

# CSE 323: Operating System Design

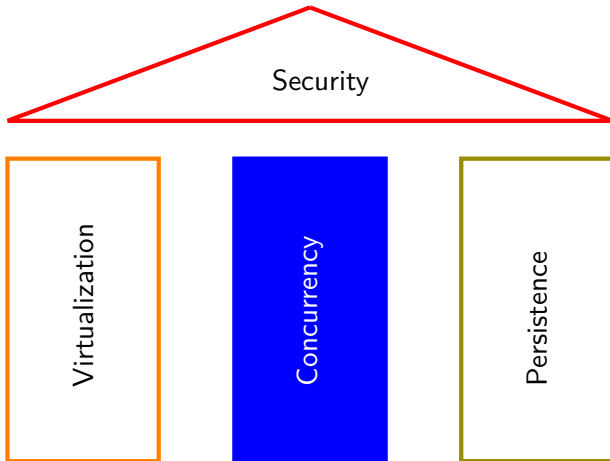
## Thread

Salman Shamil



North South University (NSU)  
Fall 2025

Original slides by Mathias Payer and Sanidhya Kashyap [EPFL]



- Thread abstraction
- Multi-threading challenges
- Key concurrency terms and definitions

This slide deck covers chapters 26 and 27 in OSTEP.

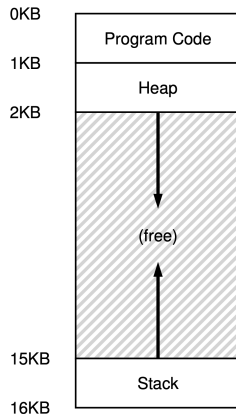
[**Credits:** Portions of the content are adapted from slides based on the OSTEP book by Prof. Youjip Won (Hanyang University) and Prof. Mythili Vutukuru (IIT Bombay), with thanks.]

# Threads: Execution Contexts

- Threads are independent execution contexts
  - similar to processes
  - EXCEPT they share the same address space

# Threads: Execution Contexts

- Threads are independent execution contexts
  - similar to processes
  - EXCEPT they share the same address space
- We only had one thread in a process so far
  - single-threaded program
  - one Program Counter (PC)
  - one Stack Pointer (SP)



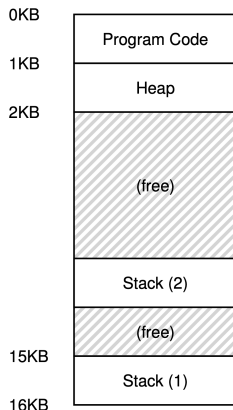
# Multi-threaded Process

- **What happens if we want multiple threads in parallel?**
  - shared address space but separate execution stream
  - is that possible with a shared stack or PC?

- **What happens if we want multiple threads in parallel?**
  - shared address space but separate execution stream
  - is that possible with a shared stack or PC?
  - each thread has separate stack and PC
    - leading to independent function calls
    - able to execute different parts
  - code and heap segments are still shared

# Multi-threaded Process

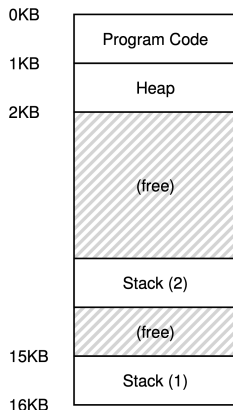
- **What happens if we want multiple threads in parallel?**
  - shared address space but separate execution stream
  - is that possible with a shared stack or PC?
  - each thread has separate stack and PC
    - leading to independent function calls
    - able to execute different parts
  - code and heap segments are still shared





# Multi-threaded Process

- **What happens if we want multiple threads in parallel?**
  - shared address space but separate execution stream
  - is that possible with a shared stack or PC?
  - each thread has separate stack and PC
    - leading to independent function calls
    - able to execute different parts
  - code and heap segments are still shared



- ***user-level threads***: scheduled by thread library in user space
- ***kernel-level threads***: scheduled directly by the OS

## Concurrency vs Parallelism

- **Concurrency:** multiple processes/threads making progress during the same time period
  - Possibly on a single core by interleaving executions
  - Better CPU utilization (e.g., when one thread is blocked on I/O, another runs)
- **Parallelism:** running multiple processes in parallel over multiple CPU cores
  - A single process can achieve parallelism with multiple threads

## Concurrency vs Parallelism

- **Concurrency:** multiple processes/threads making progress during the same time period
  - Possibly on a single core by interleaving executions
  - Better CPU utilization (e.g., when one thread is blocked on I/O, another runs)
- **Parallelism:** running multiple processes in parallel over multiple CPU cores
  - A single process can achieve parallelism with multiple threads

## How do they communicate?

- Processes need complicated Inter-Process Communication
- Extra memory footprint for IPC
- Threads can do it by simply using global variables (shared)
- **Question:** When to use threads vs processes?

