CS310 Operating Systems

Lecture 11: Thread Abstraction and its implementation

Ravi Mittal

IIT Goa

References

- Book: Operating Systems: Principles and Practice: Thomas Anderson and Michael Dahlin, Part 1 and Part 2
- CS162, Operating Systems and Systems Programming, University of California, Berkeley
- CS4410, Operating Systems, Course, Cornell University, Spring 2019, Lecture on Threads
- Operating Systems: Three Easy Pieces, by Remzi and Andrea Arpaci-Dusseau,
available for free online
- Book: Modern Operating Systems, Andrew Tenenbaum, and Herbert Bos, 4^{th} Edition, Pearson

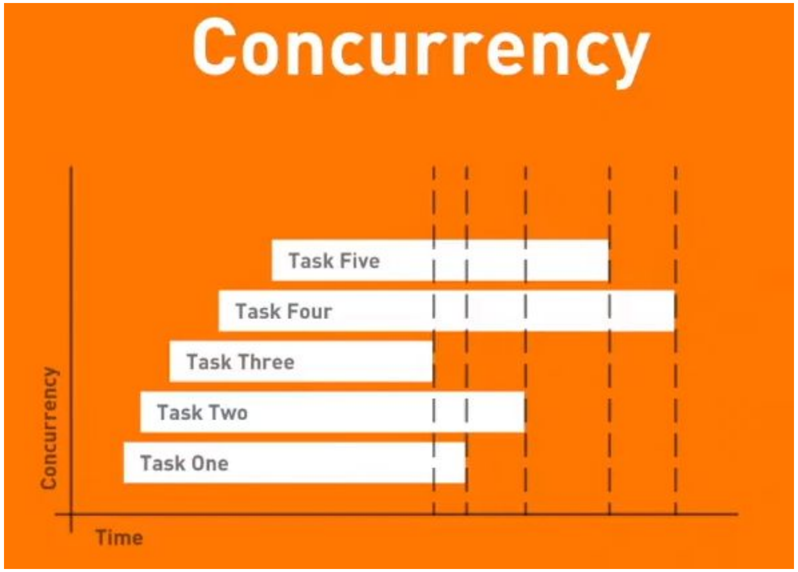
Read the following:

- Book: Operating Systems: Principles and Practice (2nd Edition) Anderson and Dahlin
  - Volume 2, Concurrency
    - Chapter 4: Concurrency and Threads
- Book: Modern Operating Systems: Tenenbaum and Bos
  - Chapter 2: Processes and Threads

Concurrency vs Parallelism

Concurrency - concept

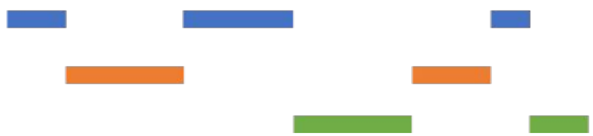
- Word concurrency refers to multiple activities that can happen at the same time
  - Analogy: juggling
- Real World is concurrent
  - Your Mother's activities
  - Role of secretary, etc



Concurrency Vs Parallelism

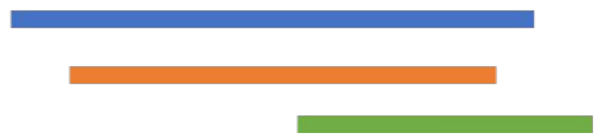
Concurrency

Tasks start, run and complete in an interleaved fashion



Parallelism

Tasks run simultaneously



We can combine both concurrency and parallelism

Smart phones are concurrent

- Mobile Phones
  - Do many tasks simultaneously on multiple processors
  - Hundreds of tasks go on in a mobile phone at any time
    - Receiving signals
    - Sending Signals
    - Running battery checking routine
    - Running music application
    - Running Browsers
    - Mobile Management task
    - Security related tasks
    - Running Internet Stack
    - Running Interface applications

