Part 2: Processes and Threads

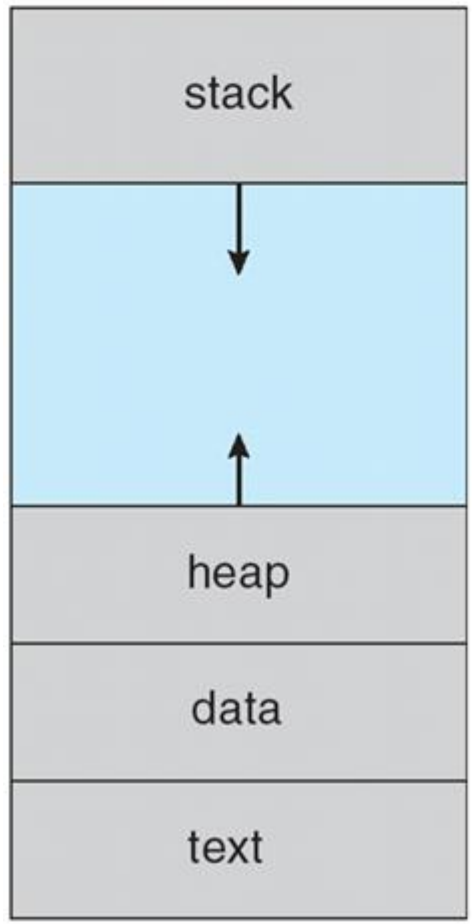
- Process Concept
- Process Scheduling
- Operation on Processes
- Cooperating Processes
- Interprocess Communication
- Threads

Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-sharing system – user programs & commands
- Textbook uses the terms *job* and *process* almost interchangeably (Job = Process).
- Process – a program in execution; process execution must progress in sequential fashion.

Process in Memory

Base + Limit



(parameters and local variables in a function)

(dynamically allocated variables, e.g., a=new int[10])

(global parameters)

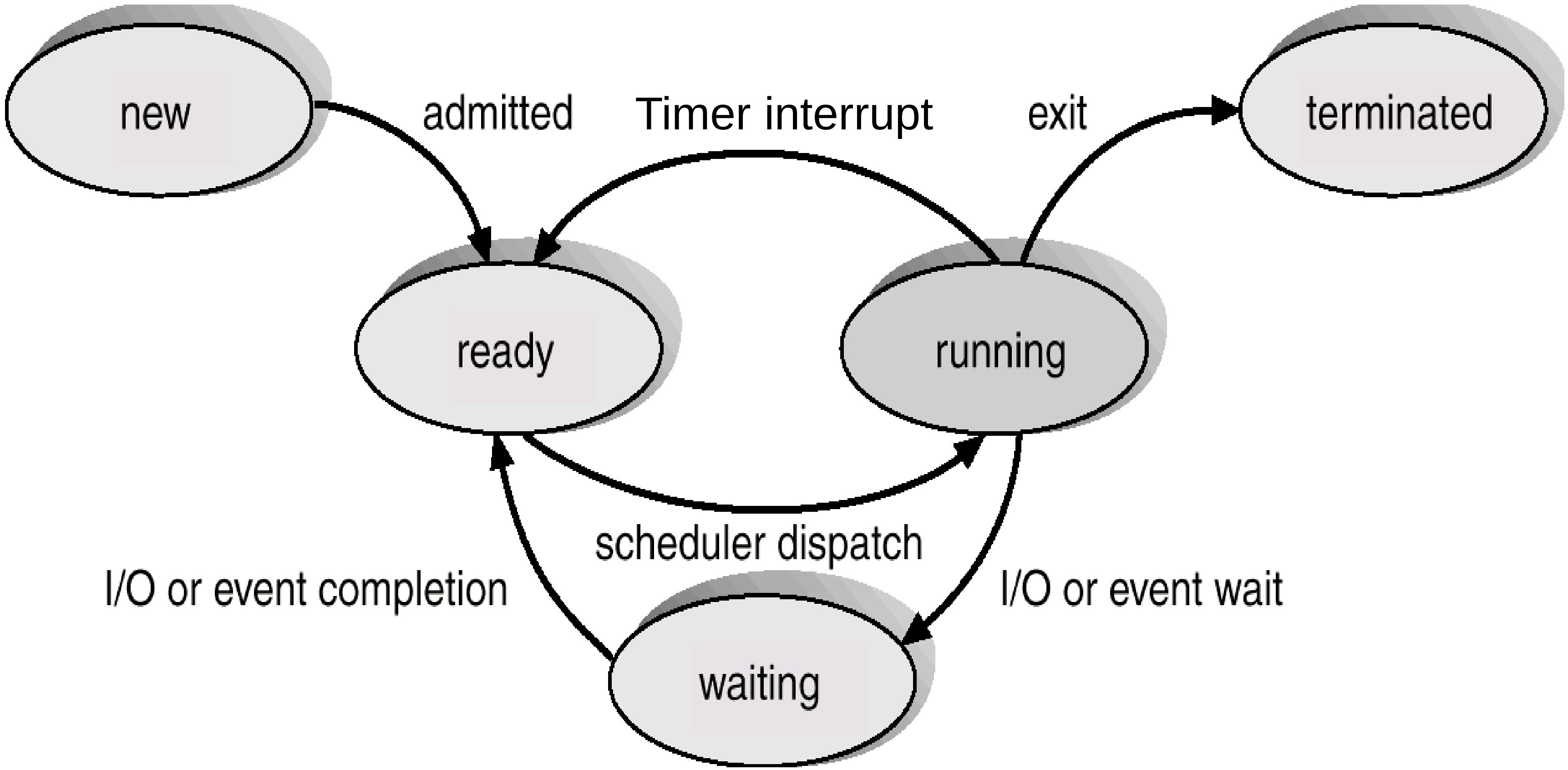
(code section, program)

Base

Process State

- As a process executes, it changes *state*
  - new: The process is being created.
  - running: Instructions are being executed.
  - waiting: The process is waiting for some event to occur.
  - ready: The process is waiting to be assigned to CPU.
  - terminated: The process has finished execution.

