Concurrency

Week 3

Threads

Topics

- **Recap**: Concurrency / Parallelism, Processes
- **Threads**
- **Process/Thread: Memory**
- **Multi-threaded programs**
- **Communication models**

Learning Outcomes

- At the end of this lesson, you will be able to:
  - Describe the concept of a thread and different ways of implementing them.
  - Identify the main application areas of threads
  - Implement a simple multi-threaded program

Recap: Some Key Definitions

- Multitasking: **Simultaneous** execution of **tasks**
- Parallelism: Using **multiple processing units** to execute multiple tasks at the same time
- Concurrency: Simultaneous execution of tasks which may **share resources**

What is the Main Idea of the Course?

- The following four questions summarize this course.
- Questions:
  1. ***How can we run multiple tasks simultaneously***? We discuss multitasking and multi-threading to answer this question
  2. ***How can simultaneous tasks share resources***? We discuss resource types and how they can be shared
  3. ***What problems may we have when we share resources***? We discuss race conditions and inconsistency of data to answer this question
  4. ***What solution do we have for these problems***? We discuss semaphores, and mutex locks to answer this question

Part One!

Part One!

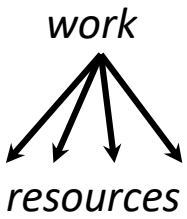
- In this part, we will discuss what a thread means and how it is created. Try to find answers to the following questions:

1. What is the difference between a process and a thread?
2. What are the two types of threads?
3. What resources are shared between the threads of a process?
4. What is used privately by each thread?
5. What multi-threading models are used?

Parallelism vs. Concurrency

Parallelism:

Use extra resources to solve a problem faster



Concurrency:

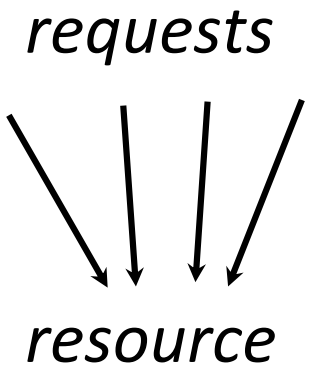
Correctly and efficiently manage access to shared resources

These concepts are related but still different:

- Common to use threads for both
- If parallel computations need access to shared resources, then the concurrency needs to be managed

Concurrency

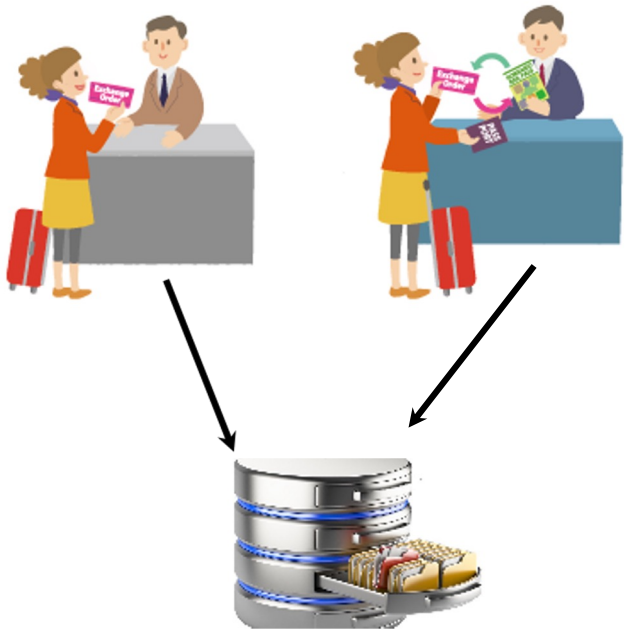
- Concurrency: Correctly and efficiently manage access to shared resources (from multiple possibly-simultaneous clients)



Concurrency Example: Shared Database

A database is used for airline ticket reservation

Multiple sales agents may access the database simultaneously

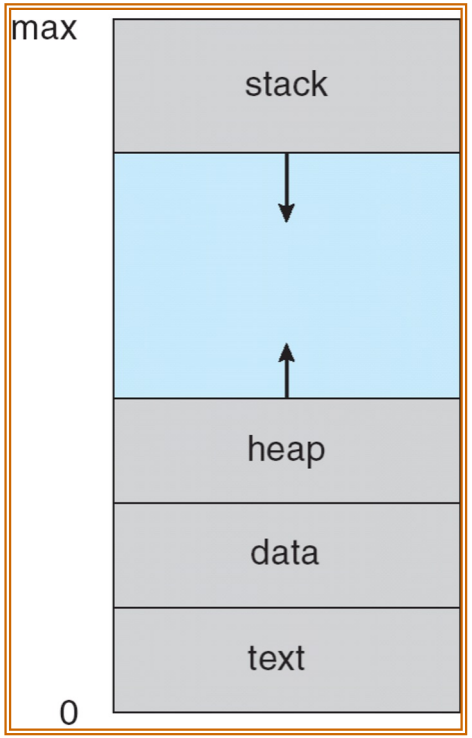


What is a Process? (recap)

- Process – a program in execution; process execution must progress in a sequential fashion
- To put a program into execution we should:
  - Copy it (at least partially!) into the main memory
  - Set the program counter to point to the first instruction
  - Initialize stack
  - Reset flags
  - The process is ready, start running …

How a Program is Put in Execution?

- To put a program in execution we should:
  - Copy it (at least partially!!) into the main memory
  - Set the program counter to point to the first instruction
  - Initialize stack
  - Reset flags
  - The process is ready, start running …
- Q: How does a process use heap and stack?



Independent tasks.

Question: Can we specify independent tasks within a process?

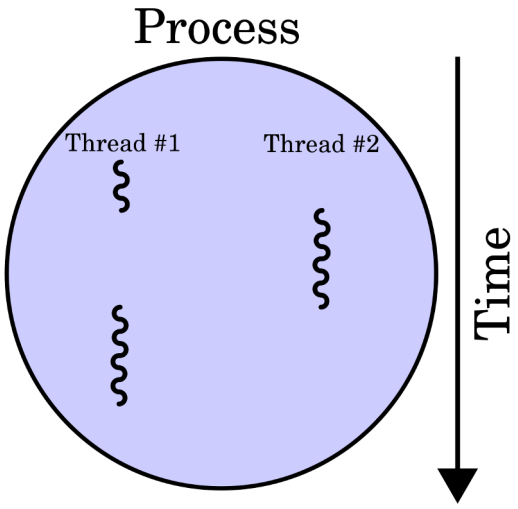
- We need a way to specify a set of program instructions from a larger code as an *independent* group of instructions

Threads

Threads

- If the cooperating processes share information for computation speed-up or modularity, they **can be parts of the same process**.

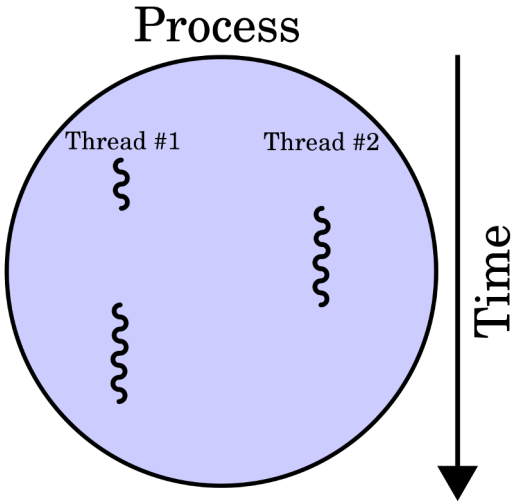
- Being part of the same process makes it possible to share:
  - the **code**
  - the **data**
  - the **resources**



Threads

- If the cooperating processes share information for computation speed-up or modularity, they can be part of a larger process.