Today's applications – inherently concurrent

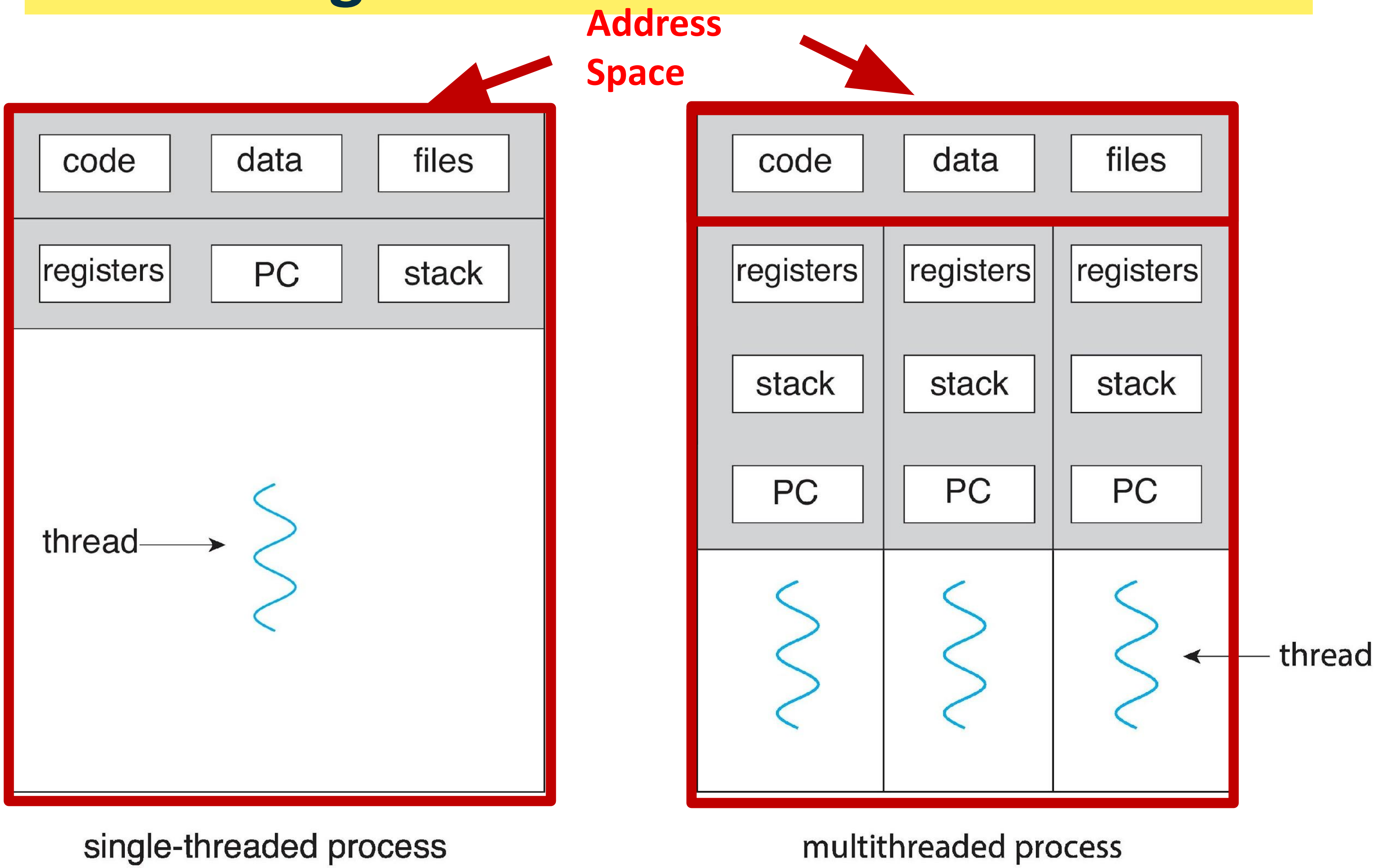
- Most applications today have user interfaces - that demand good responsiveness to users, while
  - Simultaneous executing application logic
- Simultaneous support for a large number of users
- Example:
  - Amazon.com
  - Online Banking applications
  - Self Driving cars'
- For a programmer
  - It is easy to think sequentially
  - Complex to plan, design, develop system that have thousands of tasks running concurrently and parallelly

Thread Abstraction

Recall: Thread Concept

- A single-execution stream of instructions that represents a separately schedulable task
  - Single execution Sequence: Each thread executes a sequence of instructions – just as in the sequential programming model
  - Separately schedulable task: OS can run, suspend, resume a thread at any time
  - Bound to a process: lives in an address space
- Each thread is very much like a separate process, except for one difference: they *share* the same address space and thus can access the same data
- Finite Progress Axiom: execution proceeds at some unspecified, non-zero speed