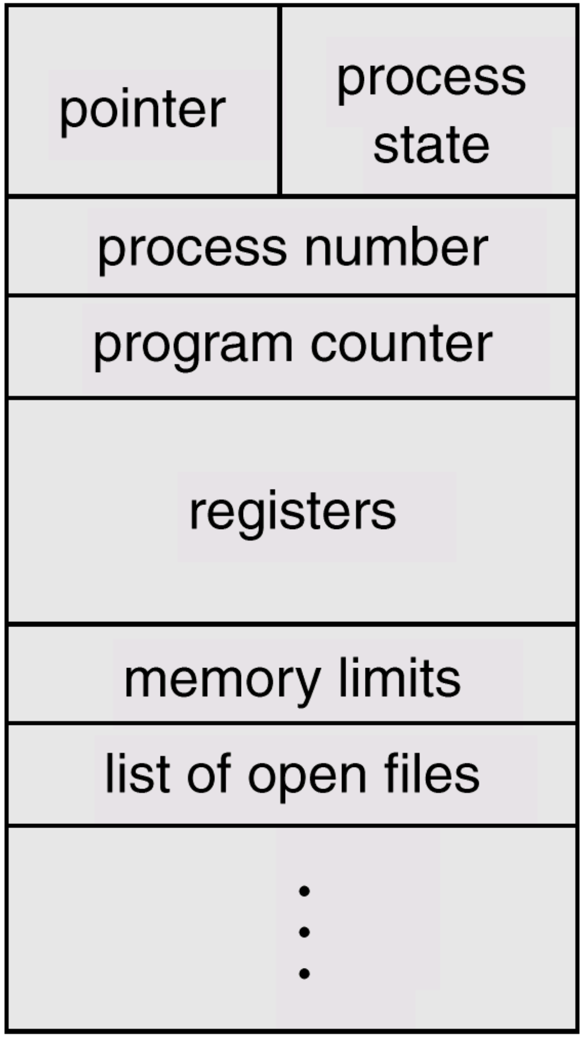
Diagram of Process State



Process Control Block (PCB)

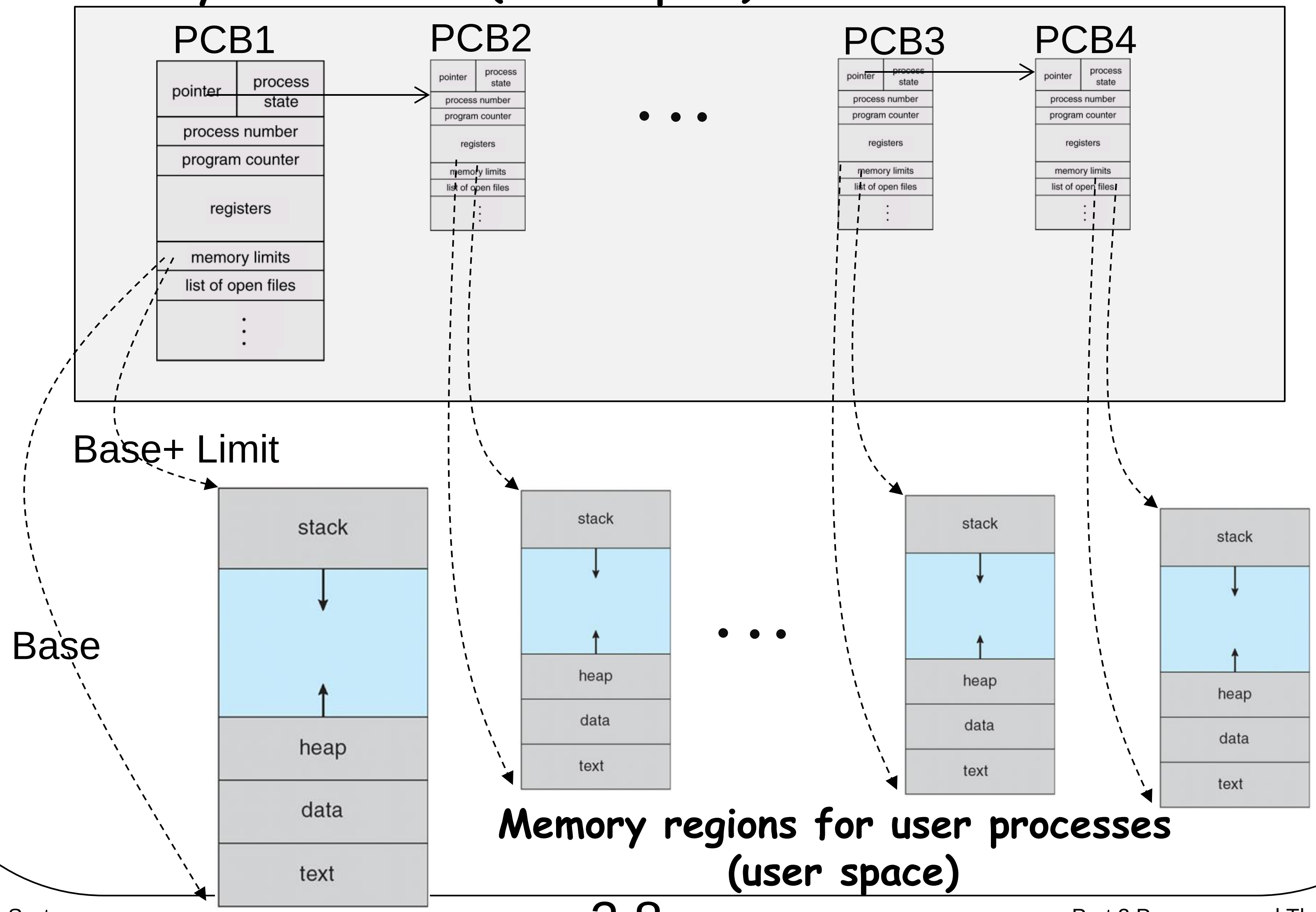
- Keep the information associated with each process:
  - Pointer
  - Process state
  - Process number (process id)
  - Program counter (pointer to the next instruction)
  - CPU registers
  - Memory management information (e.g., memory limit)
  - Information regarding open files (e.g., list of open files)
- In multi-programmed systems, each process has its PCB stored in the main memory.