- However, each one should have its own
  - Stack
  - Registers/flags
  - Program counter
- Each sub-part of a process is named a thread
- Watch here (https://www.youtube.com/watch?v=O3EyzlZxx3g).



Threads

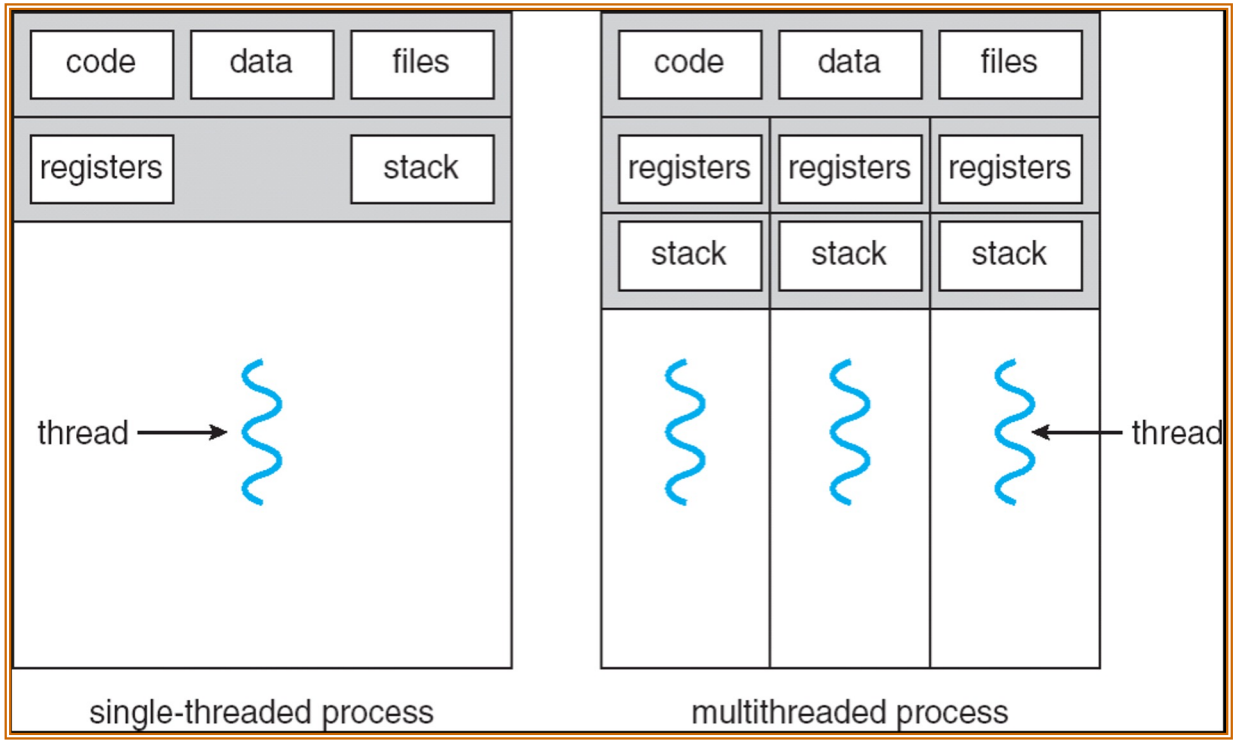
A thread is the smallest set of instructions that can be executed independently.

A thread:

- is an independent path of execution.
- can be managed independently by the scheduler.
- is a part of a process.
- has lighter state information (compare it with PCB).
- has cheaper context switching.

Single and Multithreaded Processes

- Benefits of Multithreading
  - Responsiveness
  - Easy Resource Sharing
  - Economy
  - (possible) Utilization of Multi-processor Architectures



Implementing Threads

- Threads are implemented in two ways (levels):
  - **User**-Level
  - **Kernel**-Level

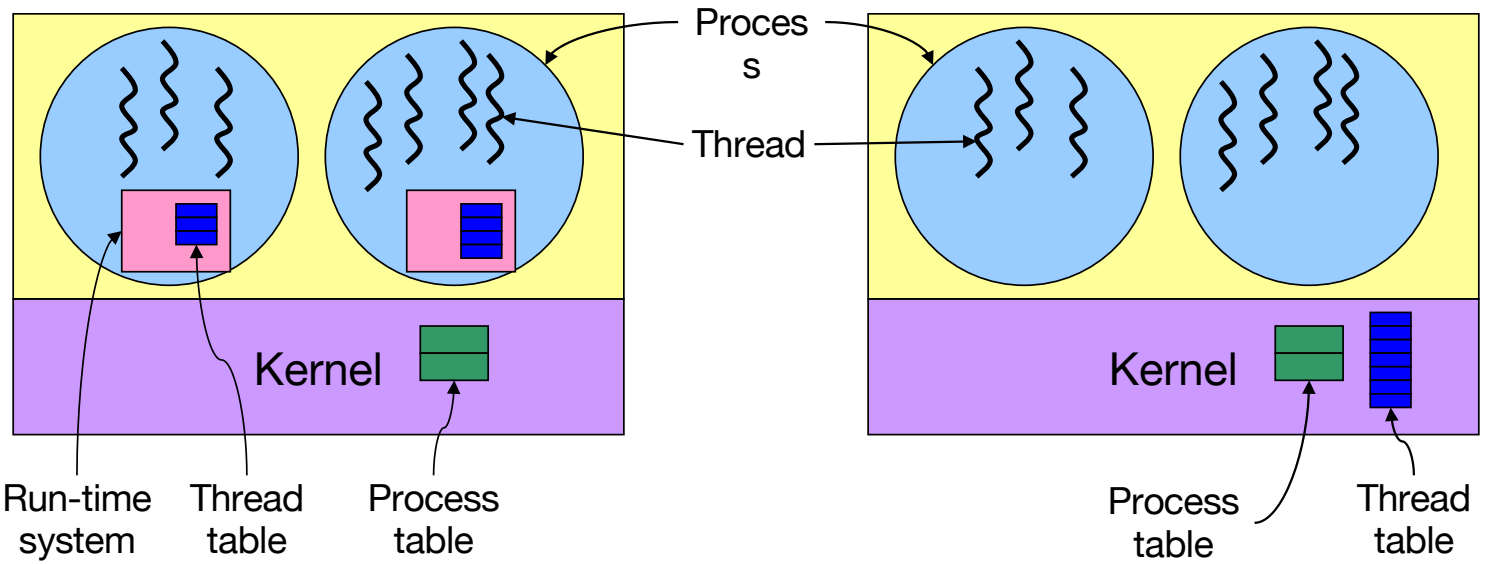
User-Level Threads *(so called also green-threads)*

- The user-level threads are implemented in user processes and the kernel is not aware of their existence.
- The kernel handles the multi-threaded process as a **single-threaded** process.
- User-level threads are **smaller** and **faster** (less overhead for context switching, being in the same memory chunk)
- The kernel is **not involved in the scheduling** of user-level threads.
- User-level threads are **OS-independent** (can be run on any operating system), however, the entire process is **blocked** if a thread performs a blocking operation.

Kernel-Level Threads

- Kernel-level threads and their context switching are managed by the operating system.
- Kernel-level threads are slower (in the creation) than user-level threads.
- Multiple kernel-level threads of the same process can be *potentially* scheduled on different processors.
- If a kernel-level thread is blocked, other threads of that process can continue running.

