System Programming - Linux
Week 12 - Semaphores

Lecturer: Dr. Aggrey Obbo (PhD)

June 13, 2025

Contents

1 Introduction

Linux semaphores serve as vital Inter-Process Communication (IPC) tools. They function as signaling devices, enabling processes to coordinate access to shared resources and avoid race conditions. This part examines their implementation and application, emphasizing how they facilitate organized collaboration among concurrent processes in a Linux setting.

Objectives

- Understand the basics of Semaphores and their function in synchronizing processes.
- To describe the different types of semaphores and clarify the distinctions between binary semaphores (mutexes) and counting semaphores.and their use cases.
- Implement Semaphores in a Linux Environment: Explain how to create, utilize, and manage semaphores in a Linux setting, including relevant system calls.
- Tackle Traditional Synchronization Challenges: Utilize semaphore principles to solve frequent concurrency problems such as the Producer-Consumer issue and the Dining Philosophers dilemma.
- Highlight Possible Challenges: Address common problems related to semaphore usage, such as deadlocks and starvation, along with methods to prevent them.

2 Concurrency and Synchronization

Concurrent processes or threads refer to several instruction sequences that seem to execute simultaneously. True simultaneous execution (parallelism) necessitates multiple CPU cores; however, concurrency on a single core is accomplished through time-slicing, in which the operating system alternates between tasks, creating the illusion of simultaneous activity. Threads are more lightweight units of execution within a single process, sharing the same memory space, whereas processes maintain their own separate memory.

2.1 Benefits concurrency and Synchronization

:

- Enhanced Throughput: Concurrency allows a system to accomplish more tasks within a specific timeframe. By overlapping operations, particularly those dependent on I/O, the CPU can shift to other ready tasks instead of remaining idle. For instance, a web server can manage numerous client requests concurrently, boosting the overall speed at which requests are processed.
- User Responsiveness: Concurrency maintains the interactivity of applications and prevents them from becoming unresponsive. If a time-consuming task (like downloading a file or performing a complicated calculation) is executed in a separate thread, the main thread can continue processing user input and refreshing the user interface, ensuring a seamless experience.
- Efficient Resource Usage: Running multiple tasks concurrently leads to better utilization of system resources such as CPU cores, memory, and I/O devices. Idle resources from one task can be redirected to another, reducing wasted cycles and increasing the total amount of work completed by the system

2.2 Multi-core Processors, Multi-threaded Applications

A multicore processor in Linux is defined as a CPU with several distinct processing units (cores) integrated onto one chip. Each core has the capability to execute instructions at the same time, enabling the Linux kernel to run multiple processes or threads in genuine parallelism, which greatly improves system performance and responsiveness, especially under heavy workloads.

In the Linux environment, a multicore processor is a solitary CPU chip that features two or more separate "cores" or processing units. Each core can independently execute instructions, which allows the operating system to execute several tasks or threads completely at once. This results in improved performance for multitasking and resource-intensive applications, leading to a faster and more efficient Linux system.

In Linux, a multi-threaded application is a single program that breaks its functions into smaller, independent units known as "threads." These threads execute concurrently within the same process, sharing resources such as memory. This enables the application to handle several tasks as if they were happening at the same time, enhancing responsiveness and efficiency, particularly on multi-core processors.

2.3 Challenges with Concurrency

Concurrency in Linux refers to the simultaneous execution of multiple tasks (processes or threads). While this enables effective utilization of system resources, it brings about significant challenges.

Consider two chefs attempting to use the same mixing bowl in a kitchen at precisely the same time. Without proper coordination, one might remove ingredients while the other adds them, resulting in a ruined recipe.

Likewise a race condition in Linux arises when the result of operations relies on the unpredictable timing of several tasks that access and alter shared resources If one task reads data while another is in the midst of updating it, the first task may obtain incorrect information, which can lead to errors in the program or system as a whole.

2.4 Critical Section

A critical section refers to a portion of code that interacts with shared resources such as variables, data structures, or hardware devices. The main issue with concurrent access to these shared resources is that if several processes or threads attempt to modify them simultaneously, it can result in race conditions and data inconsistency.

To avoid this, a critical section must be safeguarded so that only one process or thread can execute within it at any given time. Solutions to the problem of critical sections must fulfill three essential properties:

1. Mutual Exclusion: This is the fundamental principle. It asserts that if one process is executing its critical section, no other process is permitted to execute its critical section. This guarantees that shared resources are utilized by only one process at a time, helping to prevent data corruption and race conditions.