Process Control Block (PCB)



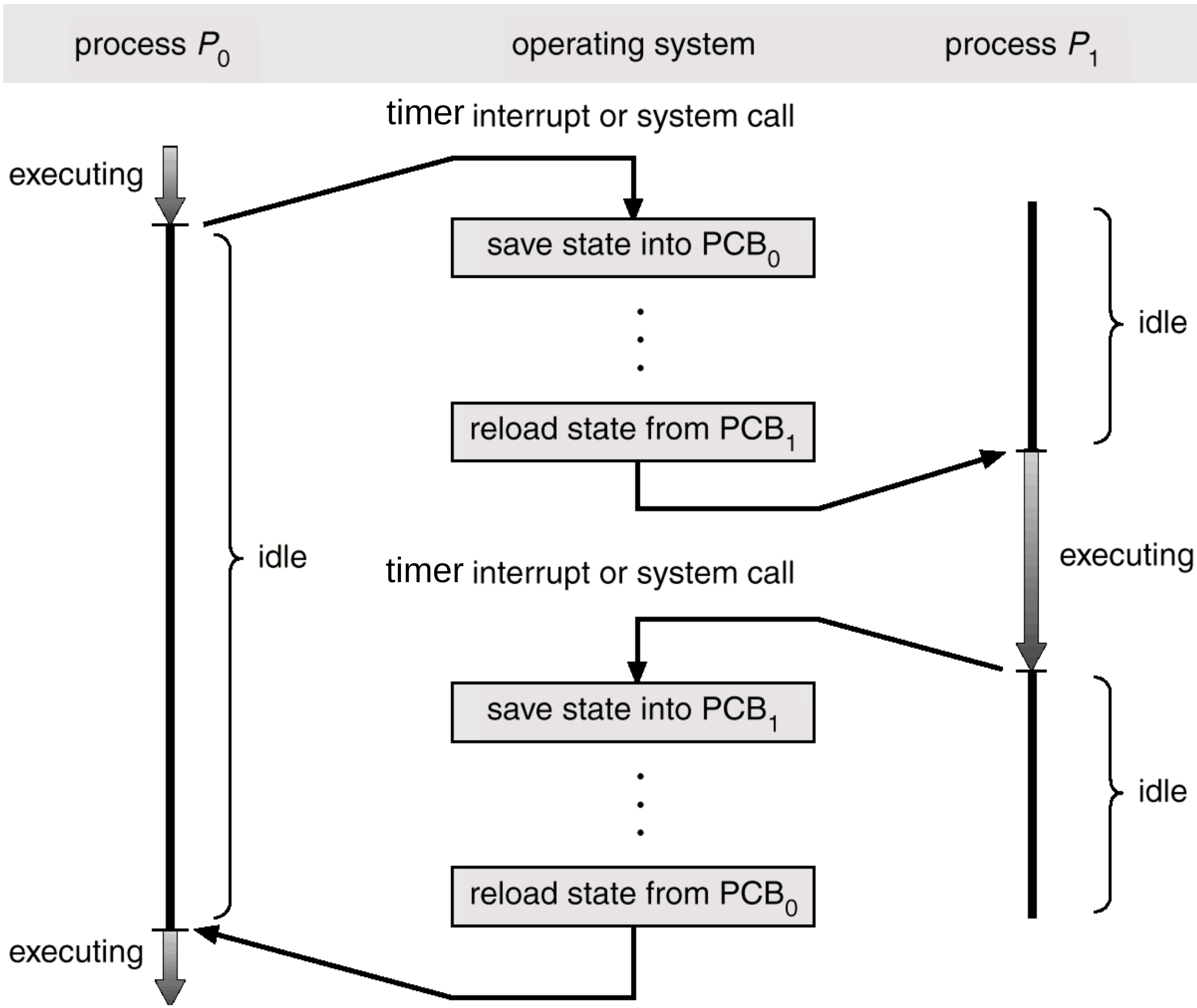
PCB and Processes in Main Memory

Memory area for OS (kernel space)



Memory regions for user processes (user space)

CPU Switch From Process to Process



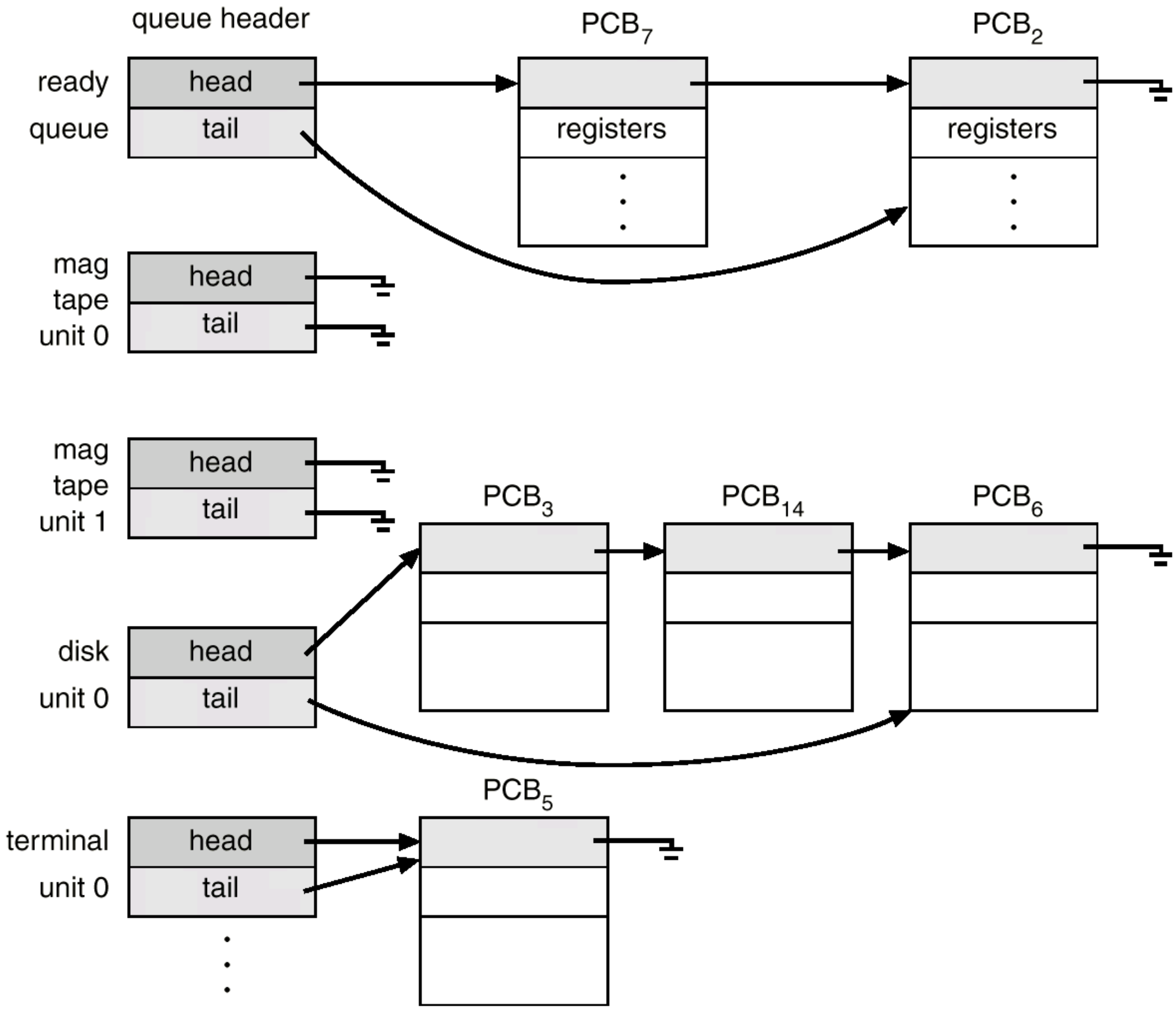
Context Switch

- When CPU switches to another process, the system must save the context (i.e. information) of the old process and load the saved context for the new process.
- Context-switch time is overhead; the system does no useful work while switching.