Implementing threads



User-level threads

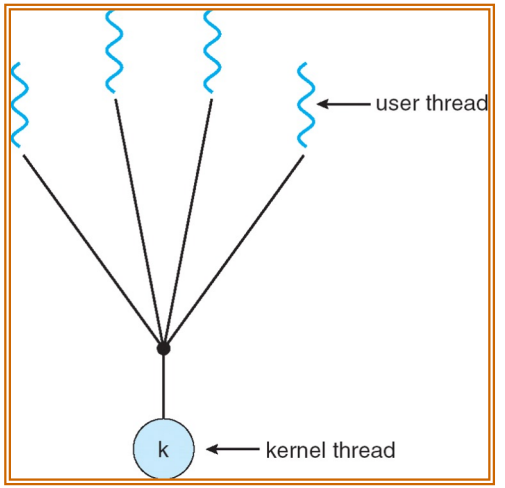
- No need for kernel support
- May be slower than kernel threads
- Harder to do non-blocking I/O

Kernel-level threads

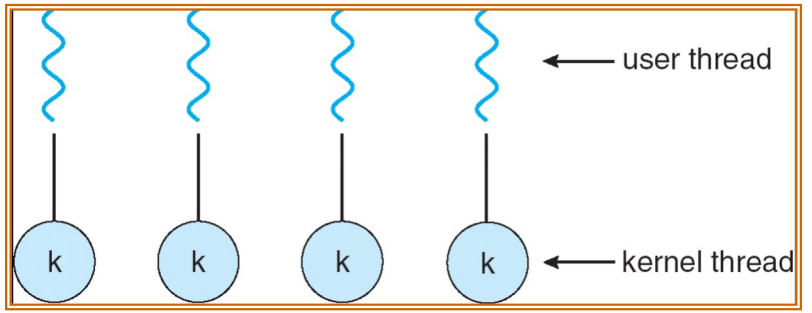
- More flexible scheduling
- Non-blocking I/O
- Not portable (system dependent)

Multithreading Models

- Many-to-One
  - Many user-level threads are mapped to a single kernel thread
  - Examples:
    - Solaris Green Threads
    - GNU Portable Threads (GNU PTh)



- One-to-One
  - Each user-level thread maps to a kernel thread
  - Examples
    - Windows NT/XP/2000
    - *NIX(Linux/MAC)
    - Solaris 9 and later
    - POSIX/ANSI-C threads



Part One!

- In this part, we will discuss what is meant by a thread and how it is created. Try to find an answer to the following questions:

1. What is the difference between a process and a thread?
2. What are the two types of threads?
3. What resources are shared between the threads of a process?
4. What is used privately by each thread?
5. What multi-threading models are used?

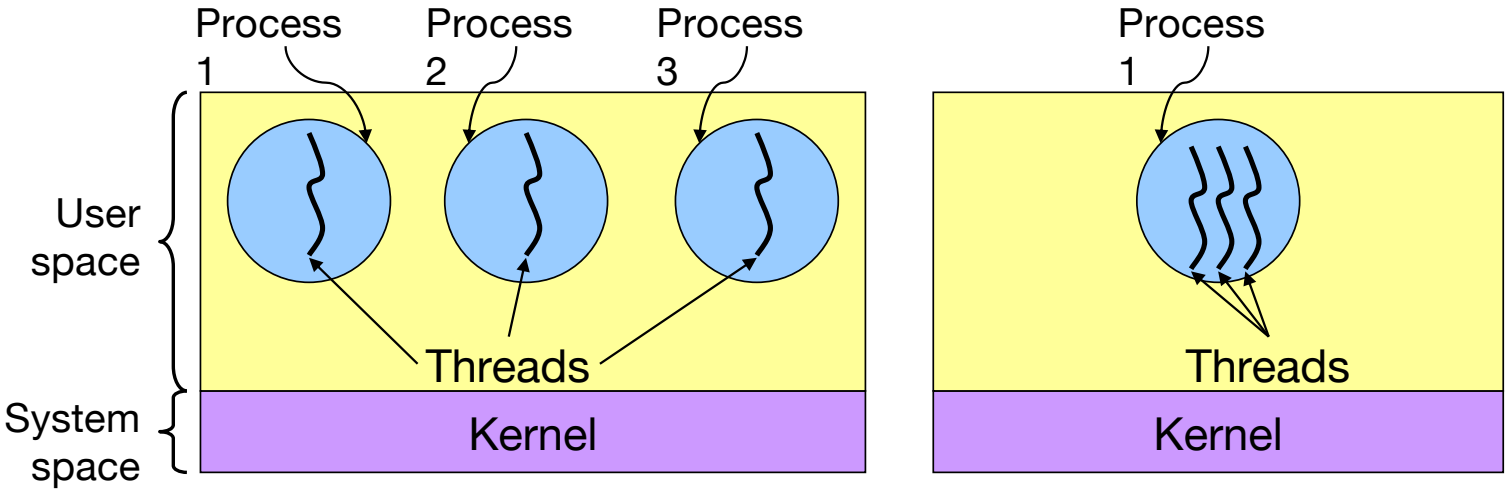
Part Two!

Part Two!

- In this part, we will discuss what a thread means and how it is created. Try to find an answer to the following questions:
  1. Why do we store less info per thread than per process?
  2. How can a process overlap I/O with computation?
  3. How are non-blocking system calls implemented in a multi-threaded model?
  4. How are non-blocking system calls implemented in a single-threaded model?
  5. Which multi-threading model is more suitable for web servers with a dispatcher-worker structure?

Threads as Processes Sharing Memory

- Processes can be viewed as address spaces
- Threads are viewed as program counter/stream of instructions
- Two examples
  - Three processes, each with one thread
  - One process with three threads