2. Progress: If no process is in its critical section and one or more processes wish to enter their critical sections, only those processes that are not engaged in their "remainder sections" should be able to take part in deciding which process will enter the critical section next. Additionally, this decision-making cannot be delayed indefinitely. The system should keep making progress by allowing a waiting process to go in.

3. Bounded Waiting: This property guarantees fairness and eliminates starvation. It stipulates that there must be a limit (or "bound") on the number of times other processes can enter their critical sections after a request has been made by a process wanting to enter its critical section and before that request is granted. Essentially, a process should not have to wait indefinitely to get its turn.

Synchronization is crucial for concurrent systems as it offers methods to control access to shared resources. By regulating the manner and timing in which tasks engage with these resources, synchronization avoids race conditions, thereby maintaining data consistency and integrity. Tools such as locks, *semaphores*, and mutexes are vital instruments that impose correct sequencing and mutual exclusion, thus alleviating frequent concurrency challenges.

3 Semaphores

Semaphores are essential synchronization mechanism that was created by Dutch computer scientist Edsger Dijkstra in the early 1960s during the development of an operating system. The goal was to address issues such as race conditions in concurrent systems.

In Linux, semaphores function as counters for shared resources, or "sleeping locks." They are effectively integer variables that can be increased or decreased.

A "counting semaphore" permits a limited number of processes to access a resource simultaneously. If a process attempts to "wait" on a semaphore when its count is zero, it is put to sleep, which avoids busy-waiting and saves CPU cycles. When another process "signals" the semaphore, the count increases, possibly waking a waiting process.

Linux offers both System V and POSIX semaphores for inter-process communication (IPC) and thread synchronization, which are vital for overseeing shared resources and preventing data corruption in multi-tasking environments.

3.1 Components of a Semaphore

Semaphores in Linux help regulate access to shared resources across multiple processes or threads, thereby preventing race conditions and maintaining data integrity. They function as integer variables that can only be manipulated through two atomic actions: wait (often referred to as P or down) and signal (known as V or up).

The key elements of a semaphore in Linux include:

1. **Integer Value:** At its essence, a semaphore stores an integer value, which signifies the number of resources or permissions currently available depending on the type of semaphore.

   Binary Semaphores (Mutexes): When the integer value is confined to either 0 or 1, it is referred to as a binary semaphore or mutex (mutual exclusion). A value of 1 indicates the resource is available, while 0 means it is currently occupied.

   Counting Semaphores: If the integer value can take on any non-negative integer, it is a counting semaphore. This type governs access to a resource that has multiple identical copies, with the value indicating how many instances are free.

2. **wait() (P / down) Operation**: This action decreases the semaphore's value.

   If the resulting value remains non-negative after the decrease, the process/thread may proceed, meaning it has successfully acquired a resource or permission. If the value turns negative, it indicates that no resources are available, causing the process/thread to be blocked (placed in a waiting queue) until a resource is freed. The Linux kernel's scheduler typically manages this blocking.

3. **signal() (V / up) Operation:** This action increases the semaphore's value.

   Incrementing the value indicates that a process/thread has released a resource or permission. If there are any processes/threads in a blocked state waiting on this semaphore, one of them (usually the one waiting the longest) will be unblocked and moved to the ready queue, allowing it to obtain the resource.

4. **Waiting Queue**: Each semaphore has an associated queue of processes or threads that are currently blocked because they called wait() when the semaphore's value did not allow for immediate access. The kernel oversees this queue, ensuring fairness in how resources are allocated.

It can be concluded that semaphores offer an effective way for processes to synchronize their actions, ensuring that critical sections of code are executed by only one process at a time in case of binary semaphores and that resource constraints are adhered to incase of counting semaphores. Linux provides system calls like semget(), semop(), and semctl() for creating, manipulating, and managing System V semaphores, whereas POSIX semaphores deliver a portable API.

3.2 Fundamental Semaphore Operations illustrated

Visualize a number that indicates the total "slots" accessible for a particular purpose.

wait() (or P()): This function is akin to attempting to acquire a slot. You inquire, "Is there a slot available?" If the answer is yes, you take one, which decreases the count of available slots. If no slots are available, you stand by until one opens up.

signal() (or V()): This function is similar to relinquishing a slot. You declare, "I have finished with my slot!" The quantity of available slots increases. If there were individuals waiting, one of them can now seize the newly available slot and move forward.