Process Scheduling Queues

- Job queue – set of all processes *with the same state* in the system.
- Ready queue – set of all processes residing in main memory, ready and waiting the CPU to execute.
- Device queues – set of processes waiting for an I/O device.
- Process migration between the various queues when process state changes.
- All the queues are stored in the main memory (kernel space).

Ready Queue And Various I/O Device Queues



Schedulers

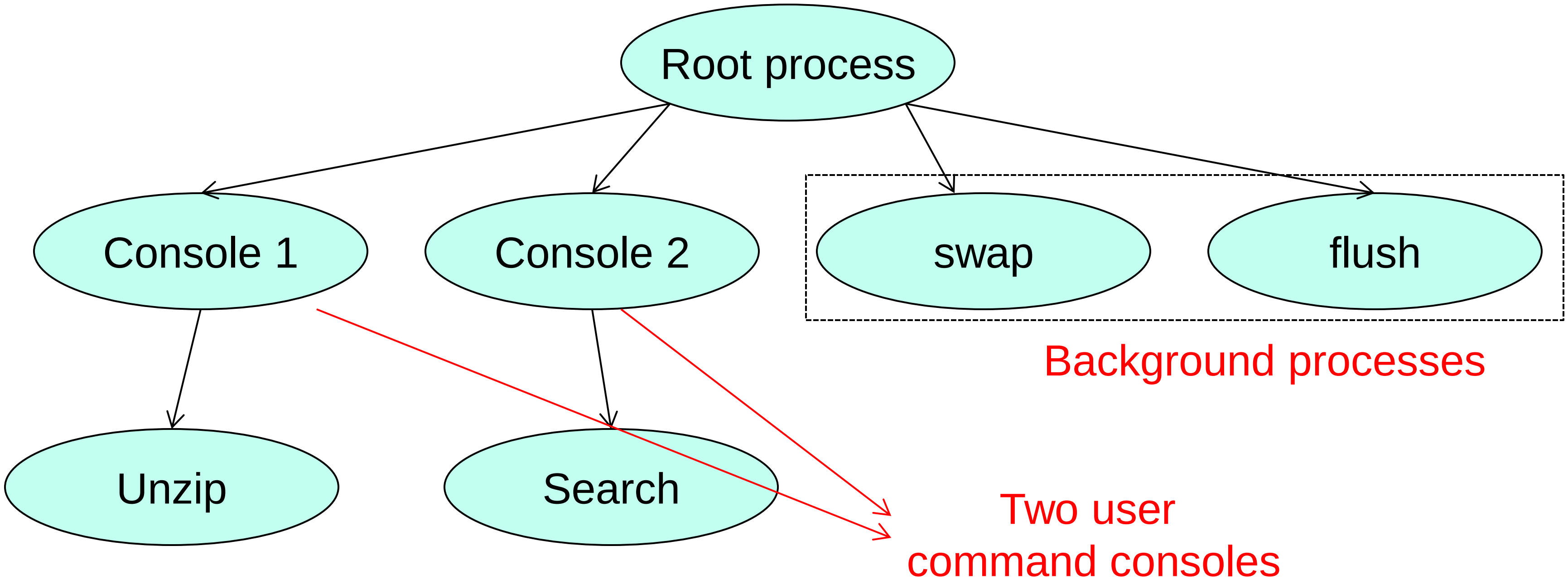
- We need multiple schedulers for different purposes.
- Long-term scheduler (or job scheduler) – selects processes from disk and loads them into memory for execution.
- Short-term scheduler (or CPU scheduler) – selects from among the processes that are ready to execute, and allocate the CPU to one of them.
- Medium-term scheduler
  - When the system load is heavy, swap out a partially executed process from memory to hard disk.
  - When the system load is lighter, such processes can be swapped in.

Schedulers (Cont.)

- Short-term scheduler is invoked at least once in every 100 milliseconds because in average a process executes for 100 milliseconds before I/O. It must be fast in making decision.
- Long-term scheduler is invoked very infrequently (seconds, minutes). May be slow in making decision.
- The long-term scheduler controls the *degree of multiprogramming* initially. Then medium-term scheduler after that.

Process Creation

- Parent process creates children processes, which, in turn create other processes, forming a tree of processes (**fork**).



Process Creation (Cont')

- Execution
  - Parent and children execute concurrently.
  - Parent waits until children terminate (**wait**).
- Examples
  - Many Web browsers nowadays folk a new process when we access a new page in a "tab".
  - OS may create background processes for monitoring and maintenance tasks.

