads
  - synchronizing between threads

User-Level vs. Kernel Threads

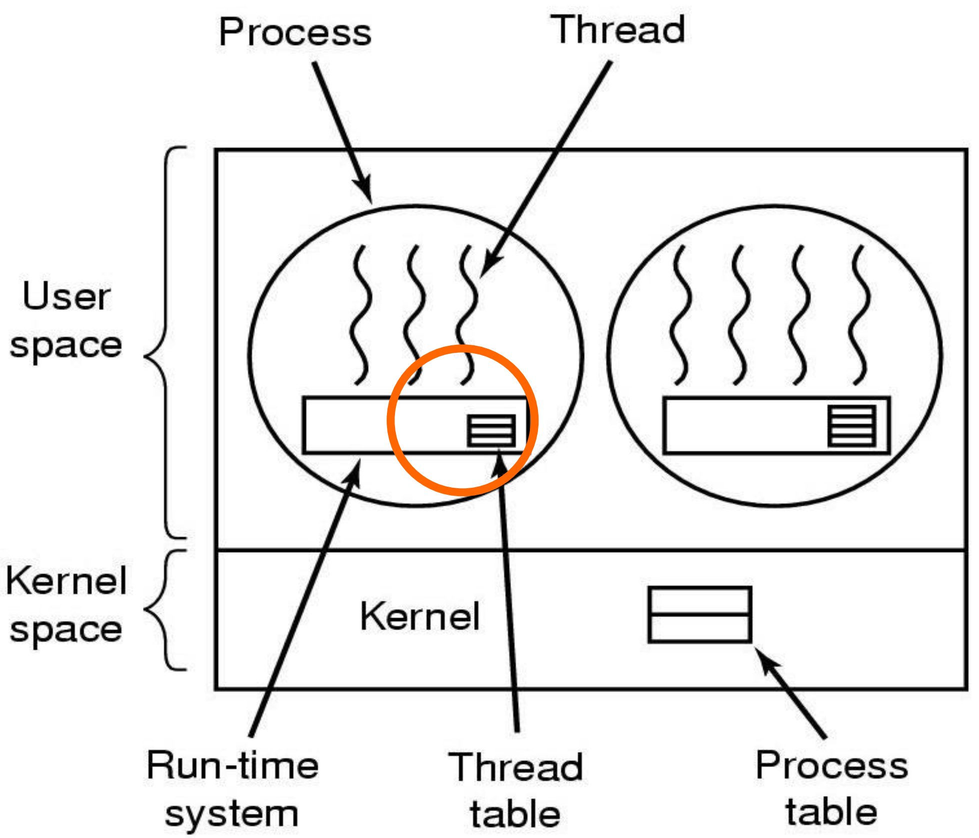
User-Level

- 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

In kernel-level threading, the OS kernel is aware of each thread.

The kernel is responsible for:

- Creating threads
- Scheduling them
- Managing context switches
- Handling synchronization and blocking

Processes are heavy because they require:

- Separate memory space
- Dedicated resources
- Full context switch (memory, file descriptors, etc.)

Kernel threads are lighter:

- They share the same address space and resources within the same process.
- But still, they're **heavier than user-level** threads due to kernel involvement in every operation.

In kernel threads, even basic operations like:Creating a thread, Destroying a thread,Blocking or unblocking a thread and Context switching must go through **system calls** (e.g., pthread_create, pthread_join).

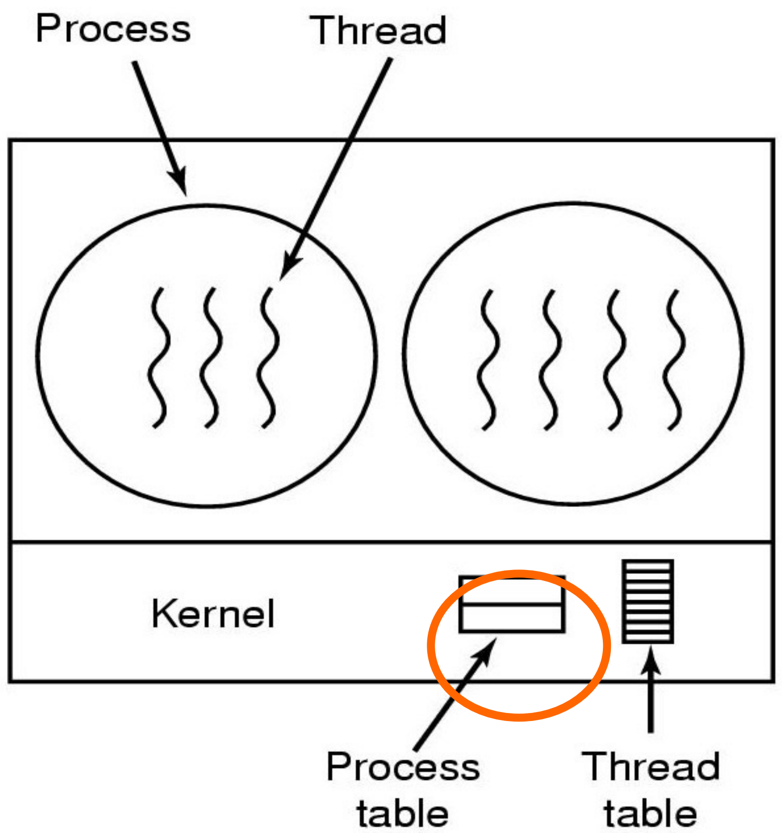
The kernel enforces strict security and protection boundaries:

- It validates **parameters** in system calls (e.g., pointers, buffer lengths)
- It checks **permissions and rights** before accessing resources
- It performs **error checking to prevent malicious or buggy behavior**

This is necessary because the kernel **must maintain system stability and security,** but it also introduces **Additional CPU cycles spent in validation** and **Performance overhead** especially when done frequently

Feature	**User-Level Threads**	**Kernel-Level Threads**
Creation	Fast (no system call)	Slower (system call required)
Context Switch	Fast (user-mode only)	Slower (user ↔ kernel mode)
Scheduling	Done in user-space	Done by kernel (less flexible)
Blocking (e.g., I/O)	Blocks entire process	Only blocks that thread
CPU utilization (multi-core)	Can’t use multiple cores	Can use multiple cores

Implementing Threads in the Kernel

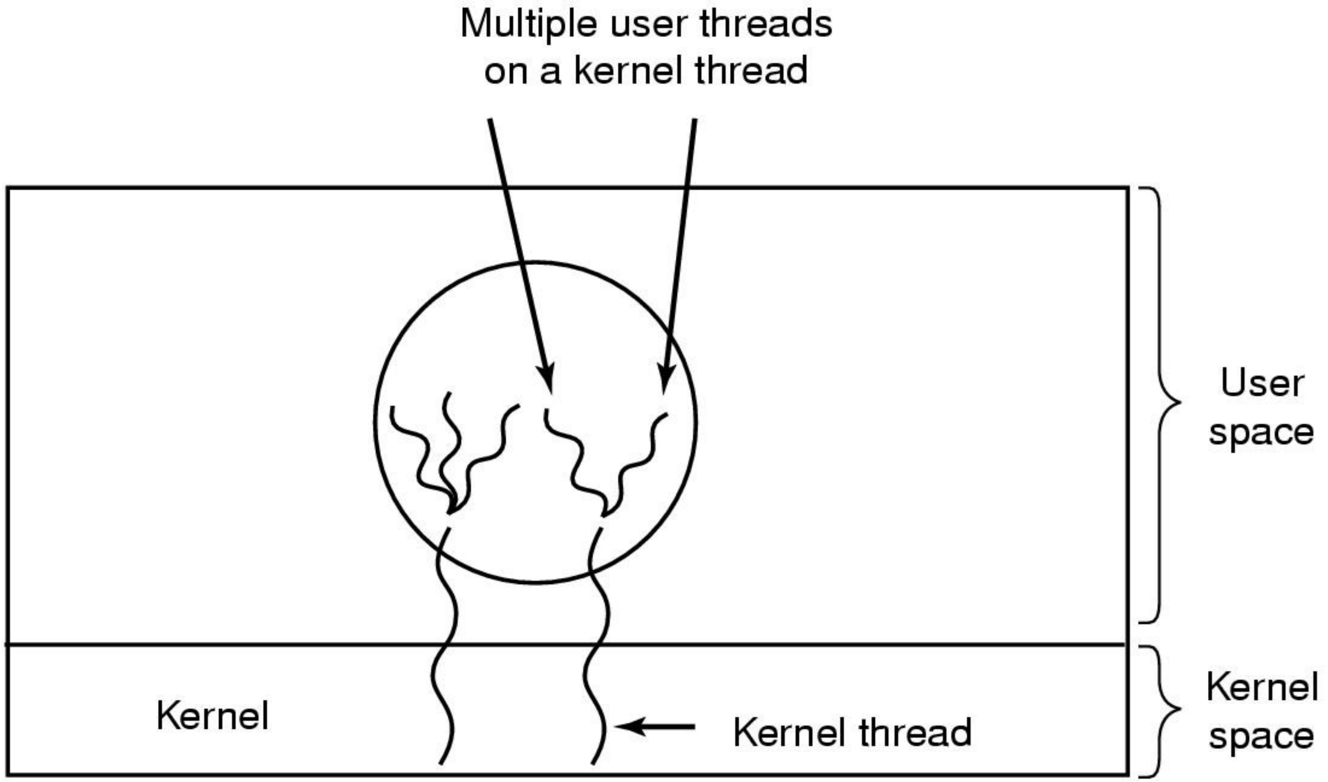


A thread package managed by the kernel

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

Thanks ☺