These two operations, acquiring and releasing, are conducted in a manner that avoids chaotic conflicts, ensuring efficient sharing of resources.

Semaphores illustrated Consider a common resource, such as a public rest room or toilet. Numerous individuals may want to use it, but only one can do so at any given time. A semaphore functions like a doorman for this rest room, monitoring who is allowed to enter.

There are two primary functions:

"Wait" (or "P" / "down"): When someone wishes to use the rest room, they "wait" on the semaphore. The doorman verifies if the rest room is available. If it is, they can enter, and the doorman marks the rest room as in use. If it is already occupied, the doorman instructs them to wait in line until it becomes available.

"Signal" (or "V" / "up"): Once someone is done using the rest room, they "signal" the semaphore. The doorman now understands that the rest room is available. If there are others waiting, the doorman allows the next person to enter.

This doorman guarantees that only one person utilizes the rest room at any time, preventing disorder and maintaining organization. Semaphores operate in much the same manner for computer programs, regulating access to shared data or devices to prevent conflicts.

Why gain permission

Gaining access to a resource or permission is crucial in computing to avoid disorder and ensure smooth operation. It refers to a process or thread obtaining exclusive or regulated access to necessary elements, such as a data set,

a printer, or a segment of code. This mechanism stops multiple entities from attempting to modify the same data at the same time, which could result in errors or corruption. By obtaining the "right" to utilize a resource, we ensure that only one authorized party accesses it at any given moment, or that its usage is controlled according to established guidelines, preserving the system's consistency and integrity.

Why release a resource
It is essential to release a resource to maintain system efficiency and stability. When a program completes its use of a resource, it must "return it." If this doesn't happen, "resource leaks" occur, where the resource stays allocated even if it's no longer utilized. This situation can exhaust available resources, hinder system performance, lead to failures in other programs, or even crash the entire operating system. Properly releasing resources ensures they are reused and accessible for other programs, enhancing system performance and preventing resource depletion.

Initializing Semaphores Semaphores are set up with an integer value that determines their "initial count." This starting count is essential as it establishes the initial condition of the shared resource protected by the semaphore.

In the case of mutual exclusion (binary semaphores): they are usually set to 1. This indicates that the resource is initially available, allowing one process to acquire it right away. For counting available resources: they are set to represent the total number of instances available for that resource. For instance, if there are 5 printers, the semaphore managing them would be set to 5. For signaling/synchronization (producer-consumer): they might be established at 0. This signifies that no resources are currently available, and any process attempting to wait() will block until another process signal()s that a resource has been made available. This initial count determines how many wait() operations can be performed successfully without blocking immediately after creation.

4 System V Vs POSIX Semaphores

System V Semaphores are older and more intricate, often deemed "heavier." They function on groups of semaphores, allowing the creation of an array with a single function call. This offers versatility for intricate synchronization tasks but results in a somewhat complex API (semget, semop, semctl). A significant characteristic is SEM_UNDO, which tries to automatically revert semaphore actions if a process halts unexpectedly, helping to avoid deadlocks. Nevertheless, operations frequently necessitate context switches to the kernel, which can adversely affect performance.

POSIX Semaphores are more recent, less complicated, and generally favored for contemporary Linux development. They manage individual semaphores rather than groups. There are two varieties: named (which have a file-like path in /dev/shm and persist beyond process exits unless unlinked) and unnamed (which are found in shared memory, specific to processes, and removed upon process termination). POSIX semaphores provide a clearer API (sem_open, sem_wait, sem_post, sem_close, sem_unlink, sem_init, sem_destroy) and tend to be more efficient due to optimizations in user space (utilizing futexes on Linux). They also offer greater portability across Unix-like operating systems.

5 POSIX Semaphores

POSIX semaphores provide a more straightforward and portable application programming interface compared to System V. They utilize either named or unnamed semaphores directly, which eliminates the complicated semget/semop functions and kernel-managed groups found in System V. This approach minimizes overhead while improving compatibility across different platforms.