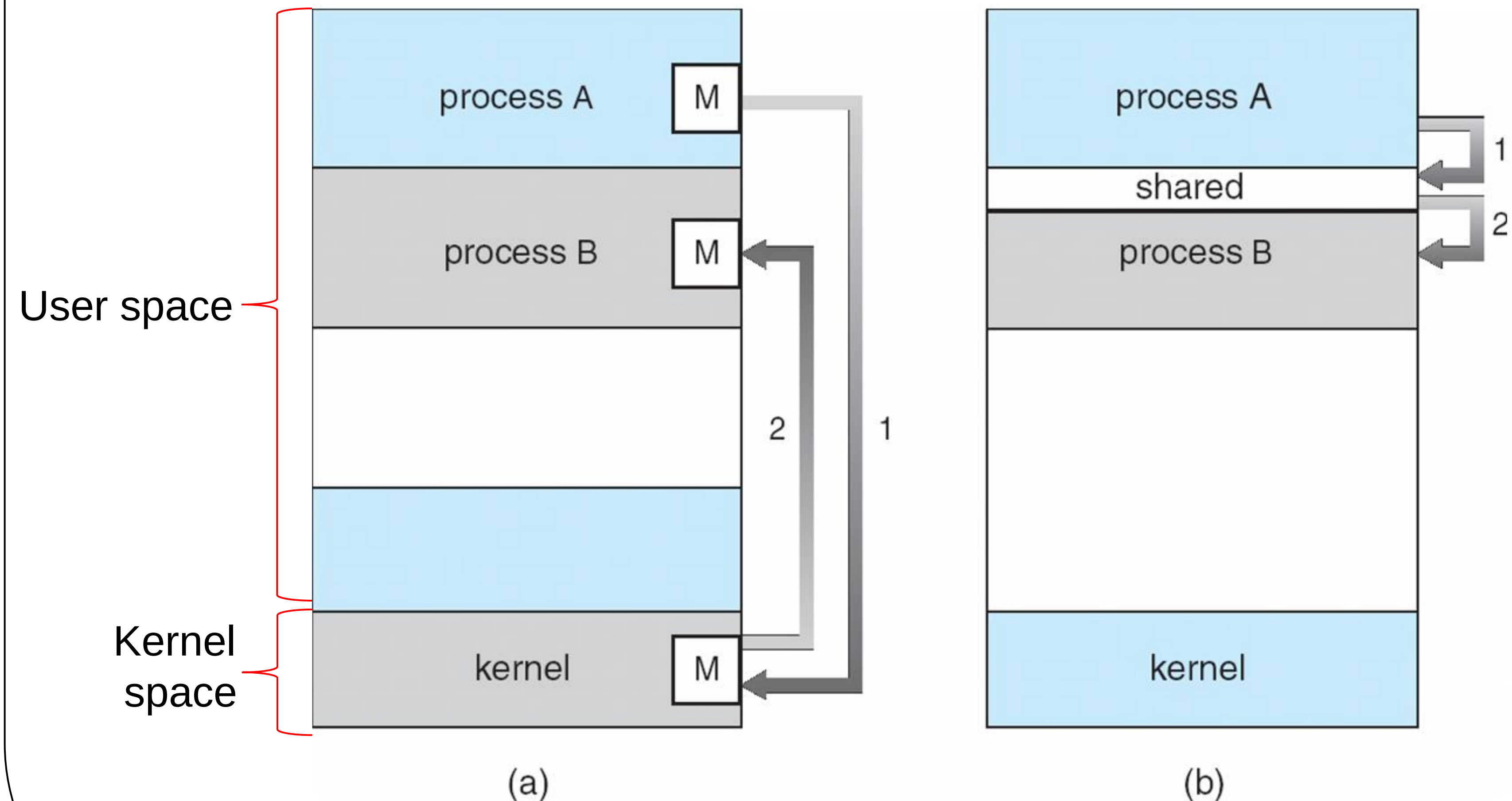
Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**). At that point:
  - A child may output return data to its parent.
  - Process' resources are deallocated by operating system.
- Parent may terminate execution of children processes (**abort**).
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - Parent is exiting.

Cooperating Processes

- *Independent* process cannot affect or be affected by the execution of another process.
- *Cooperating* process can affect or be affected by the execution of another process
- Two models of Interprocess Communication (IPC)
  – Message passing
  – Shared memory

Communication Models



Message passing

Shared memory

IPC – Message Passing

- **Message Passing** – processes communicate and synchronize their actions without resorting to shared variables.
- IPC facility provides two operations:
  – **send**(*message*) – message size fixed or variable
  – **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  – establish a *communication link* between them
  – exchange messages via send/receive

Direct vs. Indirect Communication

- **Direct communication:** Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from process Q
- **Indirect communication:** Messages are sent to or received from mailboxes (also referred to as ports).
  - Mailbox is an object into which messages are placed and removed (like a queue).
  - Each mailbox has a unique id.
  - Primitives are:
    - ❄ **send** (*A, message*): send a message to mailbox *A*
    - ❄ **receive** (*A, message*): receive a message from mailbox *A*

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process.
  - *unbounded-buffer* places no practical limit on the size of the buffer.
  - *bounded-buffer* assumes that there is a fixed buffer size.

Example: Producer & Consumer

//declare a *mailbox* with capacity of B messages

void *producer*(void)
{
  message *m*;
  while (1) {
    //pre-processing…
    while(*mailbox* is full) wait;
    send(*mailbox*, m);
  }
}

Does this example use direct or indirect communication?

void *consumer*(void)
{
  message *m*;
  while (1) {
    while(*mailbox* is empty) wait;
    receive(*mailbox*, m);
    //post-processing…
  }
}

Threads

- This part is for **self-learning**.
- A lecture video will be uploaded to NTULearn.
- This slide and the later slides in this chapter are <u>**not**</u> examinable.
- About Nachos Labs
  - Nachos uses the term “thread”.
  - All concepts and mechanisms that we learnt about processes can be applicable to Nachos threads.
    - ❄ Thread control block vs. Process control block.
    - ❄ Thread state vs. Process state.
    - ❄ System calls: fork, exit etc.

