Operating Systems

Lecture 5.3

Concurrency Control Mechanisms - II

Instructor: Muhammad Arif Butt, PhD

Lecture Agenda

- Recap: Concurrency Control Mechanisms
- Introduction to Semaphores
- System-V vs POSIX API for Semaphores
- POSIX Un-named Semaphores
  - Achieving Mutual Exclusion
  - Achieving Serialization
  - Using Counting Semaphores
- POSIX Named Semaphores
  - Achieving Mutual Exclusion
  - Achieving Serialization
  - Using Counting Semaphores
- Challenges with Synchronization Primitives

Recap
Concurrency Control Mechanisms

Concurrency Control Mechanisms

Concurrency control mechanisms are synchronization primitives and techniques used to coordinate access to shared resources among multiple concurrent processes or threads, ensuring data consistency, preventing race conditions, and maintaining system correctness in multi-threaded/multi-process environments.

Overview of Concurrency Control Mechanisms

- **Software-level Synchronization Primitives:** Implemented through collaboration between C runtime libraries and the OS kernel. Examples include mutexes, spinlocks, condition variables, barriers, semaphores, and read-write locks.
- **Hardware-level Synchronization Primitives:** Hardware-based concurrency control mechanisms rely on special atomic CPU instructions (compare-and-swap and test-set-lock), that perform indivisible/atomic operations. These instructions enable synchronization directly at the hardware level without the need for kernel intervention or traditional locking mechanisms. Hardware atomics are commonly used in high-performance, low-latency applications where the overhead of locks is prohibitive.
- **Compiler-level Synchronization Primitives:** Higher-level concurrency control mechanisms are provided by programming languages, compilers, and frameworks to simplify thread synchronization. Instead of writing complex locking code manually, developers can use simple annotations or directives. These tools automatically generate the necessary synchronization logic behind the scenes. Examples include:
  - Java's `synchronized` keyword, which locks objects implicitly to prevent concurrent access.
  - GCC's `__transaction_atomic`, which allows blocks of code to run atomically, and rolling back changes if a conflict is detected.
  - C#'s `lock(object)` statement, which generates `Monitor.Enter()/Monitor.Exit()` calls wrapped in try-finally blocks to guarantee lock release even during exceptions.

S/W Level Primitives (Pthread Library)

- **Mutexes (Mutual Exclusion Locks):** Binary locks ensuring only one thread can access a critical section at a time. Threads block/sleep when lock is unavailable, yielding CPU to other threads. Use a mutex when you need exclusive ownership of a critical section.
- **Spinlocks:** Like mutexes, spinlocks are also used to achieve mutual exclusion, but do busy waiting instead of blocking, i.e., threads continuously poll for lock availability instead of blocking. Use spinlocks when the CS is very short or scenarios where lock contention is rare.
- **Barriers:** Synchronization points where threads must wait until all participating threads reach the barrier before any can proceed, ensuring coordinated execution phases. Use in parallel algorithms where threads must complete one stage before starting the next.
- **Condition Variables:** Synchronization primitives that allow threads to wait for specific conditions to become true, working in conjunction with mutexes to provide efficient blocking/waking mechanisms. Use condition variable when you need threads to wait efficiently for a state change. (e.g., producer-consumer wait for buffer not empty/full.

By default `libpthread's` mutexes, barriers and condition variables (which are POSIX standard) and spinlocks (which are GNU/Linux extensions) are used by threads within the same process. However, if you place them in shared memory and change their attribute to PTHREAD_PROCESS_SHARED, then they can also synchronize cooperating processes.

Introduction to Semaphores

Mutexes are optimized for locking, condition variables are optimized for waiting, and semaphores can do both

Introduction to Semaphores

- A semaphore is a synchronization primitive that can be initialized to any **non-negative integer**, and can only be modified using two atomic operations:
  - **`wait() or P()`**: Decrements the semaphore value by one:
    If semaphore ≥ 0 → thread/process continues.
    If semaphore < 0 → thread/process blocks (goes to waiting queue).
    `P()` stands for *proberen* (to test) in Dutch.
  - **`post() or V()`**: Increments the semaphore value by one:
    If there are waiting threads → one is unblocked.
    Otherwise, the semaphore value is just incremented
    `V()` stands for *verhogen* (to increment) in Dutch.
- **Types of Semaphores:** Depending on which thread/process is unblocked on a `post()`:
  - Strong Semaphore, which ensures FIFO (First-In-First-Out) order of unblocking
  - Weak Semaphores, does not guarantee order as any waiting thread may be chosen arbitrarily.
- **Uses of Semaphores:**
  - Binary semaphore: Initialized with 1, to achieve mutual exclusion.
  - Counting semaphore: Initialized with N, to allow N processes to access shared pool of resources.
  - Signalling semaphore: Initialized to 0, used to serialize or signal between threads/processes.