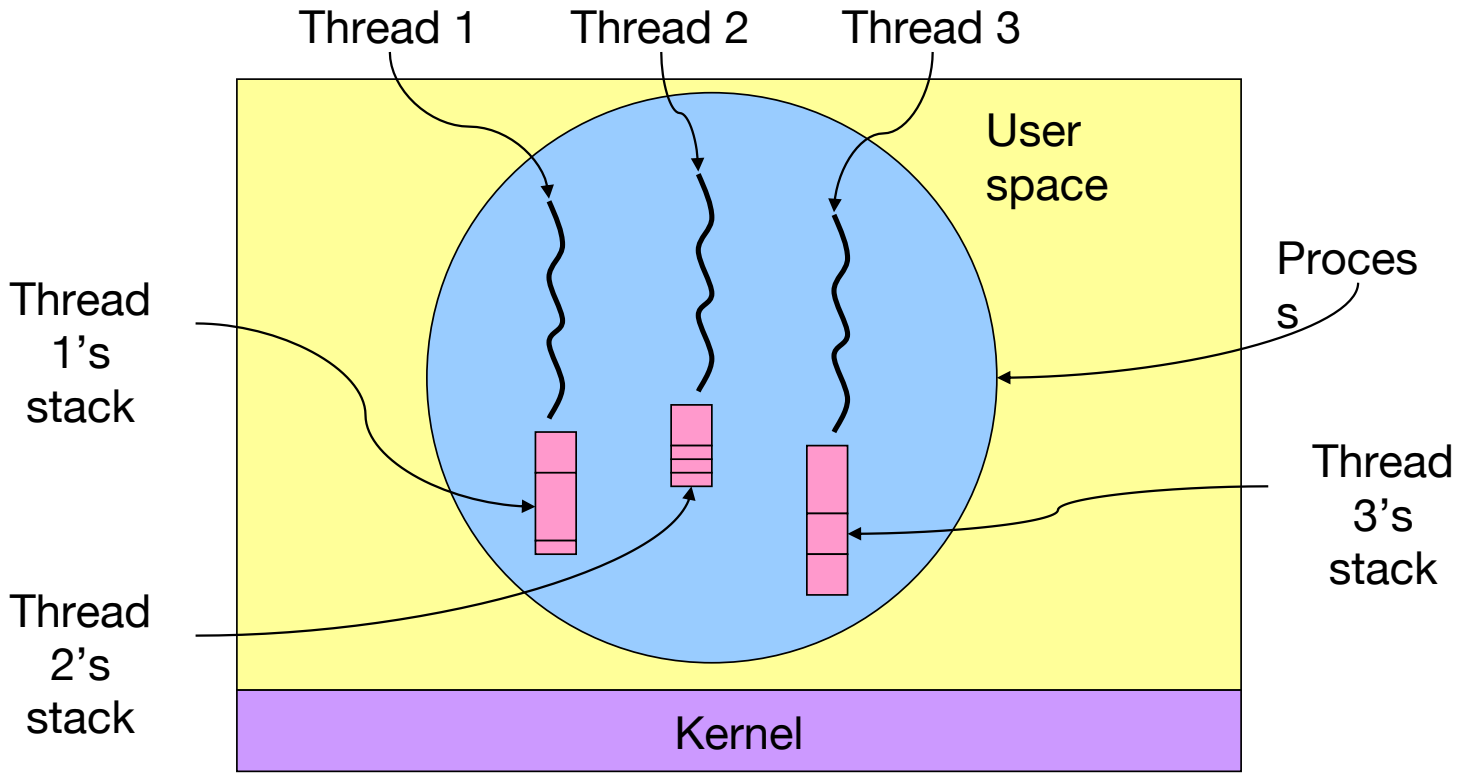


Process & Thread Information

Per process items
Address space
Open files
Child processes
Signals & handlers
Accounting info
Global variables

Per thread items	**Per thread items**	**Per thread items**
Program counter	Program counter	Program counter
Registers	Registers	Registers
Stack & stack pointer	Stack & stack pointer	Stack & stack pointer
State	State	State

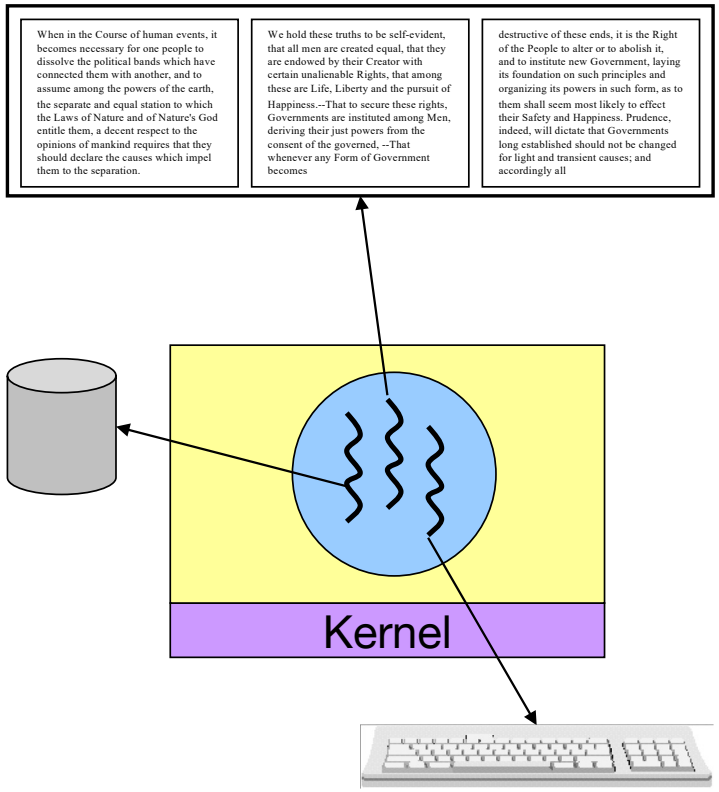
Threads & Stacks



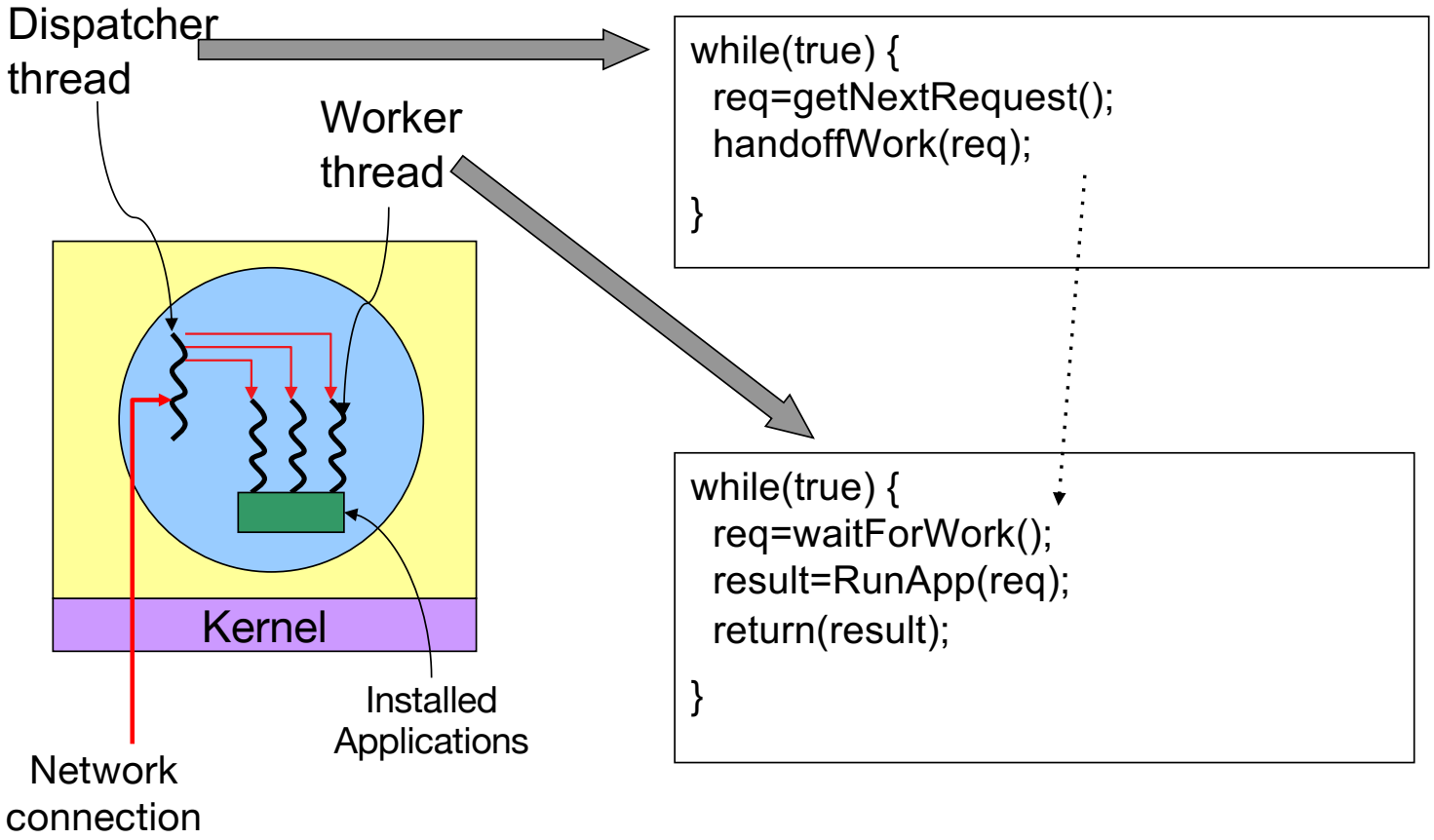
=> Each thread has its own stack!