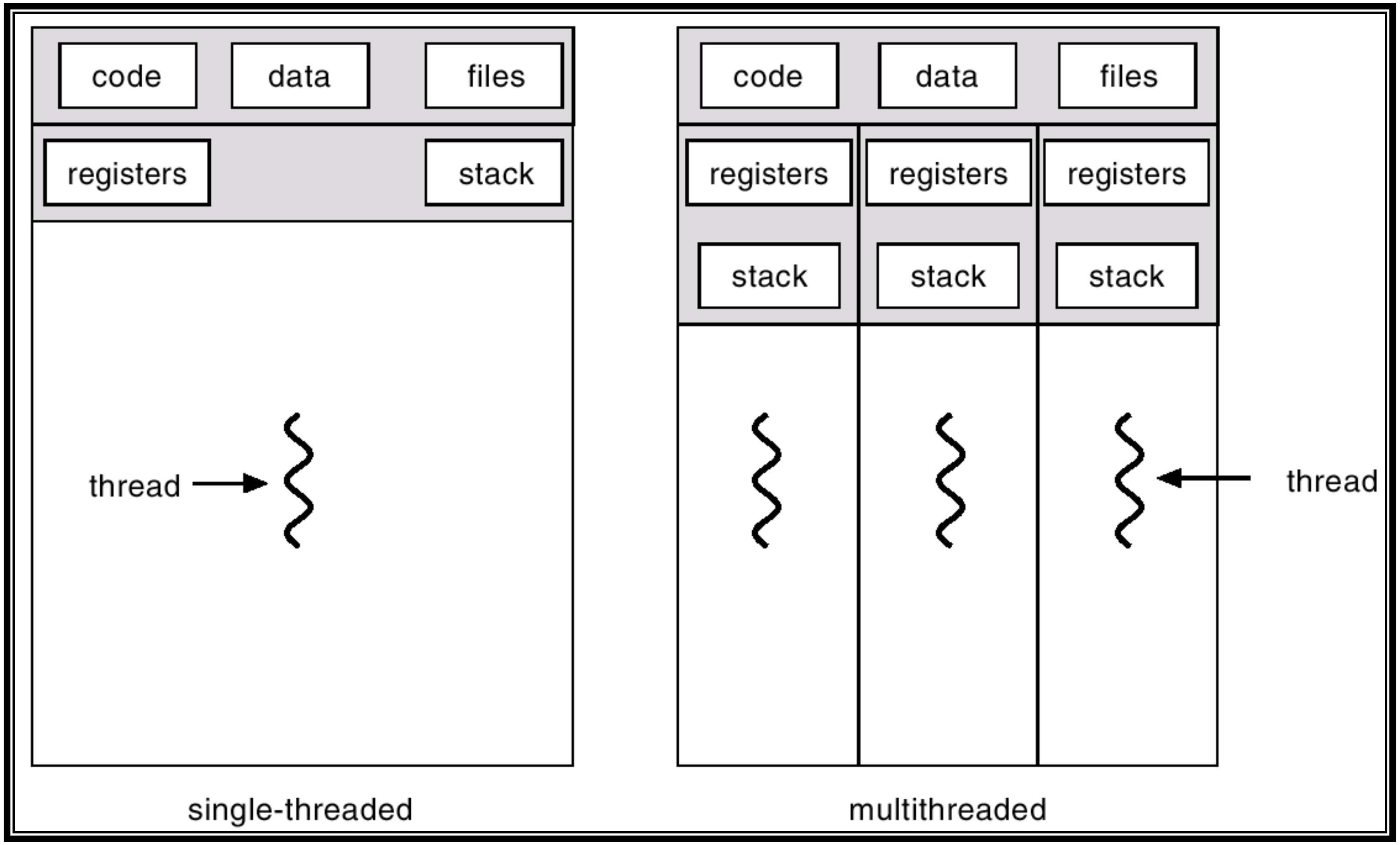
Threads

- Overview
- Single vs Multithreading Process
- Benefits of threads
- Types of threads
- Multithreading models

Overview

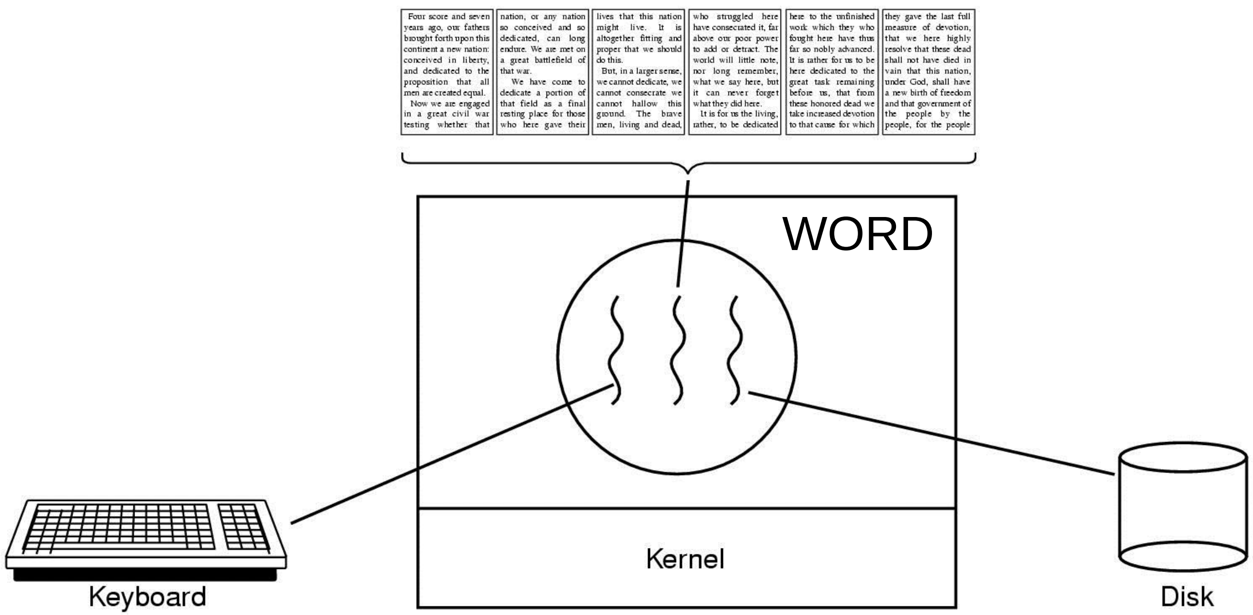
- A *thread* (or *lightweight process*) is a basic unit of CPU utilization; it consists of:
  - a thread id
  - program counter
  - register set
  - stack space
- A thread shares with its peer threads in the same process its:
  - code section
  - data section
  - operating-system resources
- A traditional or *heavyweight* process is an executing program with a single thread of control