Achieving Mutual Exclusion (Binary Semaphores)

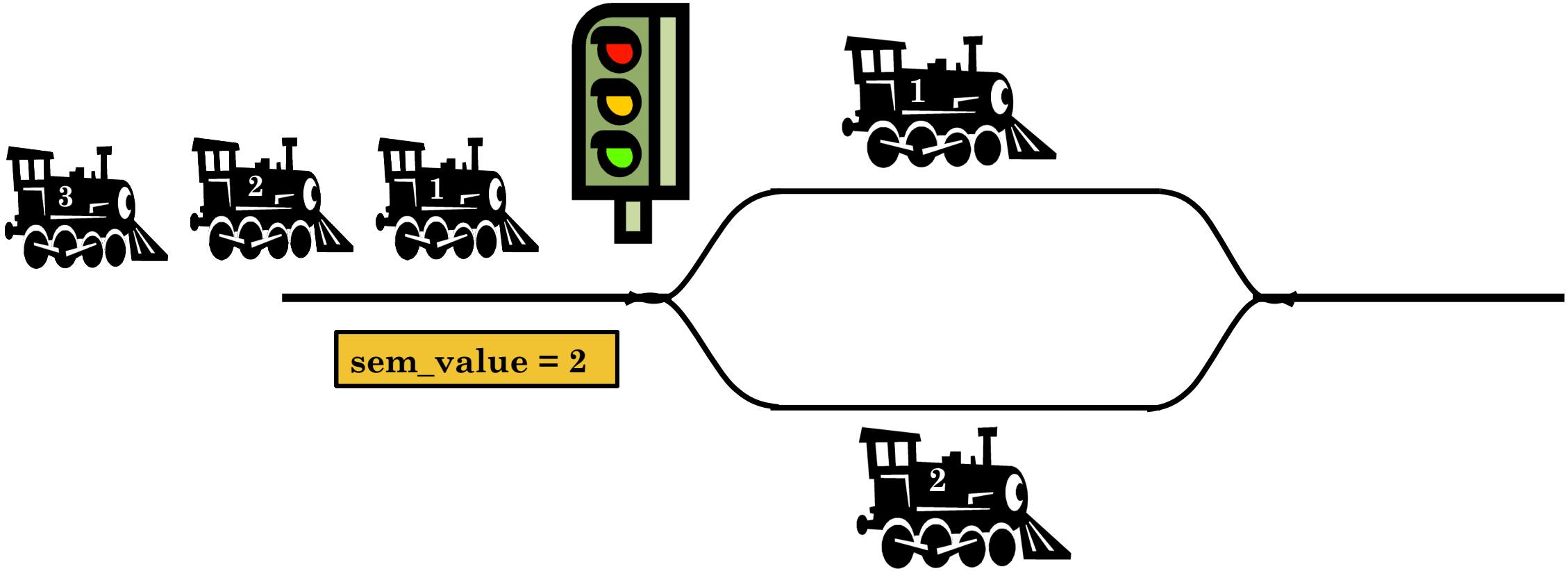
- **Mutex:** Must always be unlocked by the same thread that locked it (strict ownership model). So we use mutex, when the same thread enters and exits the critical section or while protecting data structures where one thread owns the entire operation.

- **Semaphore:** Can be posted by any thread, regardless of which thread performed the wait (no ownership concept). So we use semaphores when one thread produces/acquires, and another thread consumes/releases it. Moreover, we use semaphores, when a signal handler needs to release a lock (as only semaphore `post()` is signal-safe).