Why using threads?

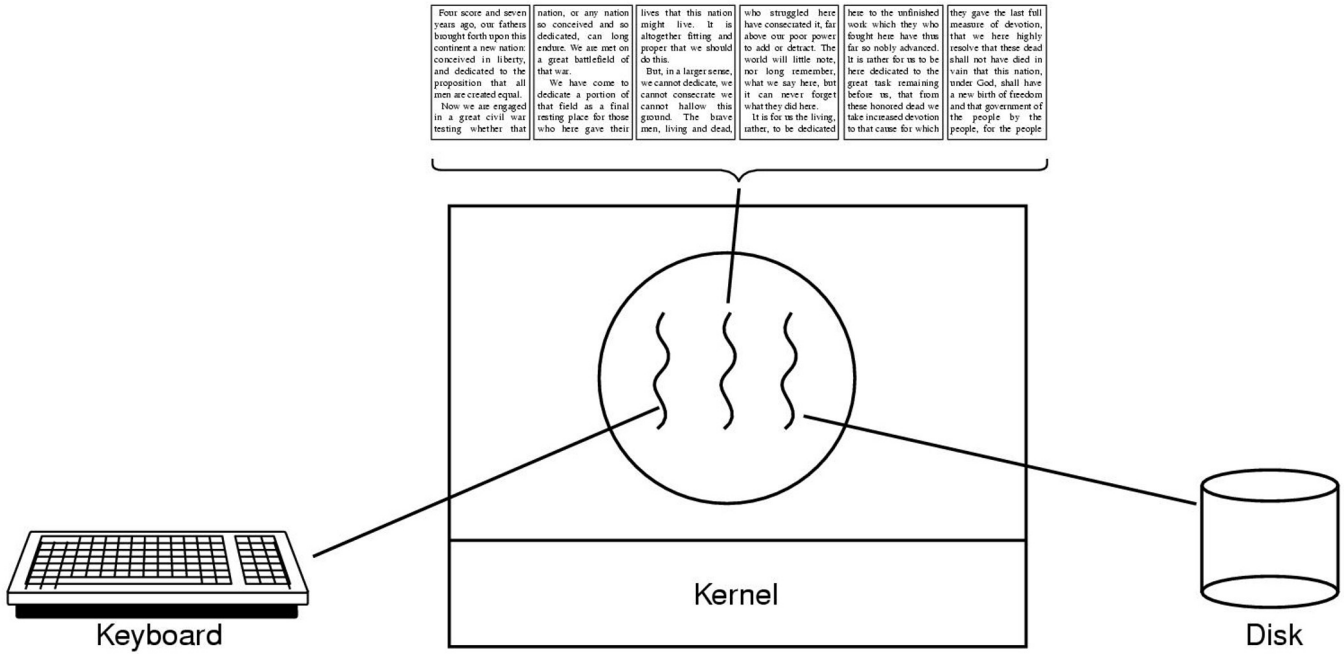
- Allow a single application to do many things at once
  - Simpler programming model
  - Less waiting
- Threads are faster to create or destroy
  - No separate address space
- Overlap computation and I/O
  - Could be done without threads, but it's harder
- Example: word processor
  - A thread to handle input
  - A thread to format document
  - A thread to write to disk



Multithreaded Application Server

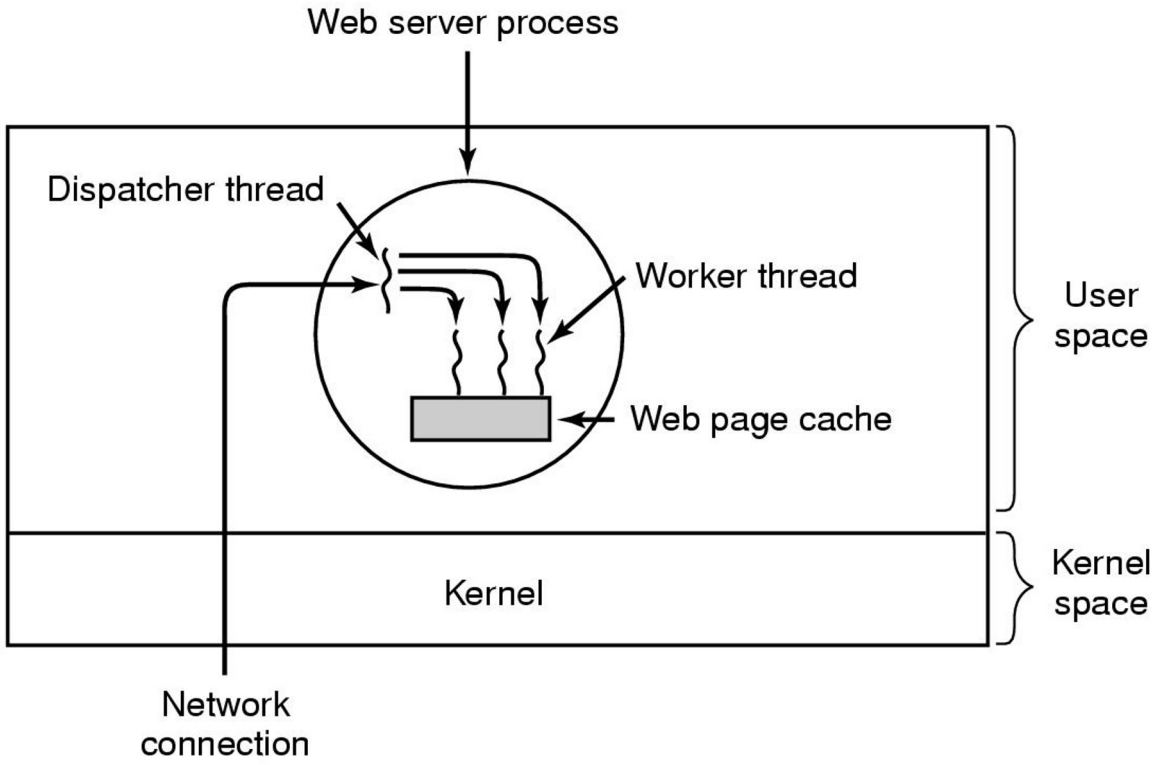


Thread Usage Example (1)



A word processor with three threads

Thread Usage Example (2)



A multithreaded Web server

Thread Usage Example (3)

```
while( true ){
      req = get_next_request( );
       handoff_work(req);
      }
```

Dispatcher thread

```
while( true ){
      req = wait_for_work( );
       page =look_for_page_in_cache(req);
       if( page == null )
             page=read_page_from_disk(req);
      Return( page);
      }
```