Single vs Multithreaded Processes

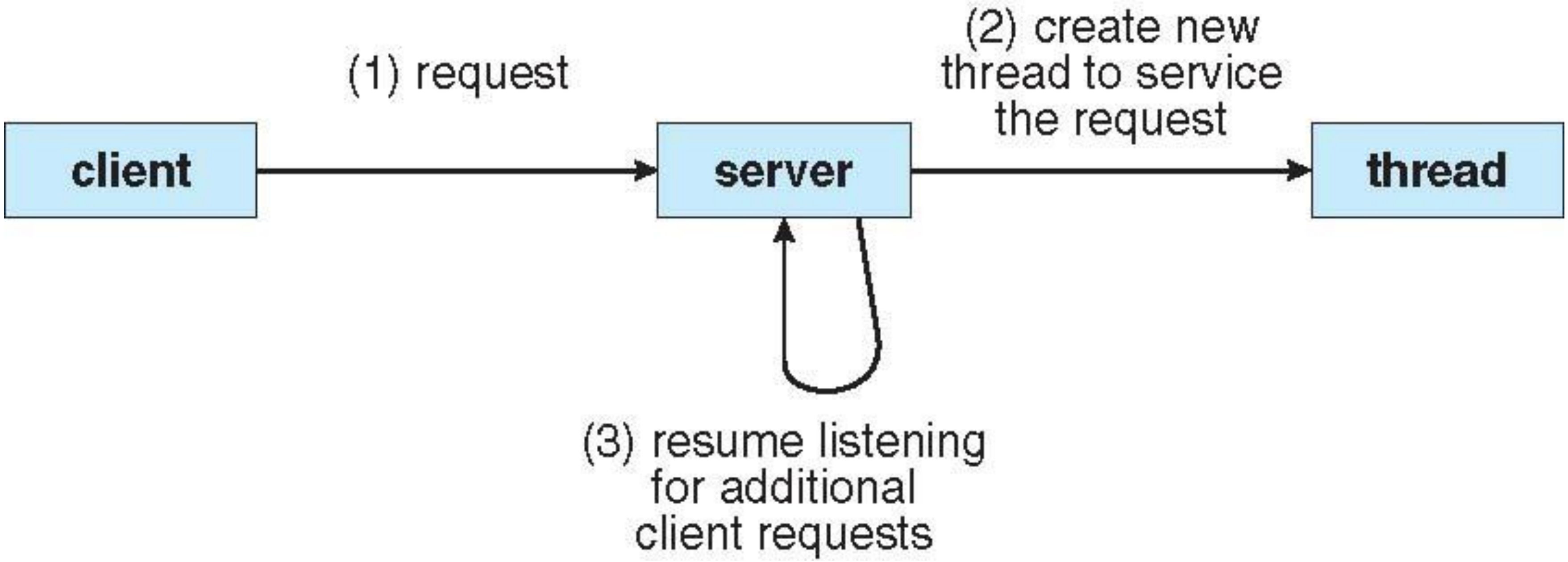


Advantage of Multi-threaded Process

- In a multi-threaded process, while one thread is blocked and waiting, a second thread can run.
  - Cooperation of multiple threads in same process confers higher throughput and improved performance.



Example: Multithreaded Server



Two types of threads

- Paradox
  - Allow users to implement an arbitrary number of threads, V.S.,
  - OS kernel can support a limited number of threads due to resource constraints.
- Solution
  - Two layers of abstraction:
    - User threads (logical threads): Supported at the user level; Allow users to create as many threads as they want.
    - Kernel threads (physical threads): Supported at the Kernel level; Slower to create and manage than that of user thread; Resources are eventually allocated in kernel threads.
  - OS maintains the mapping from user threads to kernel threads (Multithreading models).

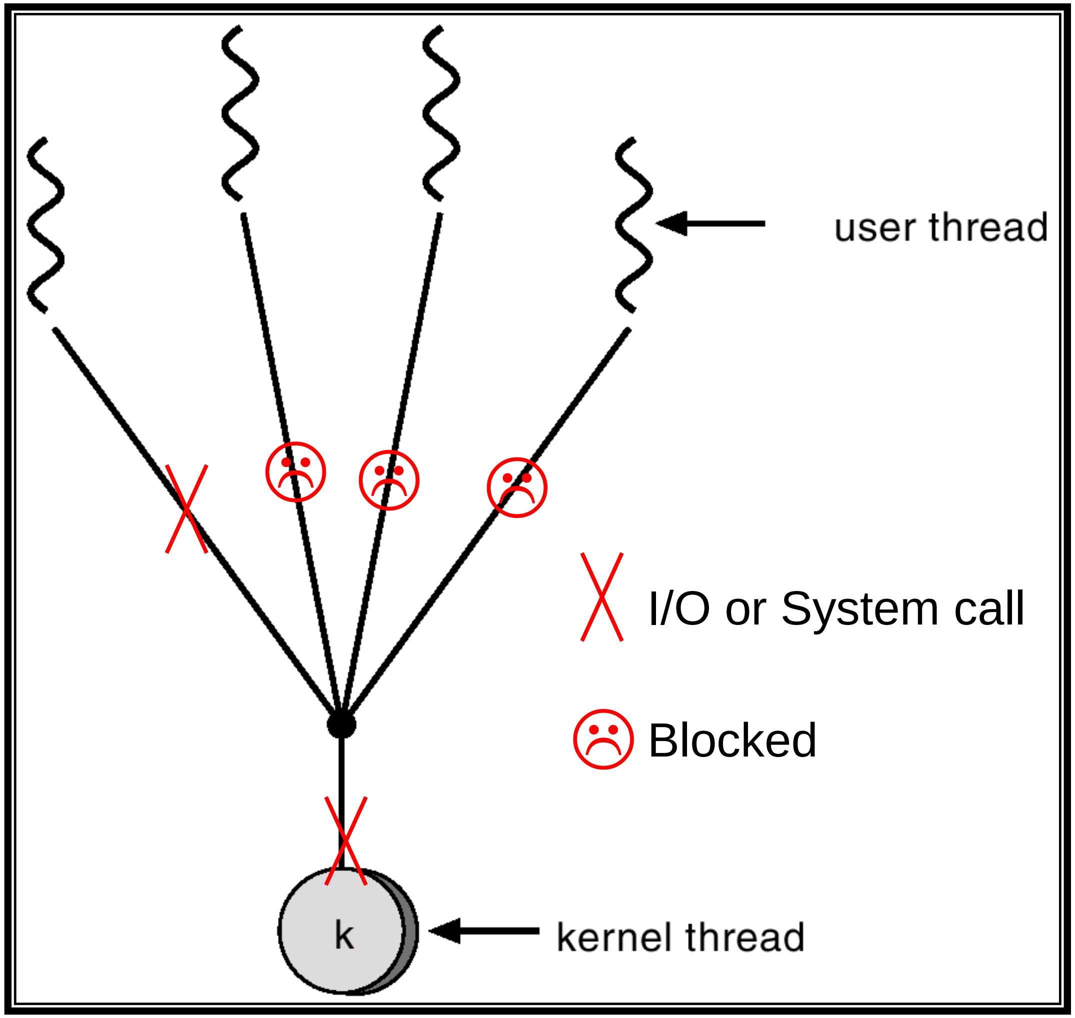
Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

- Many user-level threads mapped to single kernel thread.
- Disadvantage: Unable to run on multiprocessor due to single kernel thread (For those threads mapped to the same kernel thread)