5.1 Types of POSIX Semaphores

Named Semaphores
Named semaphores allow even unrelated processes to coordinate by using a shared name, in the same manner a file is accessed. And the functions used are as follows:

sem_open(): This function serves as a "gateway" to the semaphore. It is called to create a new named semaphore if it does not already exist or to access an existing one. It returns a pointer to the semaphore, which the process will use for sem_wait() and sem_post() functions. sem_wait() decreases the value of a semaphore, causing the calling process to wait if the value is zero, thus guaranteeing the availability of a resource. sem_post() increases the semaphore's value, which may allow a blocked process to proceed, indicating that a resource has been released.

sem_close(): When a process is finished using a named semaphore, it invokes sem_close(). This effectively "shuts the door" for that specific process. It reduces the semaphore's open count, but the semaphore continues to exist in the system.

sem_unlink(): This function acts as the "cleanup crew" for named semaphores. sem_unlink() removes the name of the semaphore from the system, making it unavailable for subsequent sem_open() calls. The actual resources of

the semaphore are only released once all processes that have it open have also called sem_close().

Unnamed POSIX Semaphores
Unnamed semaphores function like basic, localized "traffic lights" for coordination. Unlike named semaphores that can be utilized by different, unrelated programs, unnamed semaphores are primarily employed within a single process to synchronize its own threads by default intentionally share a segment of memory. This indicates they have already established a designated memory area that both can access. And their functions are as follows:

sem_init(sem, pshared, value): This is the method to "initialize" an unnamed semaphore. sem refers to the actual semaphore variable in memory, value establishes the initial count of available "tickets." and pshared argument defines its range:

If pshared is set to 0, the semaphore is meant for threads within the same process. It acts like a personal traffic light for threads in one location. If pshared is a non-zero value, the semaphore is designated for processes that share a particular memory area. This signifies that processes have intentionally created a shared memory space for communication.

sem_destroy(sem) is called to "remove" the thread and free its resources. It is important to destroy it before the memory it occupies is deallocated. If other threads or processes are still dependent on it, destroying it can result in undefined behavior or errors.

5.2 Key Operations of POSIX Semaphores

Linux semaphores are essential synchronization tools that regulate access to shared resources between processes or threads, thereby preventing race conditions. They primarily operate with a counter that indicates the availability of a resource.

sem_wait() decrements semaphore count, block if 0. It is equivalent to P operation. This function tries to reduce the value of the semaphore. If the semaphore's current value is above zero, it gets decreased, allowing the executing thread/process to continue. However, if the semaphore's value is zero, which signifies that the resource is unavailable, sem_wait() will block the calling thread/process until the semaphore's value returns above zero (signifying the resource is now available). This mechanism ensures that mutual exclusion or resource limitations are maintained.

sem_post() Increment semaphore count, unblock a waiting thread/process. This function increases the value of the semaphore. If there are any threads or processes currently halted by a sem_wait() on the same semaphore, sem_post() will release one of them. This signals that a resource has been freed or an event has taken place, allowing a waiting entity to move forward.

sem_trywait() is a non-blocking attempt to decrement. In a manner similar to sem_wait(), sem_trywait() tries to decrease the semaphore's value. However, if the semaphore's value is zero, instead of waiting, sem_trywait() will return immediately with an error. This is beneficial for scenarios where a process needs to verify resource availability without stopping its execution.

sem_getvalue() is a get current value of the semaphore. This function obtains the current value of the semaphore and stores it in the provided integer pointer. This enables a thread or process to examine the state of the semaphore without changing its value or blocking. It is often utilized for debugging or logging purposes.

5.3 Advantages of POSIX Semaphores

POSIX semaphores provide significant benefits compared to the older System V semaphores, mainly due to their design philosophy tailored for contemporary operating systems.

More Intuitive API: POSIX semaphores offer a more straightforward and user-friendly Application Programming Interface (API). Instead of working with complicated semget, semctl, and semop functions that handle groups of semaphores, POSIX delivers direct functions such as sem_open, sem_close, sem_unlink, sem_init, sem_destroy, sem_wait, and sem_post. This simplified approach lowers the learning curve for developers and reduces the chances of mistakes, making it easier to write and maintain synchronization logic.

Enhanced Portability: POSIX (Portable Operating System Interface) is a commonly accepted standard that outlines the expected behavior of operating systems. By following this standard, POSIX semaphores guarantee that code developed for one POSIX-compliant system (such as Linux, macOS, or various Unix-like systems) will typically function with little or no alteration on another. This stands in stark contrast to System V semaphores, which are often linked to specific Unix implementations and may show variability in behavior across different platforms, complicating code portability.