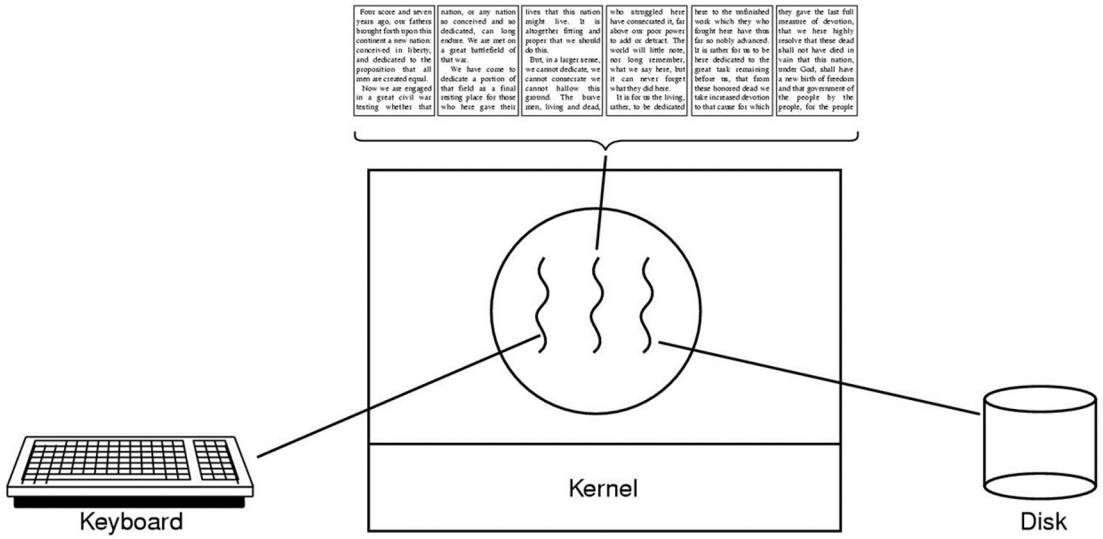
Worker Thread

Three Ways to Build a Server

- Single-threaded blocking process:
  - Slow, but easier to implement
  - No parallelism
  - Blocking system calls
- Single-threaded non-blocking process
  - Parallelism
  - Non-blocking system calls
  - The result of the system call is either sent to the caller later or polled by the caller later.
- Multi-threaded process
  - Parallelism
  - Non-blocking system calls

Concurrency Example: Multi-threaded Word Processor

- In a multi-threaded word processor:
  - When the first thread is reading the next segment from the file, the second thread formats the text, and the third thread is waiting for the user input.
  - The shared resource is CPU time



Part Two!

- In this part, we will discuss what a thread means and how it is created. Try to find answers to the following questions:
  1. Why do we store less info per thread than per process?
  2. How can a process overlap I/O with computation?
  3. How are non-blocking system calls implemented in a multi-threaded model?
  4. How are non-blocking system calls implemented in a single-threaded model?
  5. Which multi-threading model is more suitable for web servers with a dispatcher-worker structure?

Part Three!

Part Three!

- In this part, we will discuss what a thread means and how it is created. Try to find answers to the following questions:

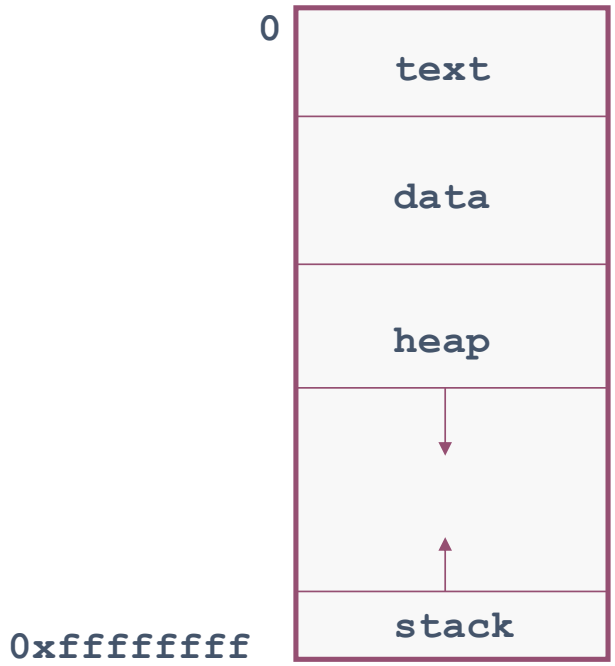
1. How is the memory space of a process organized? What segments are there?
2. Can we share all memory segments between threads?
3. What are the most common methods of inter-process communications?
4. What are the two primary mechanisms in a message-passing system?

Process / Thread: Memory

Process Memory Organization

- A process needs memory space to store
  - Its code
  - The global constant values
  - Temporary values
  - Dynamically allocated variables
  - Etc.
- The memory segments used by a process are organized based on their usage.
- [These segments might not be contiguous in memory but for *simplicity* we will assume so here]

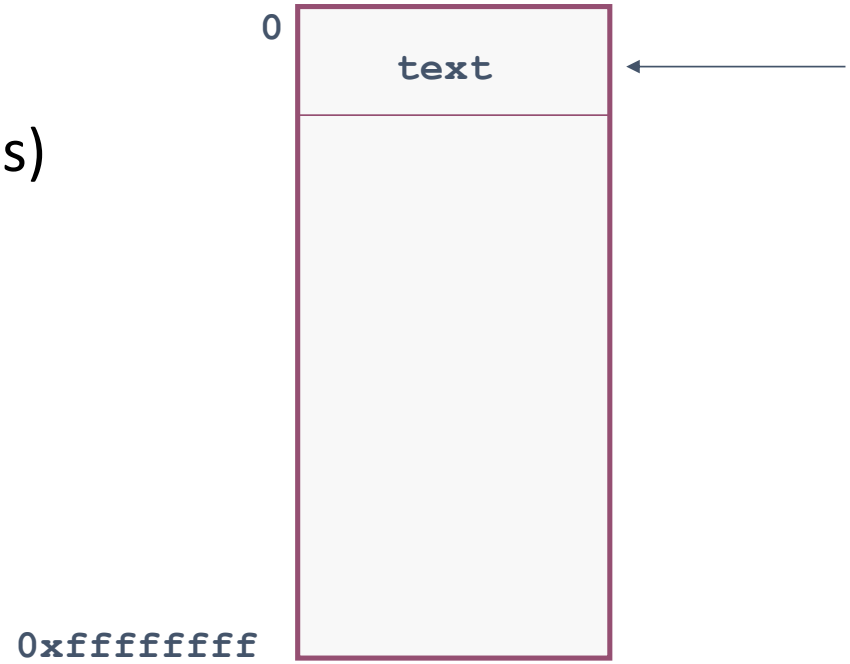
Process Memory Organization



N.B.: this is a model, not all the implementations correspond literally on the specific fields or name, but represent very well the organization concept (example x86 stack is normally getting toward zero in others the opposite).

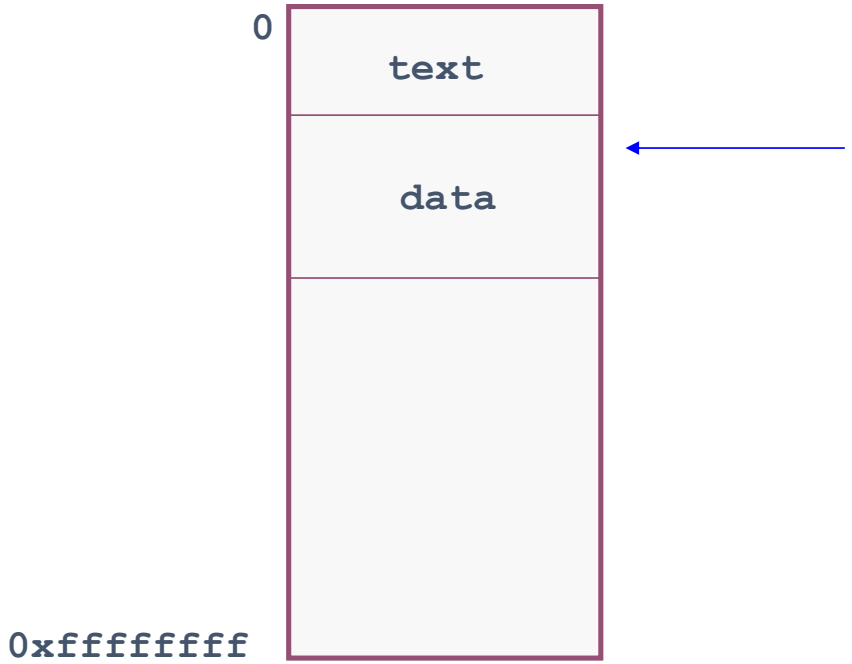
Process Memory Organization: text

- Includes program code and constants
  - binary form (executable commands)
  - loaded libraries
  - known as the “text” segment
  - space calculated at compile-time