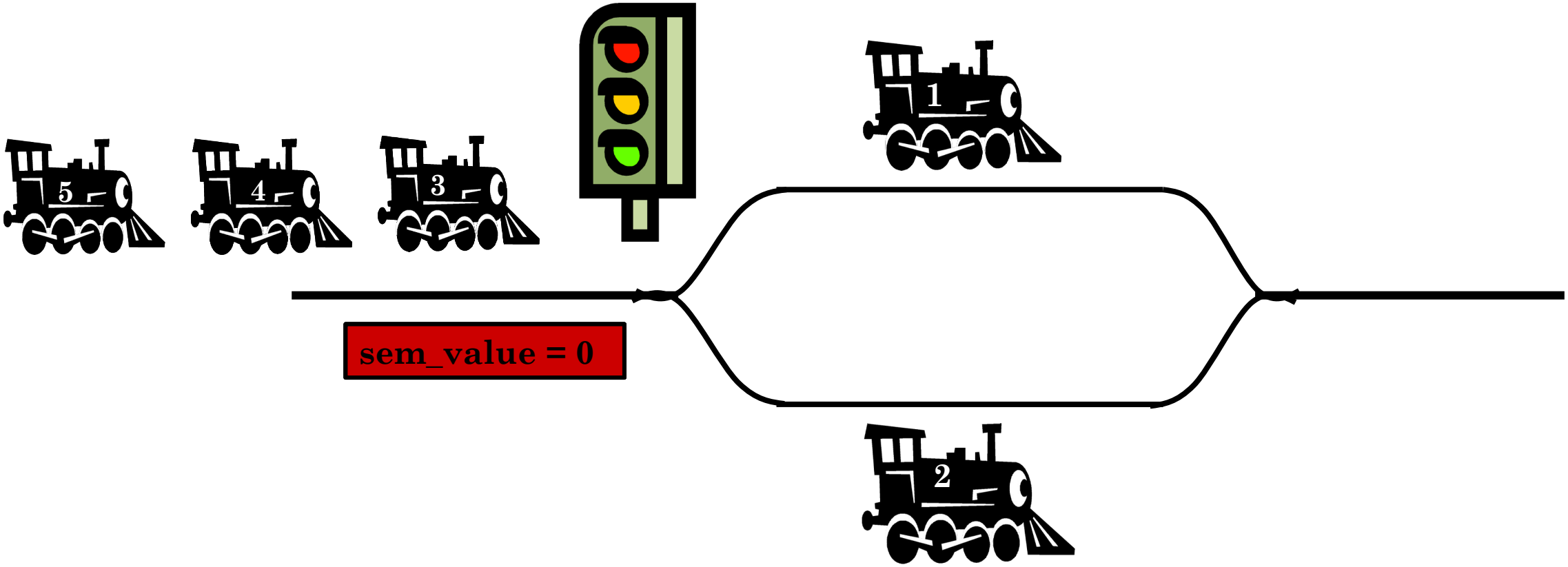
```
mutex m;
do {
    lock(m);
    <Critical Section>
    unlock(m);
    <Remainder Section>
} while (1);
```

```
semaphore s = 1;
do {
    wait(s);
    <Critical Section>
    post(s);
    <Remainder Section>
} while (1);
```

Counting Semaphores (Railway Analogy)



Counting Semaphores (Railway Analogy)



Signaling Semaphores (Serialization)

Example 1 Consider two processes P_1 and P_2 with statements A and B in them respectively. We want that statement <print B> in P_2 should be executed **after** statement <print A> in P_1 is executed. Give a semaphore based solution

```
Semaphore s = 0;
```

P_1

```
.
.
.
<print A>
post(s);
.
.
```

P_2

```
.
.
.
wait(s);
<print B>
.
.
```

Example 2 Consider three processes P_1, P_2 and P_3. Instruction A in P_1 executes after instruction B in P_2 is executed. Instruction B in P_2 executes after instruction C in P3 has executed. Give a semaphore based solution. (C < B < A)

System-V vs POSIX Semaphores

System V vs POSIX Semaphores

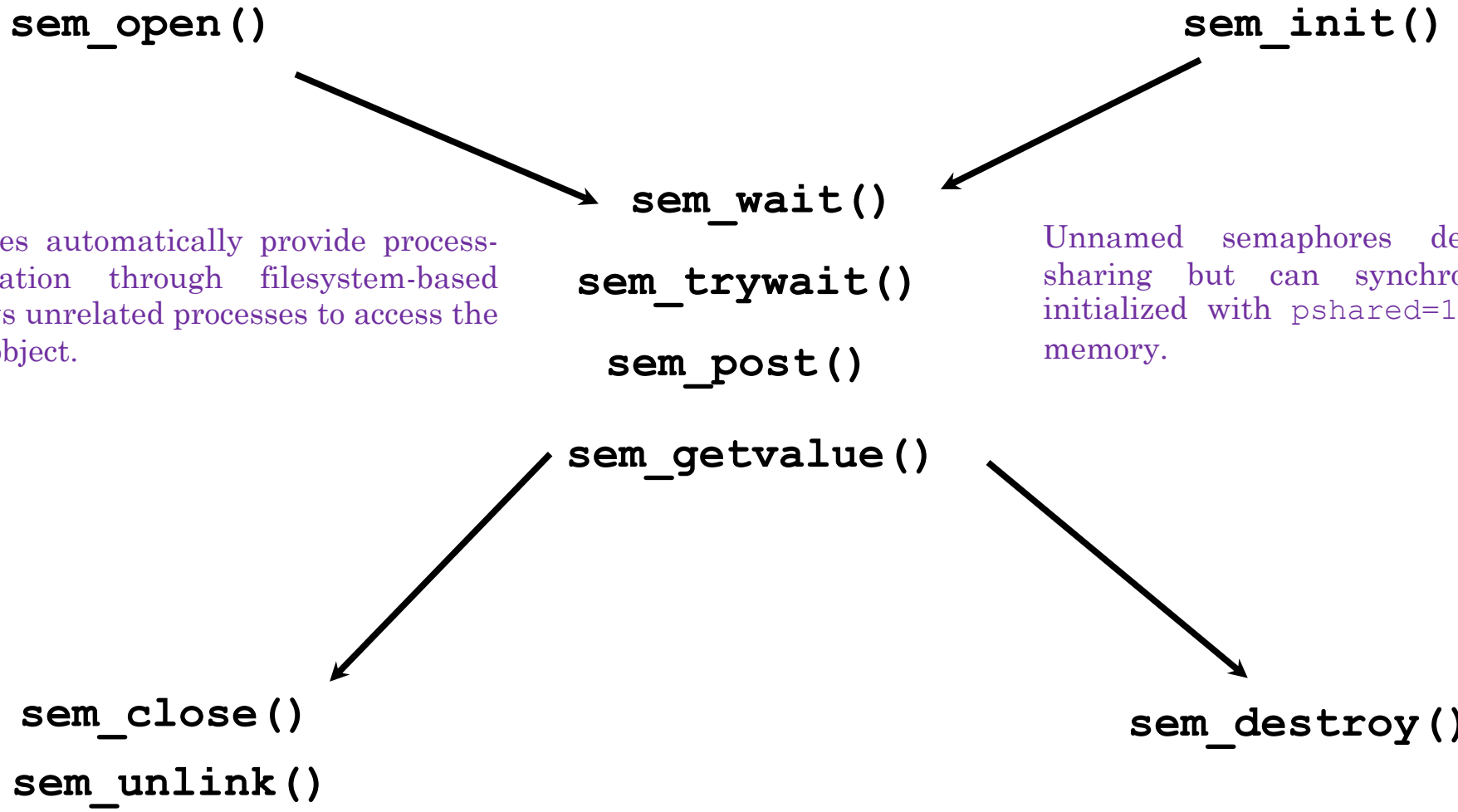
- **System V Semaphores:** Complex IPC mechanism with semaphore arrays, atomic multi-operations, and persistent lifecycle, but not signal-safe.
- **POSIX Semaphores:** Simple, modern API with signal-safe operations and automatic clean-up, but limited to single-semaphore operations.