Clear Mapping to Threads/Processes: POSIX semaphores are crafted to integrate effortlessly with both threads within a single process in case of unnamed semaphores and disparate processes that share memory or employ named

semaphores. The pshared parameter in sem_init indicates this scope, making it easier for developers to select the appropriate synchronization method. This direct relationship allows for effective management of concurrency, whether within a multi-threaded application or across separate programs that need to harmonize access to shared resources.

5.4 Disadvantages of of POSIX Semaphores

Although POSIX semaphores are simpler than those in System V, they still present considerable difficulties mainly because of their low-level characteristics.

Complex and Error Prone
Managing multiple semaphores, especially across many threads or processes, can rapidly become complicated. This complexity can directly result in various common and hard-to-identify errors:

Deadlock: This arises when two or more processes or threads become stuck indefinitely, each waiting for a resource that another holds. This situation often occurs due to improper sequences of sem_wait() and sem_post(), resulting in a circular dependency.

Starvation: A process or thread may be repeatedly denied access to a resource, despite its availability, as other processes constantly acquire the semaphore first. This scenario can happen in systems with heavy contention where scheduling favors specific tasks.

Priority Inversion: A significant problem where a high-priority task must wait for a lower-priority task that has a critical resource and is subsequently preempted. This situation compromises the predictability of real-time systems.

Signal/Wait Mismatches: Simple mistakes, such as neglecting to call sem_post() after sem_wait(), can lead to a semaphore being permanently locked, halting execution. Alternatively, invoking sem_post() too many times can disrupt key sections of code.

Lower-Level Primitive and Debugging Difficulties
Semaphores are a very fundamental synchronization mechanism. For numerous common concurrency patterns, higher-level constructs such as mutexes, condition variables, or read-write locks provide more dependable and less error-prone solutions. These constructs often encapsulate the correct semaphore logic, thus minimizing the likelihood of the errors discussed previously. When problems do occur with semaphore-based synchronization, they prove extremely challenging to debug, as the issues tend to present themselves as intermittent hangs or unexpected behaviors rather than a definitive crash, making root cause analysis a lengthy and frustrating task.

6 Semaphore Use Cases and Examples

6.1 Mutual Exclusion or the Critical Section Problem

One of the primary applications of POSIX semaphores is to establish mutual exclusion, often referred to as addressing the Critical Section Problem. As an example, consider a shared like a printer or any other global variable, that various threads or processes need to use. If multiple entities attempt to modify this resource at the same time, it could result in a "race condition," where the resulting state of the resource becomes unpredictable and incorrect.

The "critical section" refers to the part of the code where this shared resource is accessed. To avoid race conditions, it is essential to ensure that only one thread or process can execute its critical section at any given moment. This is where a binary semaphore (initialized to 1) plays a crucial role.

A semaphore is set to 1 initially, indicating that the critical section is available for access. Before entering the critical section, a thread or process invokes sem_wait(). This function reduces the semaphore's value. If the value reaches 0, it indicates that the critical section is now "locked," and any subsequent sem_wait() calls from other threads will be blocked, forcing them to wait until the semaphore's value is above zero.

After a thread has completed executing its critical section, it calls sem_post(). This increases the semaphore's value, "unlocking" it. If other threads were waiting on the semaphore, one of them will now be unblocked and permitted to enter its critical section. This protocol effectively ensures that only one thread or process can access the shared resource at any time, thus preventing data corruption and maintaining consistent behavior.

6.2 Producer-Consumer Problem

The Producer-Consumer Problem is a well-known synchronization issue in which "producers" create data items and store them in a shared, limited buffer, while "consumers" fetch and process these items from the same buffer. The

main challenge is to ensure that:

- Producers do not attempt to add items to a full buffer.
- Consumers do not attempt to remove items from an empty buffer.
- Access to the buffer itself is exclusive to prevent race conditions during the processes of adding or removing items.

To address this challenge, POSIX semaphores employ three semaphores:

1. mutex in case of a binary semaphore is initialized to 1: This semaphore ensures mutual exclusion when accessing the shared buffer. Both producers and consumers must acquire this semaphore using sem_wait(mutex)) before they modify the buffer and release it with sem_post(mutex) afterward. This measure prevents simultaneous reads and writes that could lead to data corruption.
2. empty (A counting semaphore is initialized to the size of the buffer): This semaphore monitors the number of available slots in the buffer. A producer invokes sem_wait(empty) prior to adding an item. If the buffer is full, the producer will block until a slot is free. Once an item is added, the producer calls sem_post(full).
3. full (a counting semaphore, initialized to 0): This semaphore keeps track of the number of occupied slots in the buffer. A consumer executes sem_wait(full) before removing an item. If the buffer is empty (i.e., full is 0), the consumer will block until an item is available. After consuming an item, the consumer calls sem_post(empty).