Recall: Single and Multithreaded Processes



- Address spaces encapsulate protection: Passive Part
- Threads share code, data, files, heap

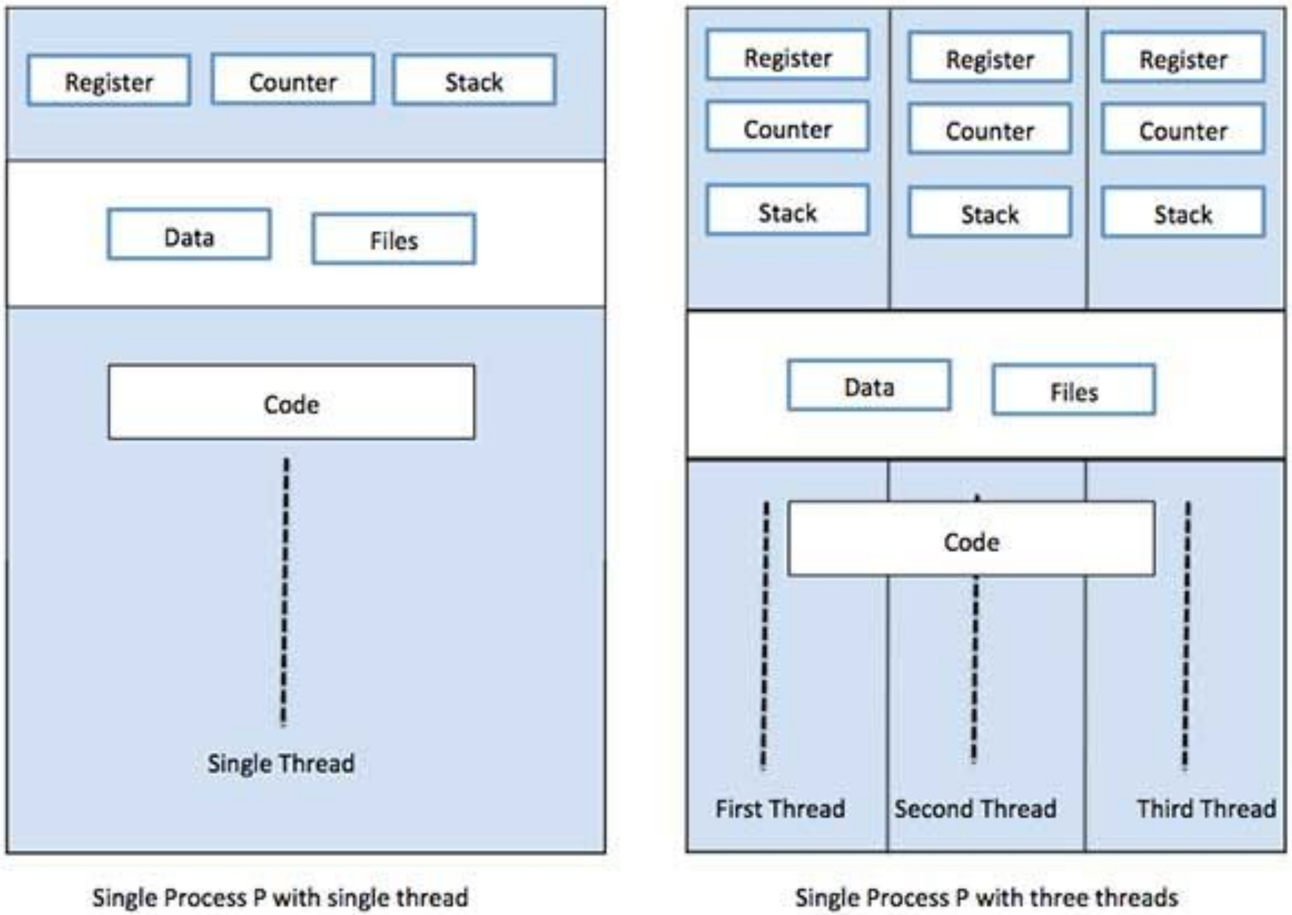
Recall: Thread - basics

- A scheduler may context switch a thread and bring in another thread for execution
- Each thread appears to be a single stream of execution
  - Programmer needs to pay attention only to the sequence of instructions within a thread and not whether or when that sequence may be suspended to let another thread run
- Main thread is the initial thread of every process
  - This is basically Main()
- Every additional thread for some function
  - Example: for I/O operation
  - For network data receiving