Aspect	System V Semaphores	POSIX Semaphores
Header File	`#include <sys/sem.h>`	`#include <semaphore.h>`
Creation API	`semget()` with IPC keys	`sem_init()` / `sem_open()`
Data Structure	Semaphore arrays (sets)	Individual semaphores
Scope	Recommended for Inter-process	Named: inter-process, Unnamed: threads
Atomicity	Multiple semaphores atomically	Single semaphore only
Persistence	Survives process termination	Auto-cleanup on exit
Signal Safety	Not signal-safe	`sem_post()` is signal-safe
Cleanup	Manual (`semctl()` removal)	Automatic destruction
Complexity	High (IPC keys, arrays)	Low (simple functions)
Portability	Traditional Unix systems	POSIX-compliant systems

POSIX API for Semaphores

Named Semaphores

Unnamed /Memory Based Semaphores



Named semaphores automatically provide process-level synchronization through filesystem-based naming that allows unrelated processes to access the same semaphore object.

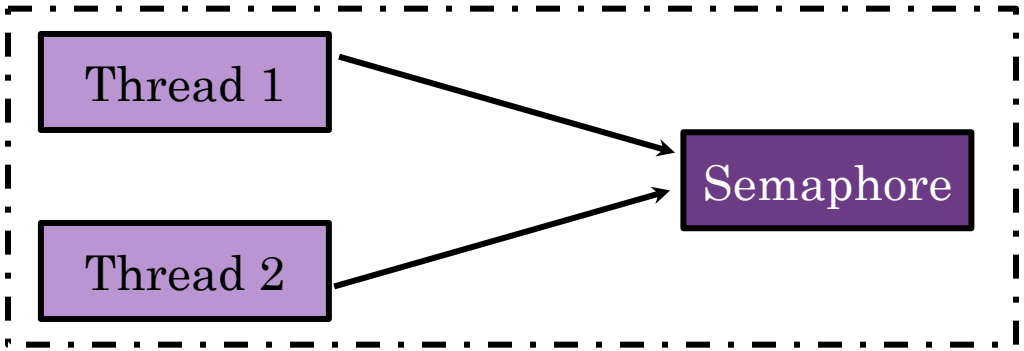
Unnamed semaphores default to thread-level sharing but can synchronize processes when initialized with `pshared=1` and placed in shared memory.

POSIX

Un-named Semaphores

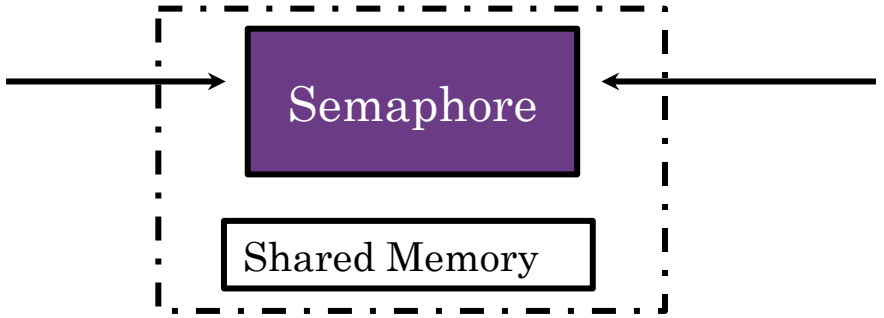
Creating Unnamed Semaphores

Memory Based Semaphore Shared between two Threads



One Process

Memory Based Semaphore Shared between two Processes



Creating Unnamed Semaphores

```
int sem_init(sem_t *sem, int pshared, int value);
```