By integrating these semaphores, POSIX delivers an efficient solution that guarantees proper synchronization and prevents buffer overflow, allowing for smooth data transfer between producers and consumers.

6.3 Dining Philosophers Problem - Deadlock Scenario

The Dining Philosophers Problem is a concurrency issue that illustrates the risk of deadlock when shared resources are not handled properly. It is illustrated as five philosophers seated around a circular table, with one chopstick placed between each pair of them. To eat, a philosopher must obtain both the chopstick on their left and the chopstick on their right.

If every philosopher uses a basic approach – picking up their left chopstick first, followed by their right chopstick – a deadlock is likely to occur. For instance, suppose all five philosophers simultaneously grab their left chopstick. Each philosopher would then be holding one chopstick while waiting indefinitely for the right chopstick, which is currently held by their neighbor. As there is no way for any philosopher to secure their second chopstick, eating becomes impossible, and no one will release their held chopstick. The system remains stuck in a continuous waiting state, clearly demonstrating the concept of deadlock.

POSIX semaphores can be employed to represent this issue, with each chopstick symbolized by a binary semaphore (set to 1 initially). A straightforward solution that uses sem_wait() on the left fork followed by the right fork will indeed cause deadlock. Addressing the Dining Philosophers Problem with semaphores generally involves methods to eliminate one of the four criteria for deadlock (mutual exclusion, hold and wait, no preemption, circular wait). Common approaches include:

- Restricting the number of philosophers: Permitting a maximum of (N-1) philosophers to attempt eating at the same time. Where N is the number of chop sticks
- Asymmetrical grabbing: Philosophers in even-numbered positions pick up their left fork first, then their right fork, while those in odd-numbered positions pick up their right fork first, then left.
- Arbitrator/Waiter: Implementing a shared semaphore or a "waiter" who allows eating only if both forks are accessible.

Semaphores guarantee synchronized access in the producer-consumer scenario, enforce mutual exclusion via mutexes, and ingeniously address deadlocks in the dining philosophers challenge, making semaphores indispensable. They offer strong solutions to avoid race conditions and promote efficient resource usage in intricate multi-threaded settings.

6.4 Best Practices and Pitfalls

Best Practices

For effective semaphore-based synchronization, the following suggestions could be beneficial:

Encapsulation: Develop functions or classes that encapsulate sem_wait() and sem_post(). This conceals low-level intricacies, resulting in cleaner and less error-prone code.

Clear Ownership: Clearly specify which thread or process is responsible for signaling (sem_post()) and waiting (sem_wait()) on each semaphore. Lack of clarity can lead to misuse and bugs.

Resource Ordering: When obtaining multiple semaphores, always do so in a predetermined and consistent sequence across all threads/processes. This is vital for avoiding deadlocks.

Error Checking: It is essential to verify the return values of all sem_ functions. Neglecting this can obscure significant issues that cause unexpected behavior.

Timeouts: Prefer using sem_timedwait() instead of sem_wait() when feasible. This helps prevent a thread from becoming blocked indefinitely if a semaphore is never signaled, allowing for effective error handling or alternative actions.

Common Pitfalls

Concurrency problems can arise due to pitfalls associated with semaphores:

Failing to initialize: Semaphores that are not initialized can exhibit undefined behavior, frequently resulting in crashes or erratic synchronization.

Incorrect starting values: A binary semaphore set to 0 instead of 1, or a counting semaphore initialized with an incorrect count, can quickly lead to deadlocks or permit excessive access.

signal() prior to wait(): If sem_post() is executed before sem_wait() when the sequence is expected to be the other way around, it can disrupt the synchronization logic and cause race conditions.

Neglecting to release: Omitting sem_post() after sem_wait() keeps a semaphore permanently locked, which results in indefinite blocking or deadlocks.

Releasing without acquisition: Invoking sem_post() on a semaphore that hasn't been acquired by the current thread can disrupt its state or trigger unexpected unlocks, breaching mutual exclusion.

Confusing System V with POSIX: These APIs are separate; mixing them within the same application can result in compilation errors, runtime problems, or unpredictable behavior due to their different internal structures.