Thread Abstraction

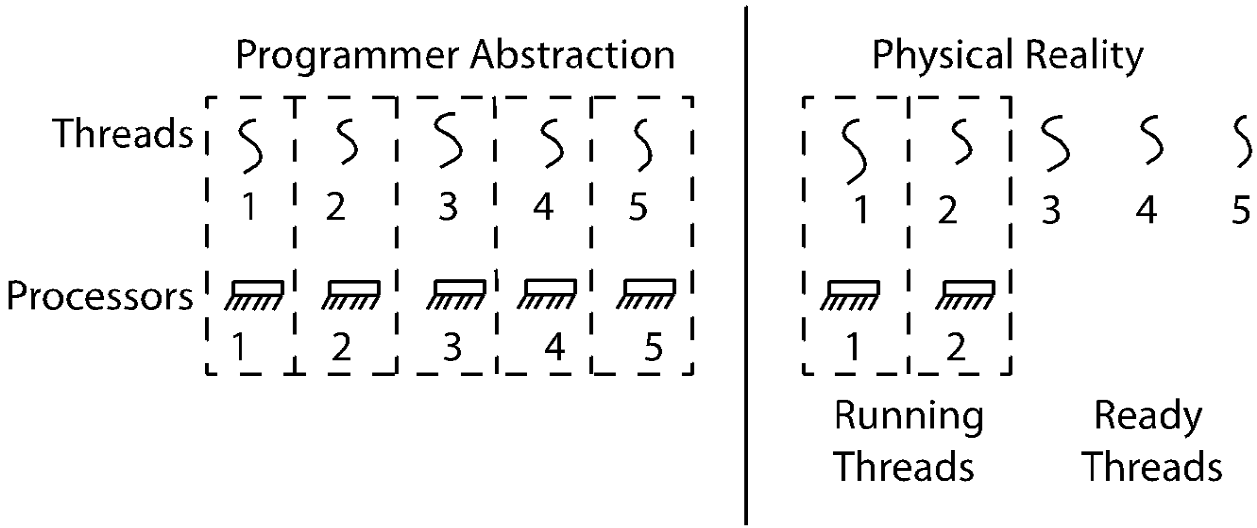
- It helps programmers to create as many threads as needed
  - Without worrying about number of processors in a system
- OS provides illusion of a very large (nearly infinite) number of virtual processors
- OS manages the illusion by transparently
  - Suspending and resuming threads so that at a given time a subset of threads are actively running

Thread Abstraction

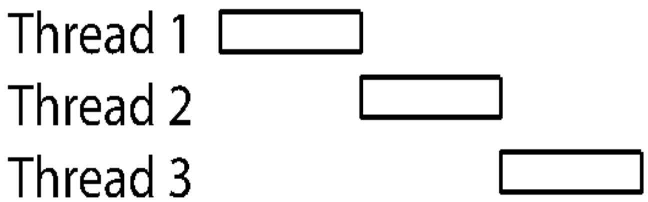


Thread – a basic element to support concurrency

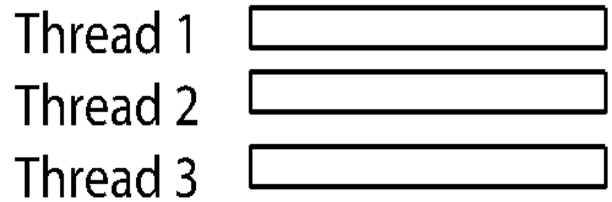
- How to write a concurrent program – with so many activities?
- Think it as a set of sequential stream of execution – or threads
  - That interact and share results in very precise ways
- With threads we can define a set of tasks that run concurrently while code for each task is sequential



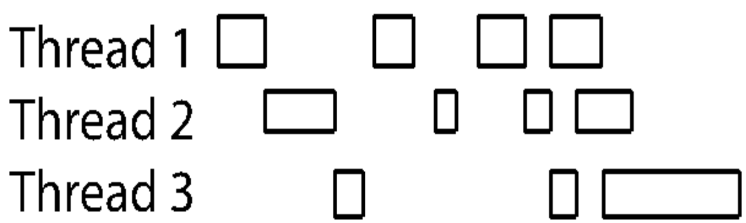
Possible Executions



a) One execution



b) Another execution



c) Another execution

Programmer vs. Processor View

Single Thread's Execution

Programmer's View	Possible Execution #1
.	.
.	.
.	.
$x = x + 1;$	$x = x + 1;$
$y = y + x;$	$y = y + x;$
$z = x + 5y;$	$z = x + 5y;$
.	.
.	.
.	.