- The `sem_init()` library call initializes the unnamed semaphore at the address pointed to by its first argument `sem` with `value` mentioned as third argument.
- If `pshared` is zero, then semaphore is shared between the threads of a process, and `sem` has to be global, so that it is accessible among all the threads of a process.
- If `pshared` is non-zero, then semaphore is shared between processes, and `sem` has to be located in a region of shared memory.
- After a successful call, the address of semaphore `sem` can be used as the argument to `sem_wait()` and `sem_post()` calls by the processes or threads.
- Initializing a semaphore that has already been initialized results in undefined behavior.

Incrementing, Decrementing Semaphore

```
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

- The **`sem_wait()`** library call decrements the semaphore pointed to by `sem`. If the semaphore value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until the value of semaphore value rises above zero.
- The **`sem_post()`** library call increments the semaphore pointed to by `sem`. If the semaphore's value becomes greater than zero, then another process or thread blocked in a `sem_wait()` call will be woken up and proceed to lock the semaphore.
- On success both the functions returns 0. On error, the value of the semaphore is left unchanged, a -1 is returned and `errno` is set to indicate the error.

Destroying Unnamed Semaphores


```
int sem_destroy(sem_t *sem);
```

- The `sem_destroy()` call destroys the unnamed semaphore at the address pointed to by `sem`. Only a semaphore that has been initialized by `sem_init()` should be destroyed using `sem_destroy()`.
- Destroying a semaphore that other processes or threads are currently blocked on produces undefined behavior.
- Using a semaphore that has been destroyed produces undefined results, until the semaphore has been reinitialized using `sem_init()`.
- On success the call returns 0. On error a -1 is returned, and `errno` is set to indicate the error.

Achieving ME using Un-named Semaphores

```
long balance = 0;
void * inc(void * arg);
void * dec(void * arg);
sem_t s;
int main(){
    pthread_t t1, t2;
    sem_init(&s, 0, 1);
    pthread_create(&t1, NULL, inc, NULL);
    pthread_create(&t2, NULL, dec, NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    sem_destroy(&s);
    printf("balance:%ld\n", balance);
    return 0;
}
```

```
void * dec(void * arg){
    for(long i=0;i<100000000;i++){
        sem_wait (&s);
        balance--;
        sem_post (&s);
    }
    pthread_exit(NULL);
}

void * inc(void * arg){
    for(long i=0;i<100000000;i++){
        sem_wait (&s);
        balance++;
        sem_post (&s);
    }
    pthread_exit(NULL);
}
```

Demonstration

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Serialization using Un-named Semaphores

```
void* f1(void * arg);
void* f2(void * arg);
void* f3(void * arg);
sem_t semA, semB;
int main(){
    pthread_t t1, t2;
    sem_init(&semA, 0, 0);
    sem_init(&semB, 0, 0);
    pthread_create(&t1, NULL, f1, NULL);
    pthread_create(&t2, NULL, f2, NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    pthread_join(t3,NULL);
    sem_destroy(&semA);
    sem_destroy(&semB);
    printf("balance:%ld\n", balance);
    return 0;
}
```

```
void * f1(void * arg){
    sem_wait (&semB);
    fprintf(stderr, "%s", " Arif Butt");
    pthread_exit(NULL);
}

void * f2(void * arg){
    sem_wait (&semA);
    fprintf(stderr, "%s", " fun with");
    sem_post (&semB);
    pthread_exit(NULL);
}