# Creating Threads

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#include "common.h"
#include "common_threads.h"

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int main(int argc, char *argv[]) {
    if (argc != 1) {
        fprintf(stderr, "usage: main\n");
        exit(1);
    }

    pthread_t p1, p2;
    printf("main: begin\n");
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: end\n");
    return 0;
}
```

# Shared data is useful but not so simple!

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

// shared global variables
int max;
volatile int counter = 0;
// ^ no caching on register

void *mythread(void *arg) {
    char *letter = arg;
    int i; // on stack
           // (private per thread)
    printf("%s: begin \
           [addr of i: %p]\n",
           letter, &i);
    for (i = 0; i < max; i++) {
        counter = counter + 1;
        // shared: only one
    }
    printf("%s: done\n", letter);
    return NULL;
}
```

```
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, \
                "usage: main-first <loopcount>\n");
        exit(1);
    }
    max = atoi(argv[1]);

    pthread_t p1, p2;
    printf("main: begin \
           [counter = %d]\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done \
           [counter: %d] \
           [should: %d]\n",
           counter, max*2);

    return 0;
}
```

**Will the final count always be  $2 \times \text{max}$ ?**

# Uncontrolled Scheduling

- assembly instructions for `counter = counter + 1` (in x86)

```
100    mov 0x8049a1c, %eax
105    add $0x1, %eax
108    mov %eax, 0x8049a1c
```

[Critical Section] consider a context switch after 'add'.

# Uncontrolled Scheduling

- assembly instructions for `counter = counter + 1` (in x86)

```
100    mov 0x8049a1c, %eax
105    add $0x1, %eax
108    mov %eax, 0x8049a1c
```

[Critical Section] consider a context switch after 'add'.

OS	Thread 1	Thread 2	(after instruction)		
			PC	eax	counter
	<i>before critical section</i>		100	0	50
	<i>mov 8049a1c,%eax</i>		105	50	50
	<i>add \$0x1,%eax</i>		108	51	50

# Uncontrolled Scheduling

- assembly instructions for `counter = counter + 1` (in x86)

```
100    mov 0x8049a1c, %eax
105    add $0x1, %eax
108    mov %eax, 0x8049a1c
```

[Critical Section] consider a context switch after 'add'.

OS	Thread 1	Thread 2	(after instruction)		
			PC	eax	counter
<i>interrupt save T1 restore T2</i>	<i>before critical section</i>		100	0	50
	<i>mov 8049a1c,%eax</i>		105	50	50
	<i>add \$0x1,%eax</i>		108	51	50
			100	0	50



# Uncontrolled Scheduling

- assembly instructions for counter = counter + 1 (in x86)

```
100    mov 0x8049a1c, %eax
105    add $0x1, %eax
108    mov %eax, 0x8049a1c
```

[Critical Section] consider a context switch after 'add'.

OS	Thread 1	Thread 2	(after instruction)		
			PC	eax	counter
interrupt save T1 restore T2	<i>before critical section</i> mov 8049a1c,%eax add \$0x1,%eax		100	0	50
			105	50	50
			108	51	50
		mov 8049a1c,%eax add \$0x1,%eax mov %eax,8049a1c	100	0	50
			105	50	50
			108	51	50
			113	51	51

# Uncontrolled Scheduling

- assembly instructions for `counter = counter + 1` (in x86)

```
100    mov 0x8049a1c, %eax
105    add $0x1, %eax
108    mov %eax, 0x8049a1c
```

[Critical Section] consider a context switch after 'add'.

OS	Thread 1	Thread 2	(after instruction)		
			PC	eax	counter
interrupt save T1 restore T2	<i>before critical section</i> mov 8049a1c,%eax add \$0x1,%eax		100	0	50
			105	50	50
			108	51	50
		mov 8049a1c,%eax add \$0x1,%eax mov %eax,8049a1c	100	0	50
			105	50	50
			108	51	50
			113	51	51
	<i>interrupt</i> save T2 restore T1		108	51	51

# Uncontrolled Scheduling

- assembly instructions for `counter = counter + 1` (in x86)

```
100    mov 0x8049a1c, %eax
105    add $0x1, %eax
108    mov %eax, 0x8049a1c
```

[Critical Section] consider a context switch after 'add'.

OS	Thread 1	Thread 2	(after instruction)		
			PC	eax	counter
	<i>before critical section</i> mov 8049a1c,%eax add \$0x1,%eax		100	0	50
			105	50	50
			108	51	50
	<i>interrupt</i> <i>save T1</i> <i>restore T2</i>		100	0	50
		mov 8049a1c,%eax	105	50	50
		add \$0x1,%eax	108	51	50
		mov %eax,8049a1c	113	51	51
	<i>interrupt</i> <i>save T2</i> <i>restore T1</i>		108	51	51
		mov %eax,8049a1c	113	51	51

## Race Condition

Concurrent execution of threads leading to different results depending on the order of execution. Such programs are *indeterminate*, producing different outputs across runs.

# Concurrency Terms

## Race Condition

Concurrent execution of threads leading to different results depending on the order of execution. Such programs are *indeterminate*, producing different outputs across runs.

## Critical Section

Portion of code resulting in a race condition, usually by accessing a *shared* resource (e.g., a variable or data structure).

# Concurrency Terms

## Race Condition

Concurrent execution of threads leading to different results depending on the order of execution. Such programs are *indeterminate*, producing different outputs across runs.

## Critical Section

Portion of code resulting in a race condition, usually by accessing a *shared* resource (e.g., a variable or data structure).

## Mutual Exclusion

Guarantees a single thread executes a critical section at a time, preventing race conditions. **[Atomicity]**

**Next:** We need to design synchronization primitives for **mutex**.