Programmer vs. Processor View

Single Thread's Execution

Programmer's View	Possible Execution #1	Possible Execution #2
.	.	.
.	.	.
.	.	.
$x = x + 1;$	$x = x + 1;$	$x = x + 1$
$y = y + x;$	$y = y + x;$	....
$z = x + 5y;$	$z = x + 5y;$	thread is suspended
.	.	other thread(s) run
.	.	thread is resumed
.	.	....
		$y = y + x$
		$z = x + 5y$

Programmer vs. Processor View

Single Thread's Execution

Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
$x = x + 1;$	$x = x + 1;$	$x = x + 1$	$x = x + 1$
$y = y + x;$	$y = y + x;$	....	$y = y + x$
$z = x + 5y;$	$z = x + 5y;$	thread is suspended	....
.	.	other thread(s) run	thread is suspendec
.	.	thread is resumed	other thread(s) run
.	.	....	thread is resumed
		$y = y + x$	....
		$z = x + 5y$	$z = x + 5y$

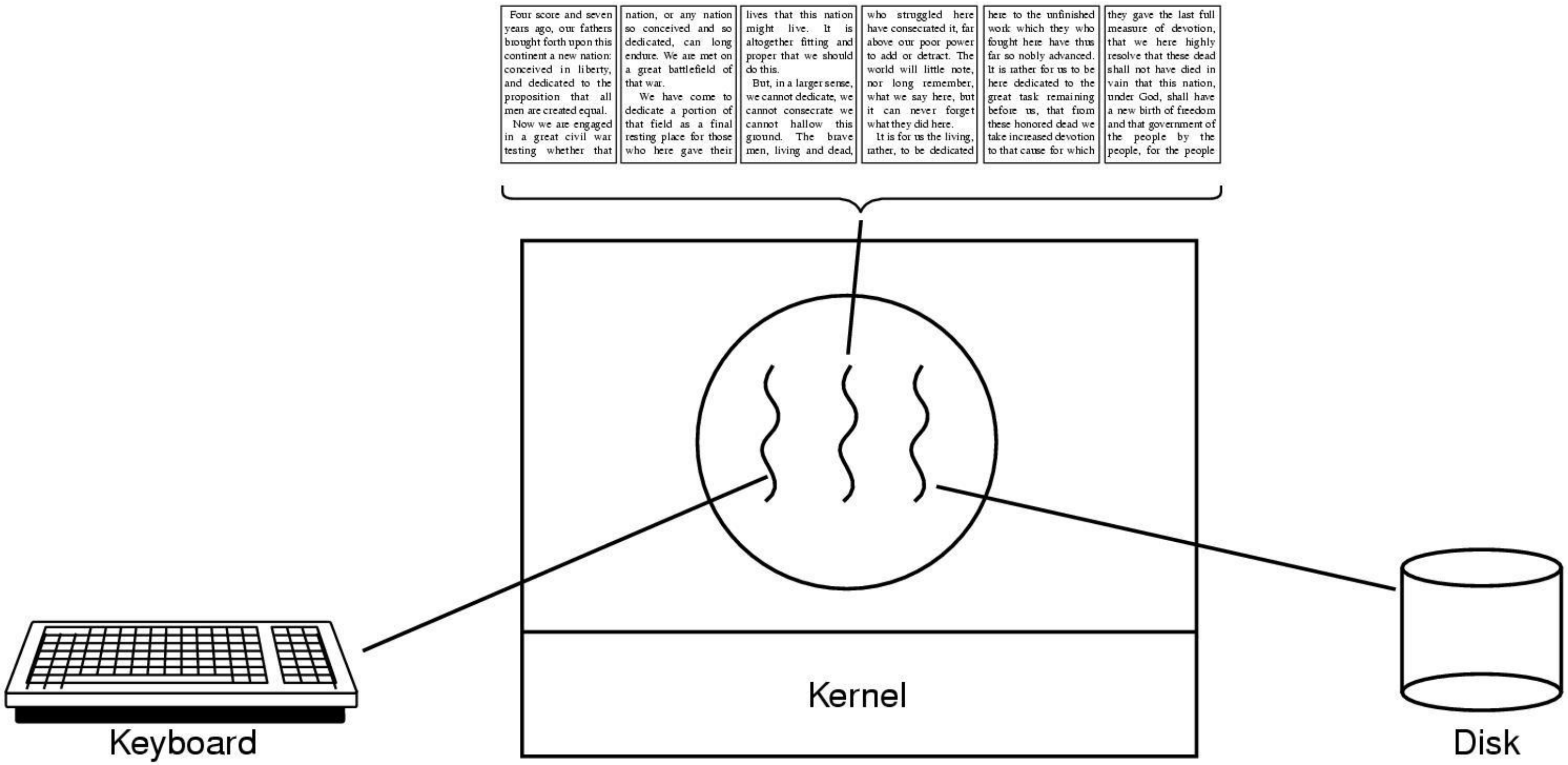
Example 1: Word Processor

- Example: A writing book consisting of a number of chapters
- Good to keep all chapters in a single file for ease of operations ; say total 800 pages
- Consider
  - User deletes a sentence in Page # 1. Now, user wants to make change in page no 600 and gives command to go there
  - Word processor is now forced to reformat all pages (at least up to page no 600)
  - If there is a single process to do all tasks – User may face delays
  - Solution: two threaded program – one interacts with the user another handles reformatting; The second thread starts work as soon as the user deletes the line on page 1
  - How about addition of the third thread for auto saving?

Adapted from Tenenbaum's book: Modern Operating Systems

Example 1: Word Processor

A word processor with three threads



- If the program were single threaded
  - Whenever a disk backup started, command from keyboard and mouse would be ignored unless backup completed

Adapted from Tenenbaum's book: Modern Operating Systems

Example 1 Word Processor

- The 1st thread works on user interaction
- The 2nd thread works on formatting document when it is told to do so
- The 3rd thread writes the contents of RAM to disk periodically (auto save feature)

- Note that all 3 thread work on common document – they share a common memory – all have access to the document being edited
  - Three processes instead of thread will complicate the task