void * f1(void * arg){
    fprintf(stderr, "%s", "Learning is");
    sem_post (&semA);
    pthread_exit(NULL);
}
```

Demonstration

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

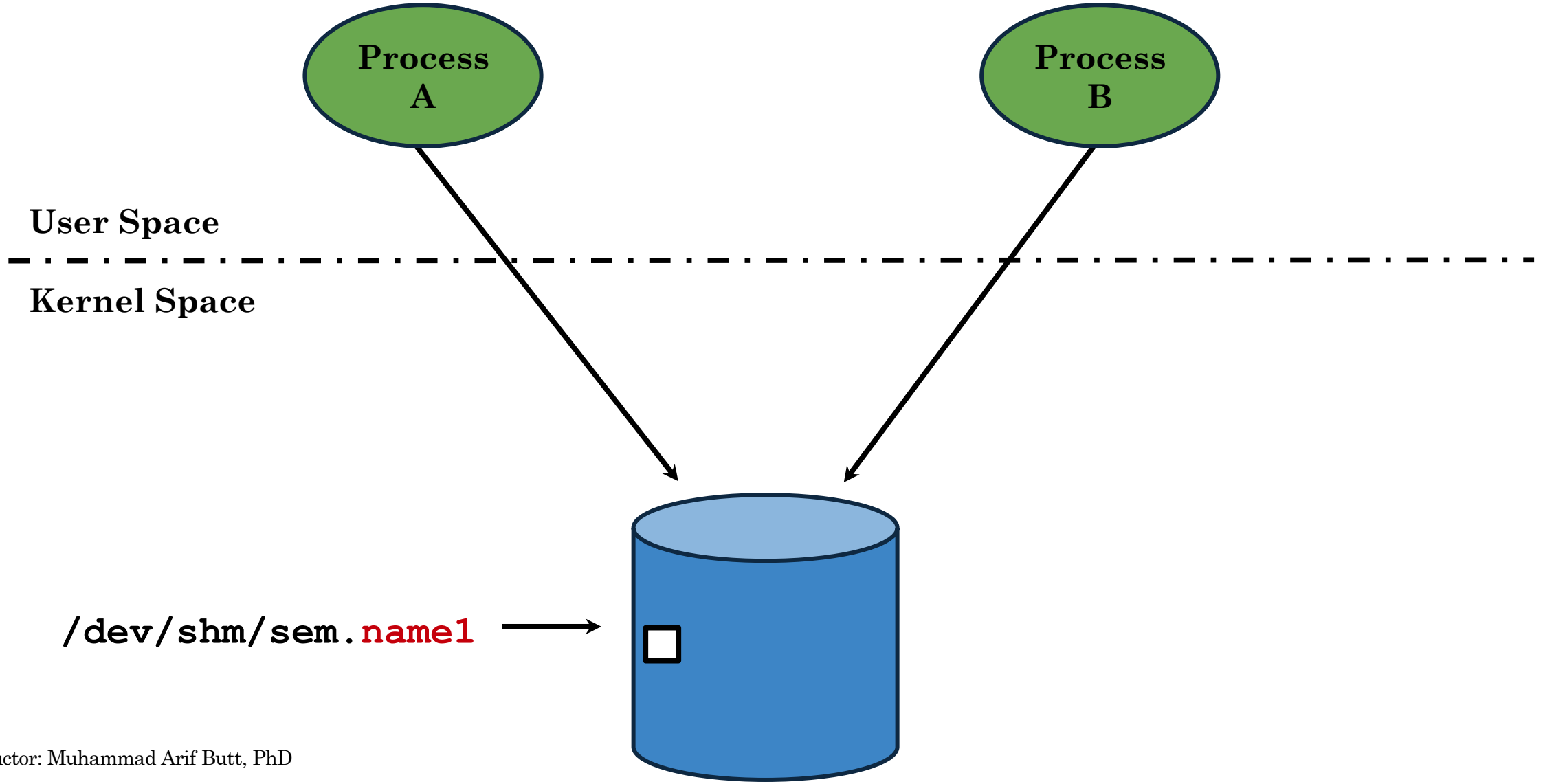
Demonstration

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*


POSIX

Named Semaphores

Creating a Named Semaphore



Creating a named Semaphore

```
sem_t *sem_open(const char *name, int oflag, mode_t mode, int value);
```

- The **sem_open()** library call creates a new semaphore or opens an existing semaphore identified by its first argument name of the form `/somename`, i.e., a null-terminated string of up to `NAME_MAX-4` (i.e., 251) characters consisting of an initial slash, followed by one or more characters, none of which are slashes.
- The second argument **oflag** is mostly `O_CREAT`, in which case the semaphore is created if it does not already exist. If both `O_CREAT` and `O_EXCL` are specified, then an error is returned if a semaphore with the given name already exists.
- If `O_CREAT` is specified in `oflag`, then two additional arguments must be supplied. The **mode** argument specifies the permissions to be placed on the new semaphore. The **value** argument specifies the initial value for the new semaphore. Binary semaphores usually have an initial value of 1, whereas counting semaphores often have an initial value greater than 1.
- The return value is a pointer to sem_t data type, which is then used as the argument to **sem_wait()**, **sem_post()** and **sem_close()** calls.

Closing and Removing a Named Semaphore

```
int sem_close(sem_t *sem);
int sem_unlink(const char *name);
```

- A un-named semaphore is automatically closed on process termination, while a named semaphore has to be closed by using the **`sem_close()`** library call and passing it the **`sem_t`** variable received via a previous **`sem_open()`** call.
- However, closing a named semaphore does not remove it from the system, as they are at least kernel-persistent. They retain their value even if no process currently has the semaphore open.
- So to remove a named semaphore from the system we can use the **`sem_unlink()`** call. A semaphore has a reference count of how many times they are currently open. Removing of semaphore from filesystem occur when the reference count becomes zero and also after the last process that has opened the semaphore calls **`sem_close()`**.
- On the shell on Linux, you can use the **`rm(1)`** command to delete the related file in the **`/dev/shm/`** directory.

Mutual Exclusion using Named Semaphores