7 Lab Exercise

Instruction

1. Use the code below to initiate a semaphore and save it as a .c file
2. Compile the program
3. Run the program in the background
4. To observe the created semaphore, use the command "ipcs -s"
5. Bring the program to the foreground and terminate it.(Use fg and then Ctrl+C)
6. Remove the semaphore using ipcrm -s semID
7. Verify using ipcrm -s semID

Sample Semaphore Code

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <errno.h> // For errno

    // Define union semun for semctl
union semun
int val; /* used for SETVAL only */
struct semid_ds *buf; /* for IPC_STAT and IPC_SET */
unsigned short *array; /* for GETALL and SETALL */
struct seminfo *_buf; / * forIPC_INFO * /
;

    int main()
key_t key;
int semid;
union semun sem_arg;
int nsems = 1; // Number of semaphores in the set
int sem_flags = IPC_CREAT — 0666; // Create if not exists, with rw-rw-rw- permissions

    // Generate a unique key for the semaphore set
if ((key = ftok(".", 'A')) == -1)
perror("ftok");
return 1;


    // Create a semaphore set with 1 semaphore, or get an existing one
// semget(key, nsems, sem_flags)
// Key: unique identifier
// nsems: number of semaphores in the set (here, 1)
// sem_flags: IPC_CREAT to create, 0666 for permissions
if ((semid = semget(key, nsems, sem_flags)) == -1)
perror("semget");
return 1;


    printf("Semaphore set created with ID:

    // Initialize the semaphore value to 1 (making it a binary semaphore/mutex)
// semctl(semid, sem_num, cmd, arg)
// semid: ID of the semaphore set
// sem_num: index of the semaphore within the set (0 for the first one)
// cmd: SETVAL to set a value
// arg: union semun containing the value
sem_arg.val = 1;
if (semctl(semid, 0, SETVAL, sem_arg) == -1)
perror("semctl SETVAL");
// Clean up the semaphore if initialization fails
semctl(semid, 0, IPC_RMID, 0);
return 1;


    printf("Semaphore 0 initialized to 1.");
printf("Leaving semaphore active. Press Ctrl+C to terminate this program.");
printf("You can now use 'ipcs -s' in another terminal to observe it.");
```

```
    // Keep the program running so the semaphore persists for observation
// In a real application, processes would attach to and use this semaphore.
while(1)
// This loop keeps the process alive, holding the semaphore.
// In a real scenario, this program would manage access using semop().
// For demonstration, we just keep it alive so 'ipcs -s' can see it.
sleep(1);


    // Note: The IPC_RMID call is commented out here as the program is designed
// to be terminated manually (Ctrl+C) to allow observation.
// In a production application, you would typically call IPC_RMID when done.
// if (semctl(semid, 0, IPC_RMID, 0) == -1)
// perror("semctl IPC_RMID");
// return 1;
//
// printf("Semaphore set with ID:

    return 0;
```

8 Conclusion

This lecture has presented an extensive exploration of semaphores, reinforcing understanding of their essential function in inter-process synchronization. The lecture notes started by breaking down the fundamental idea of a semaphore, recognizing its role as a signaling mechanism vital for coordinating concurrent tasks. It then then highlighted the differences between binary semaphores, which are often used as mutexes to ensure mutual exclusion, and counting semaphores, which facilitate access to a limited number of resources. Importantly also, was that it ventured into the practical application of these robust tools within a Linux setting, looking closely at the specific system calls and processes needed to effectively create, use, and manage semaphores in actual scenarios.

In addition, the lecture notes equips the reader with skills to tackle classic synchronization problems directly. By leveraging semaphore principles, it also equips one with skills to successfully navigate the challenges of the Producer-Consumer problem, ensuring smooth data transfer, and addressed the Dining Philosophers problem, showing how to prevent deadlocks effectively. Finally, it examined the inherent challenges that come with semaphore usage, thoroughly investigating potential issues such as deadlocks and starvation. With strategies in hand to detect and avert these challenges, one should be more equipped to create resilient, efficient, and error-free concurrent systems. This foundational knowledge is crucial for any developer looking to excel in the complexities of multi-threaded or multi-process programming.

List of References

1. “Operating System Concepts” by Silberschatz, Galvin, Gagne
2. Silberschatz, A., & Galvin, P. B. (2013). Operatings System Concepts.
3. Stevens, W. R., & Rago, S. A. (2013). Advanced programming in the UNIX environment. Addison-Wesley.