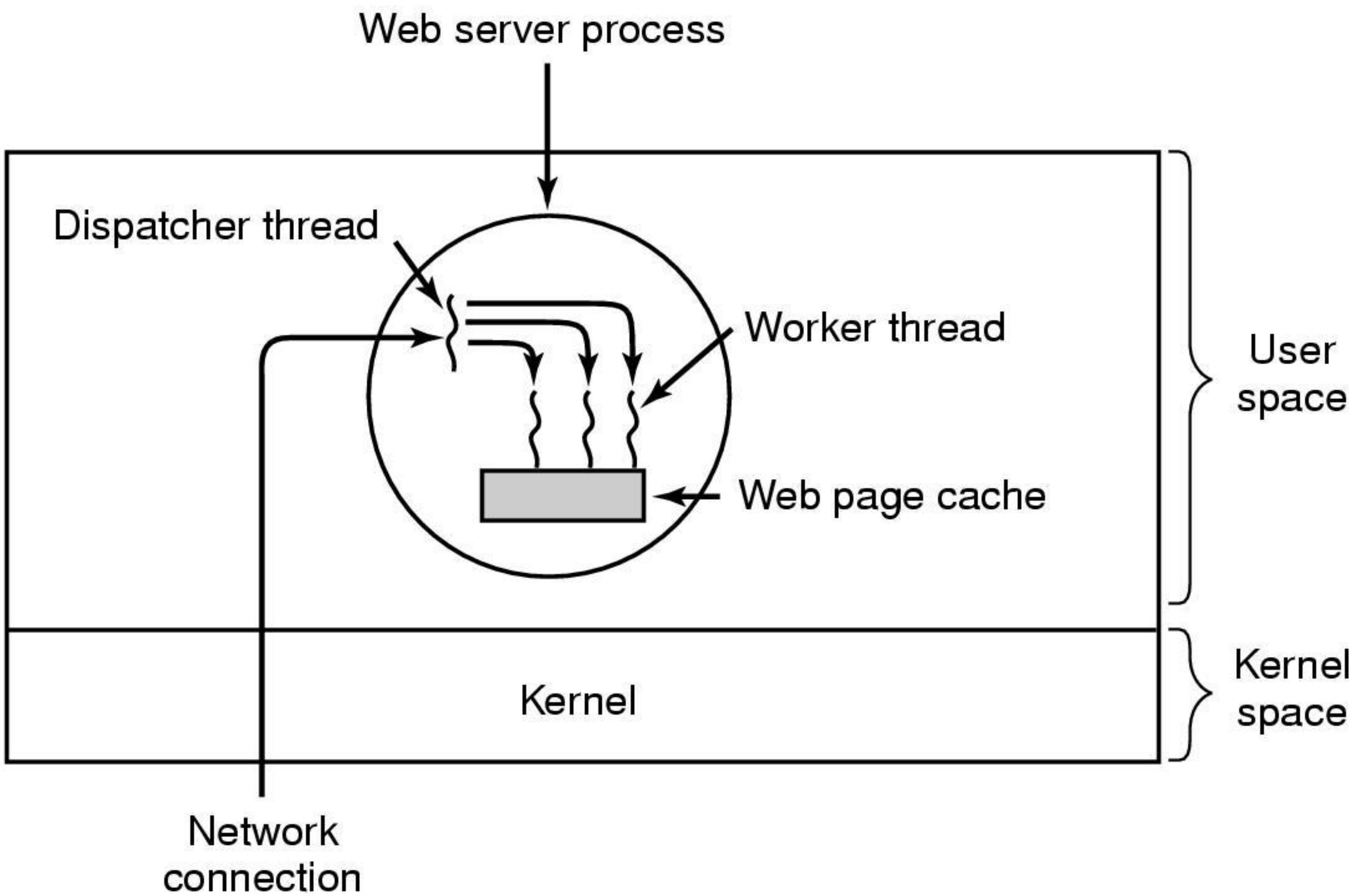
Example 2: Processing of a large amount of data

- Normal approach:
  - Read in a block of data
  - Modify it
  - Write it back
  - The process will use block system call at read and write operation
  - CPU will go idle
- Better Approach:
  - Processes can be structured as input thread, processing thread and an output thread
  - Input thread: reads data into input buffer
  - Processing thread: takes data from input buffer, processes data and puts the results in an output buffer
  - Output thread: Writes buffer data to the disk
    - The soln works if syscall blocks only calling thread not the entire process

Example 3: A multithreaded web server (1)

- Page request from clients – page is sent back
- Web page cache – collection of heavily used pages in MM
- Dispatcher Thread
  - reads incoming request for work from network
  - Chooses an idle (blocked) worker thread and handover the request and make it into ready state
- Worker thread: When scheduled, worker thread checks if the page is available in webpage cache
  - If yes, deliver it to another process for sending it over the network
  - If not, get the page from disk ; block till it gets the page from disk
- Both dispatcher and worker threads operate in infinite loop

Example 3: A multithreaded web server (2)



Adapted from Tenenbaum's book: Modern Operating Systems

Example 3: A multithreaded web server (3)

Dispatcher thread

```
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
```

(a)