```
void inc();
void dec();
long *balance;
sem_t *sem;
int main(){
  key_t key1 = ftok("file1", 65);
  int shm_id1=shmget(key1, 8, IPC_CREAT | 0666);
  balance = (long*)shmat(shm_id1, NULL, 0);
  *balance=0;
  sem = sem_open("/balance_sem", O_CREAT, 0666, 1);
  int cpid = fork();
  if (cpid == 0){
     inc();
     shmdt(balance);
     exit(0);
   }else{
     dec();
     waitpid(cpid,NULL,0);
     fprintf(stderr, "Balance is: %ld\n", *balance);
     shmdt(balance);
     shmctl(shm_id1, IPC_RMID, NULL);
     sem_close(sem);
     sem_unlink("/balance_sem");
     return 0;
  }
}
```

```
void inc(){
    sem_wait(sem);
    int temp = *balance;
    usleep(100000);
    temp = temp + 1;
    usleep(100000);
    *balance = temp;
    sem_post(sem);
    return;
}
void dec(){
    sem_wait(sem);
    int temp = *balance;
    usleep(100000);
    temp = temp - 1;
    usleep(100000);
    *balance = temp;
    sem_post(sem);
    return;
}
```

Demonstration

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Challenges with Synchronization Primitives

Synchronization Primitives: Challenges

- All synchronization primitives (mutexes, condition variables, barriers, semaphores) share fundamental challenge and that is they require distributed coordination across multiple threads with global correctness requirements.
- **Core Problems:**
  - *Scattered Operations:* Synchronization calls are distributed across multiple threads throughout the codebase, making system-wide analysis and debugging extremely difficult.
  - *Initialization and Lifecycle Errors:* Improper initialization, incorrect attributes, or resource management mistakes can cause undefined behavior or system instability.
  - *Global Correctness Requirement:* Every participating thread must follow the synchronization protocol perfectly—there is no partial correctness in concurrent systems.
  - *Catastrophic Failure Mode:* One programming error in any thread can cause system-wide deadlocks, data corruption, or complete application failure.
- Above problems may result in following issues:
  - *Violation of M.E:* Multiple threads in Critical Sections simultaneously.
  - *Deadlock:* Threads waiting for each other in circular dependencies.
  - *Starvation:* Thread(s) are indefinitely blocked due to unavailability of a resource.

Violation of Mutual Exclusion

Suppose the programmer wants to achieve mutual exclusion, but by mistake he/she has placed the post operation before the wait operation in P_0

- If P_1 executes first, decrements s to 0 and enter its CS. Suppose a context switch occur, and P_0 executes, and instead of a wait gives a post to s, and enter it CS. Both P_0 and P_1 are in CS, thus M.E is violated.

```
Semaphore s = 1;
```

P_0

```
post(s);
<Critical Section>
nothing/wait(s);
...
```

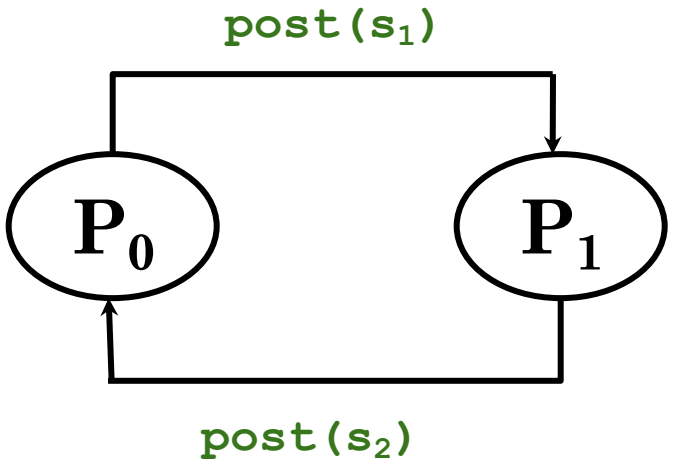
P_1

```
wait(s);
<Critical Section>
post(s);
...
```

Deadlock

Deadlock is a situation, where a set of processes are stuck forever waiting for each other in a circular wait → no progress possible.