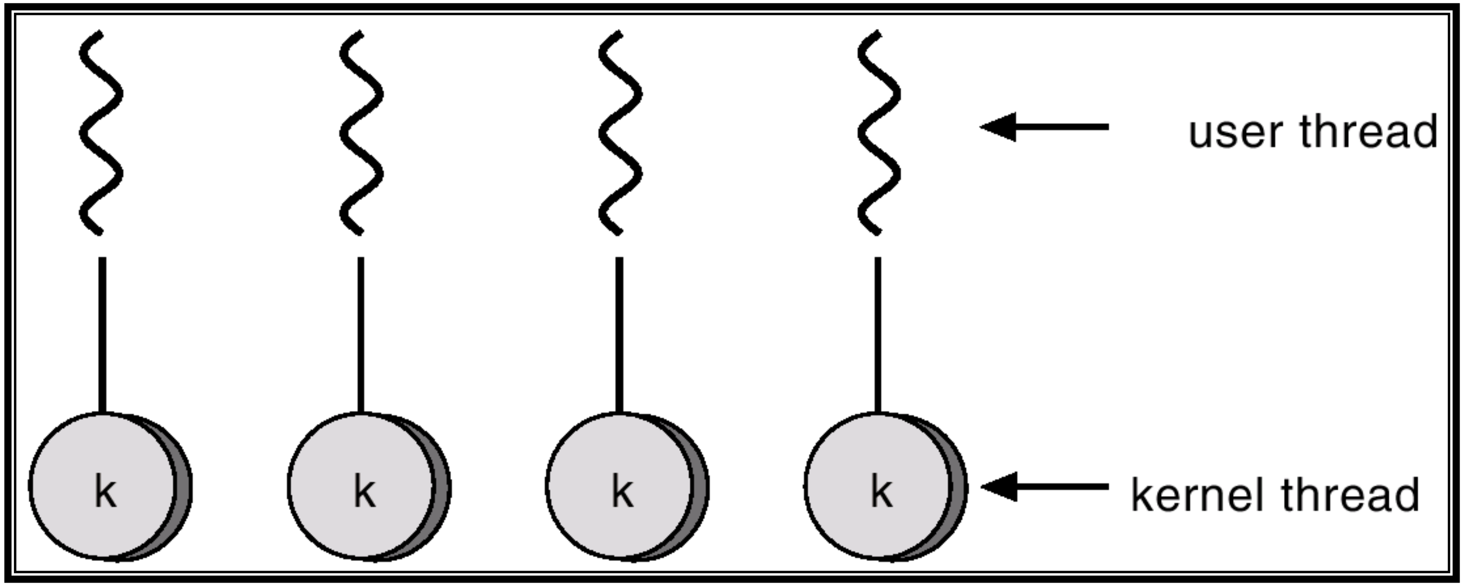
Many-to-One



One-to-One

- Each user-level thread maps to kernel thread.
- Examples: Windows 95/98/NT/2000, OS/2
- Provide more concurrency than the many-to-one model
- Disadvantage:
  - Creating a user thread requires creating the corresponding kernel thread
  - May create too many kernel threads which is a burden to the system (if not so many processors in the system)

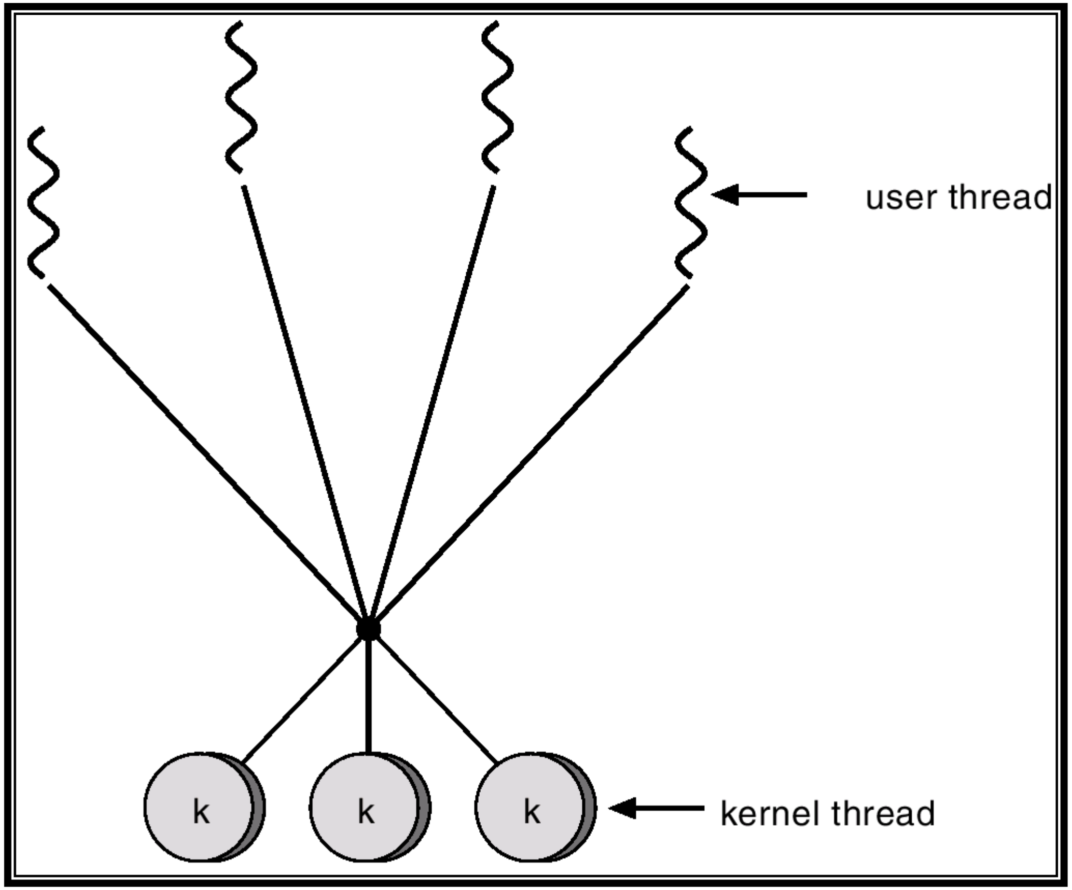
One-to-one



Many-to-Many

- Allows many user level threads to be mapped to many kernel threads.
- Allows the operating system to create a sufficient number of kernel threads.
- Examples: Solaris 2, Windows NT/2000
- Do not have the diadvantages of one-to-one and many-to-one model.

Many-to-Many



Advanced Readings

- Beej's Guide to Unix Interprocess Communication, (pdf in EdveNTUre)
  - http://beej.us/guide/bgipc/
  - Very good introduction on the IPC implementation inside Unix.
- Tutorials on how to use thread libraries:
  - POSIX pthreads, http://randu.org/tutorials/threads/
  - Windows Process and Thread Functions. http://msdn.microsoft.com/en-us/library/windows/desktop/ms684847(v=vs.85).aspx