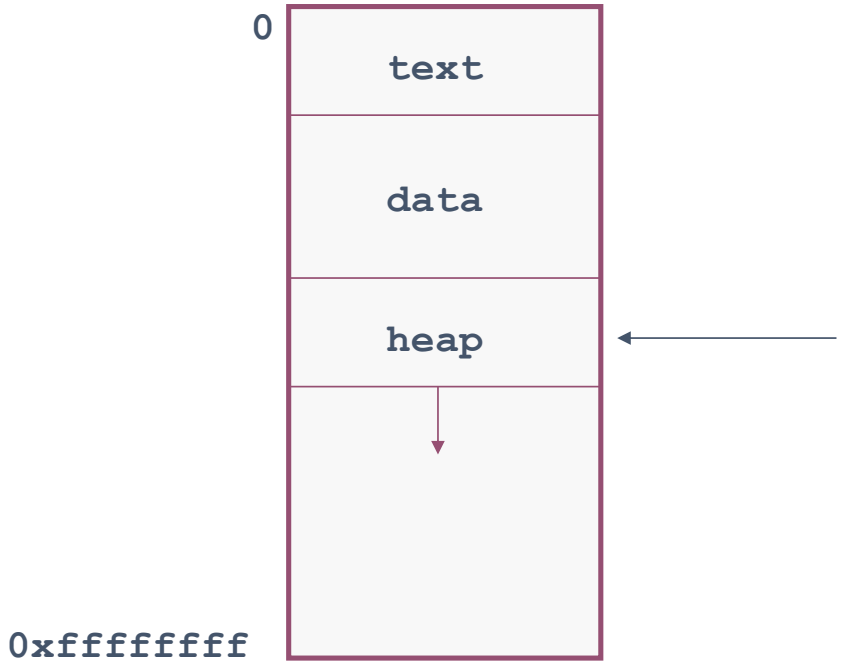
Process Memory Organization: data

- Data: initialized global data in the program
  - Ex: int size = 100;



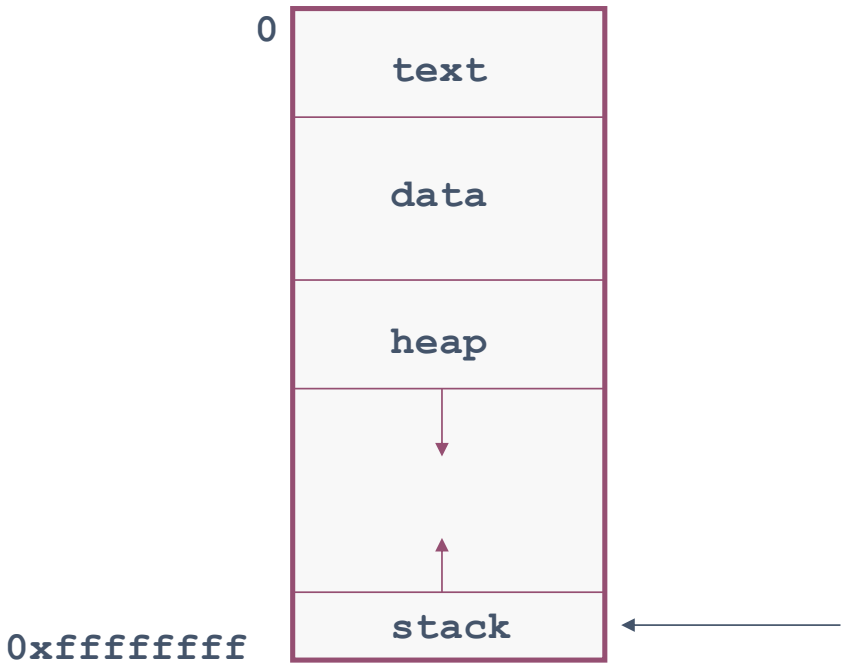
Process Memory Organization: heap

- Heap: dynamically-allocated spaces
  - Ex: MyArray = new int[100]
  - OS knows nothing about its
    - space
    - content
  - dynamically grows as the program runs



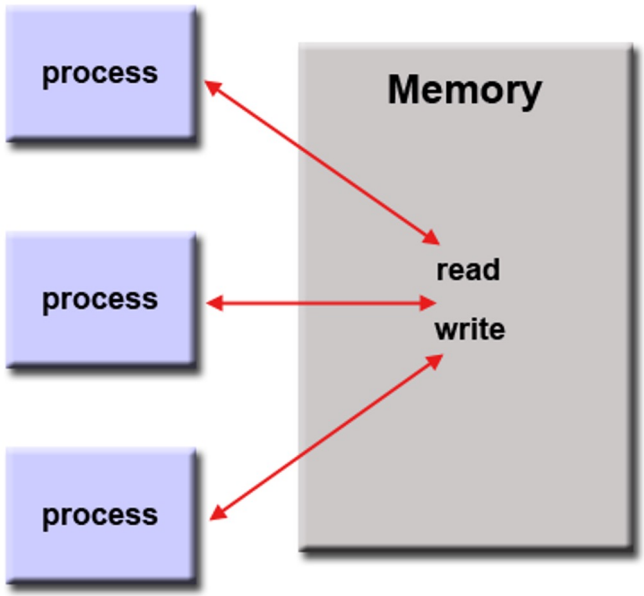
Process Memory Organization: stack

- Stack: local variables and functions
  - support function call/return and recursive functions
  - grows to a low address (upward in the figure)



Communications between threads

- Concurrent threads can communicate through a shared resource:
  - A piece of memory where they can write/read their messages.
- Sharing resources in concurrency can be done in different ways.



Communications between threads

- The most common ways of sharing resources are:
  - Shared memory: Parts of the main memory is shared between processes/threads
  - Message Passing: Processes/threads inform each other about the results of their tasks by communicating messages

Shared Memory with Threads

The concurrency model with shared memory is assumed to have multiple threads

Traditional View: A running program (process) has:

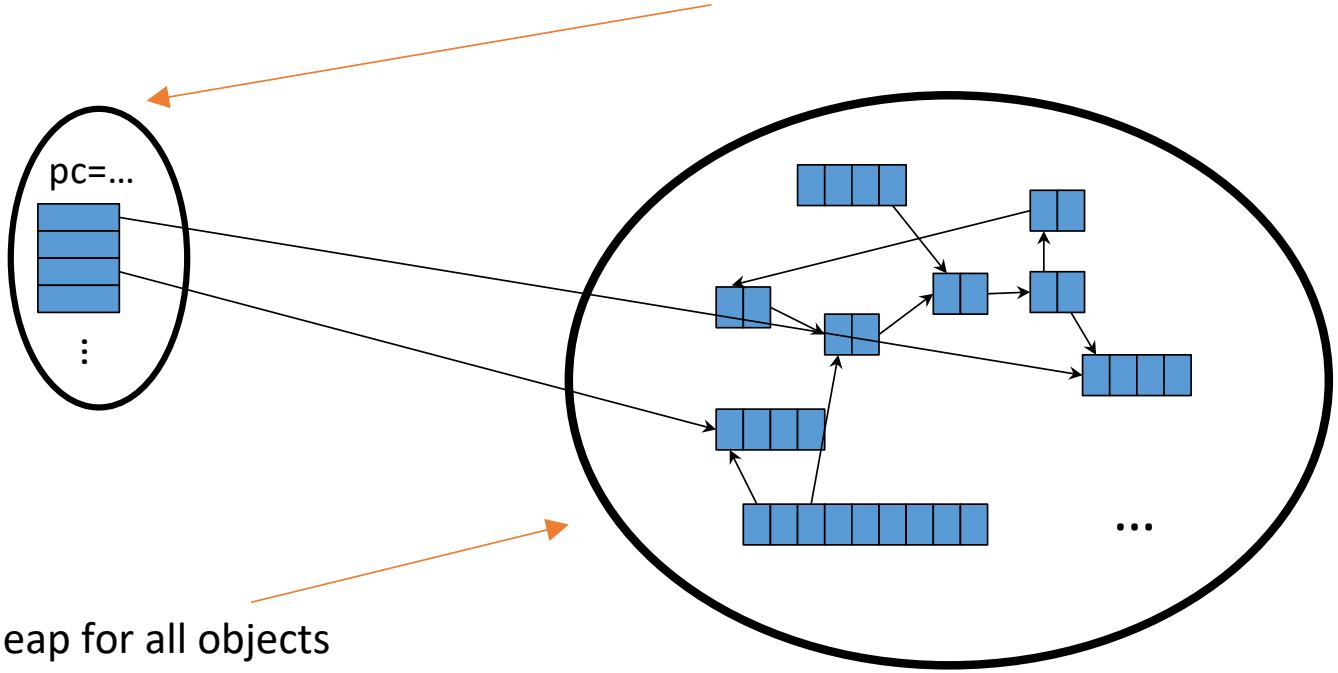
- One program counter (the next statement that is going to be executed)
- One **stack** (each stack frame holding local variables)
- Objects in the **heap** created by memory allocation (i.e., new) (same name, but no relation to the heap data structure)
- Static fields in the class shared among objects

Single-Threaded Process View

Stack with local variables

Program counter for the next statement

Local variables are primitives or heap references

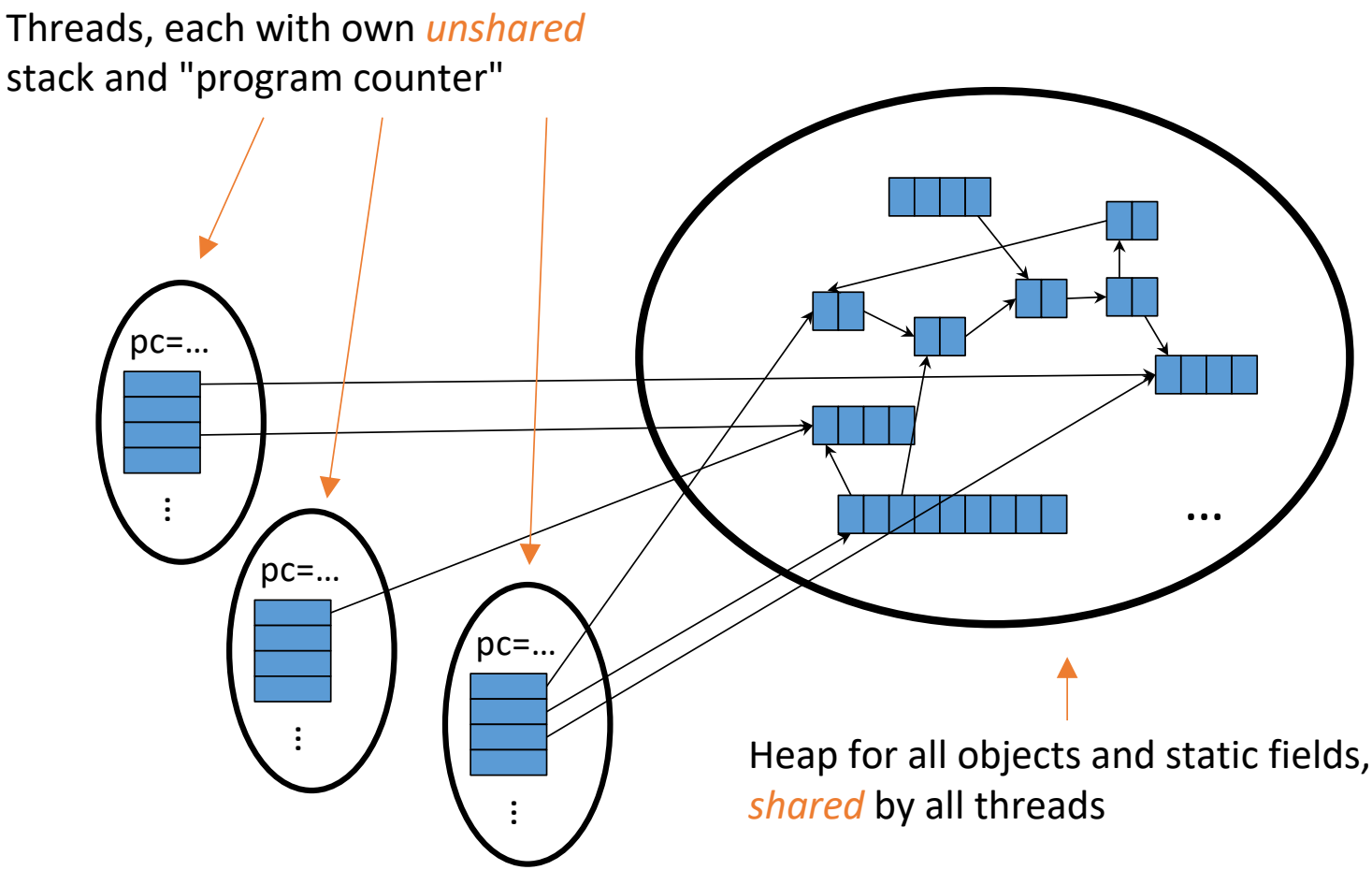


Heap for all objects
and static fields

Multi-Threaded Process View

- A set of threads, each with a program counter and a stack but no access to another thread's local variables
- Threads can share objects and static fields
- Communication among threads occurs through:
  - writing values to a shared location to be read by another thread

Multi-Threads & Shared Memory View



Message Passing

- Shared-memory communication is not very effective in some concurrent programming models.
- For example, the standard programming model of Unix (Linux, etc.) was based on **processes** rather than threads.
  - We can take the other major approach to manage concurrency, called **message-passing**.

Message Passing Mechanisms

- Two primary mechanisms are needed in a message-passing system:
  - A method of creating and specifying communication channels
  - A method of sending and receiving messages (no need for network).

Part Three!

- In this part, we will discuss what a thread means and how it is created. Try to find answers to the following questions:

1. How is the memory space of a process organized? What segments are there?
2. Can we share all memory segments between threads?
3. What are the most common methods of inter-process communications?
4. What are the two primary mechanisms in a message-passing system?

Summary

- Threads are the smallest set of instructions that can be executed independently
- Multi-threaded programs can overlap blocking I/O or system calls with processing to efficiency
- Threads can help share resources easily, improve responsiveness, and speed up processing.
- The memory space of a process is organized as code (text), data, heap, stack, etc. Not all segments are shared between threads.
- Communication models (we will continue this next week)
  - Message Passing
  - Shared memory

Quiz time!

Head to teams… You know the drill.

Time to apply

Follow the instructions for exercise Week 3