```
semaphore s1 = 1;
semaphore s2 = 1;
```



P_0	P_1
`wait(s1);`	`wait(s2);`
`wait(s2);`	`wait(s1);`
.	.
`post(s1);`	`post(s2);`
`post(s2);`	`post(s1);`

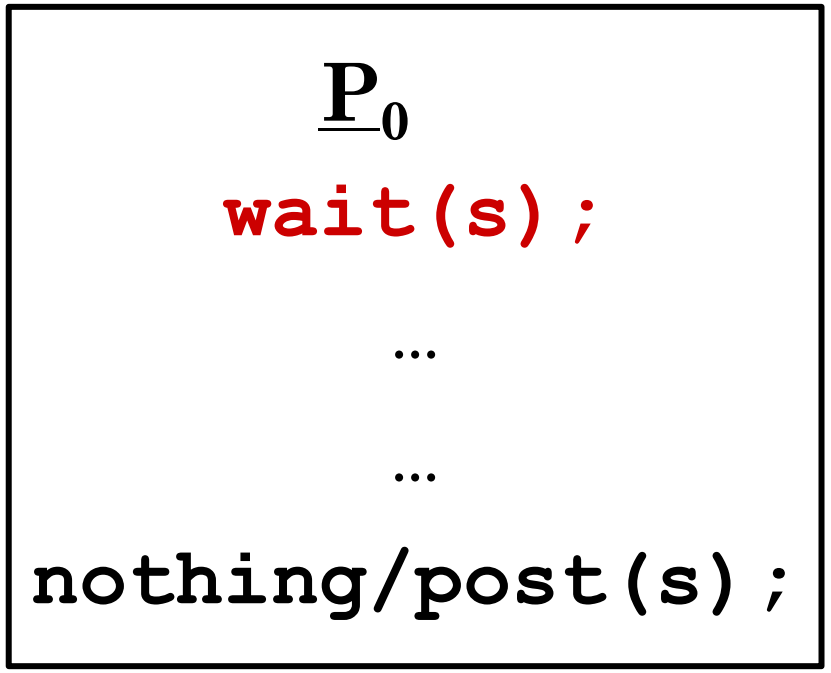
s1	s2
0	0

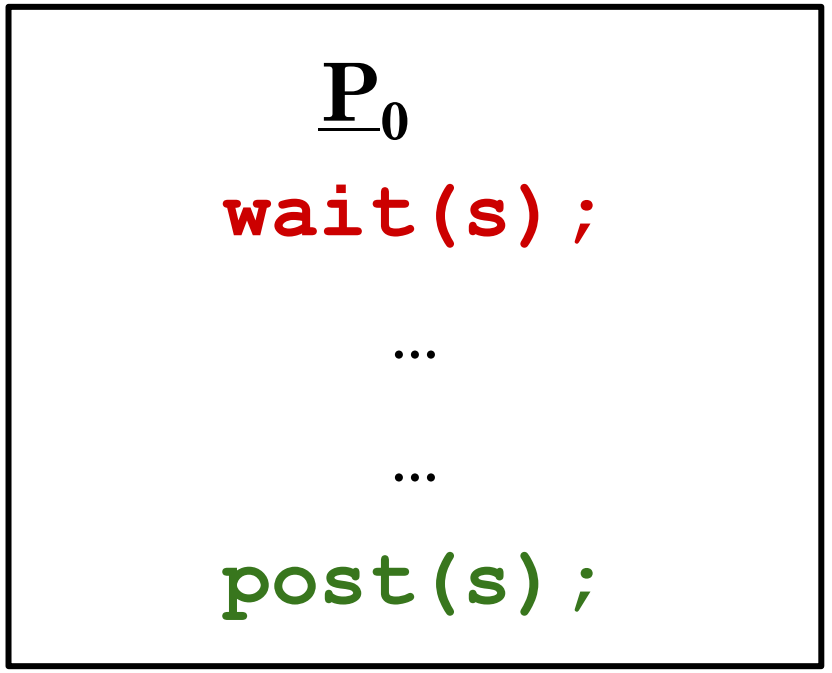
Blocked ...

Starvation

Starvation is a situation, where a process keeps waiting indefinitely because it is never chosen (unfair scheduling or resource allocation), even though progress by others continues.

- 👉 **Deadlock** = no one moves.
- 👉 **Starvation** = some move, one starves.





Solutions and Best Practices

One possible solution is use Compiler-level synchronization primitives, which shift the responsibility of enforcing M.E / serialization form the programmer to the compiler. Instead of writing complex locking code manually, developers can use simple annotations or directives. These tools automatically generate the necessary synchronization logic behind the scenes. Examples include:

- Java's `synchronized` keyword, which locks objects implicitly to prevent concurrent access.
- GCC's `__transaction_atomic`, which allows blocks of code to run atomically, and rolling back changes if a conflict is detected.
- C#'s `lock(object)` statement, which generates `Monitor.Enter()/Monitor.Exit()` calls wrapped in `try-finally` blocks to guarantee lock release even during exceptions.

Best Practices:

- *Proper Initialization:* Always initialize synchronization objects before use and destroy them properly.
- *Correct Attributes:* Use appropriate mutex types, semaphore initial values, and condition variable configurations.
- *Resource Management:* Implement proper clean-up and avoid leaks using RAII patterns where possible.
- *Universal Compliance:* Ensure all threads follow the synchronization protocol without exception.
- *Error Handling:* Design robust error paths that maintain synchronization invariants.
- *Testing and Validation:* Use thread sanitizers, stress testing, and static analysis tools.

To do

- Watch SP video on Programming with POSIX Semaphores https://www.youtube.com/watch?v=KupTFYvxRnE

Coming to office hours does NOT mean that you are academically weak!