Worker thread

```
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)

Adapted from Tenenbaum's book: Modern Operating Systems

Recall: Why Threads?

- To express a natural program structure
  - Updating the screen, fetching new data, receiving user input — different tasks within the same address space
- To exploit multiple processors
  - Different threads may be mapped to distinct processors
- To maintain responsiveness
  - Splitting commands, spawn threads to do work in the background
- To mask long latency of I/O devices
  - Do useful work while waiting
  - Instead of waiting, the program may do something else
    - CPU can perform other computation, or
- Threads are the natural way to avoid getting stuck
  - Why not processes instead of threads?
    - Threads are light weight: Share data and address space
    - Processes are sound choice when using logically separate tasks

Why Threads?

- Threads are lighter weight than processes
  - Easier to create and destroy than processes
    - Creating thread is 10-100 times faster than creating a process
    - Specially when the number of threads needed changes rapidly and dynamically,
- Parallel programs must parallelize for performance
- Programs with user interface need threading to ensure responsiveness
- Network and disk bound programs use threading to hide network/disk latency

- A multithreaded program is a generalization of the same basic programming model
  - Each individual thread follows a single sequence of steps (eg loops, call/return from functions, conditions etc)
  - A program can have several such threads in execution at the same time

Recall: Thread

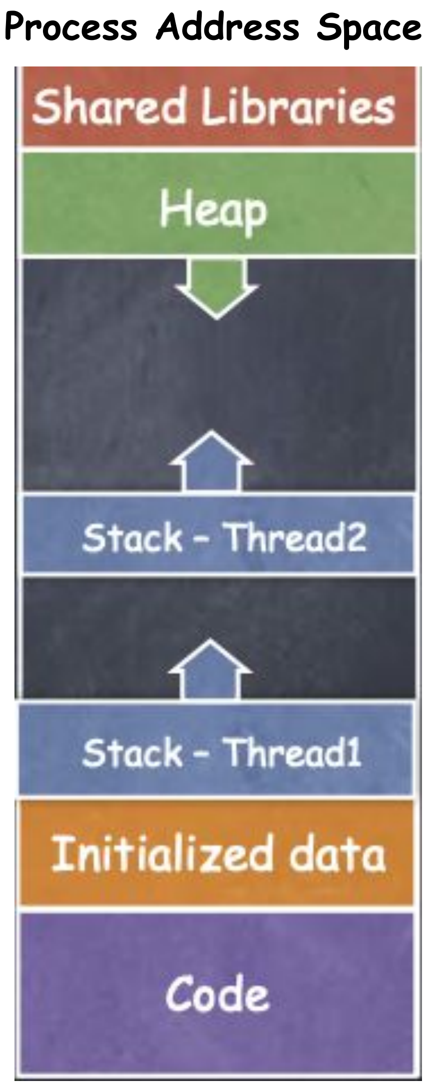
- All threads within a process share
  - Heap
  - Global/static data
  - Libraries
- Each thread has a separate
  - Program Counter
  - Stack
  - registers

Any stack-allocated variables, parameters, return values, and other things that we put on the stack

Process Address Space

Recall: Threads

- All threads within a process share
  - Heap
  - Global/static data
  - Libraries
- Each thread has a separate
  - Program Counter
  - Stack
  - registers

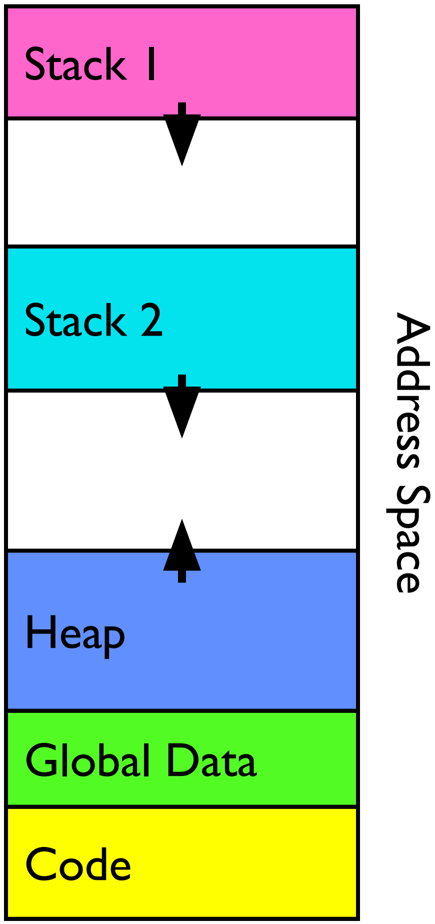


Suspension and termination

- Suspending a process involves suspending all threads of the process since all threads share the same address space
- Termination of a process, terminates all threads within the process

Memory Footprint: Two-Threads

- If we stopped this program and examined it with a debugger, we would see
  - Two sets of CPU registers
  - Two sets of Stacks



- Questions:
  - How do we position stacks relative to each other?
  - What maximum size should we choose for the stacks?
  - What happens if threads violate this?
  - How might you catch violations?

Lecture Summary

- Thread is single unique execution context
    - Program counter, registers, stack
        - Registers hold the root state of the thread: PC, Gen Regs
        - Rest of the thread’s state is in memory
- Threads naturally implement concurrency
- A Process consists of more or more threads
- Threads are light weight
- Processes are more strongly isolated than threads