**Synchronization Tools**

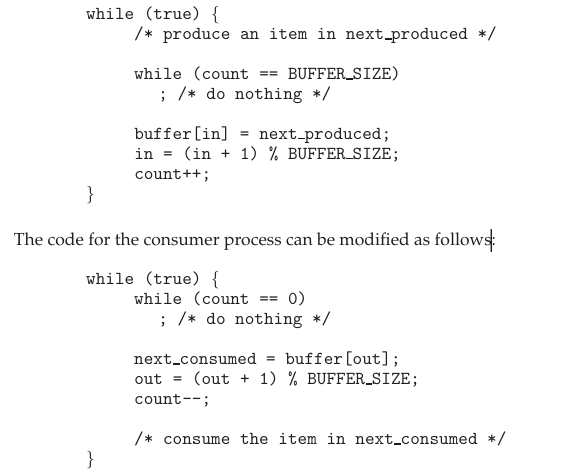
**Background**

Concurrent execution involves the CPU scheduler rapidly switching between processes to provide the illusion of simultaneous execution. Processes may be interrupted at any point, and the CPU may be assigned to execute instructions of another process, leading to partial completion of processes before others are scheduled.

Parallel execution, on the other hand, involves the simultaneous execution of two instruction streams representing different processes on separate processing cores.

However both concurrent and parallel execution can raise challenges related to the integrity of shared data among processes. These issues are significant in scenarios where multiple processes are accessing and modifying shared data concurrently or in parallel.

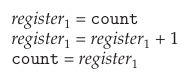
Recall the producer-consumer problem, there was a limitation in the solution where at most BUFFER SIZE - 1 items could be stored in the buffer simultaneously. To address this, the proposed modification involves introducing an integer variable called "*count*" initialized to 0. Count is incremented when adding items to the buffer and decremented when removing items.

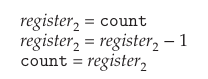


However, it's noted that although these modifications work correctly when executed separately, they may not function properly when executed concurrently.

For instance, if the current value of "count" is 5 and both the producer and consumer processes execute "count++" and "count--" statements concurrently, the final value of "count" may not be predictable. Due to the interleaved execution of these statements, the final value of "count" could be 4, 5, or 6.

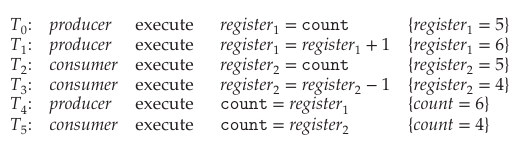
The count++ and count—instructions can be written as following:





Here register1 and register2 are local CPU registers. However, due to the interleaved nature of concurrent execution and the potential for context switching between processes, the final value of "count" can be incorrect. The registers will be saved and restored by interrupt handler.

One such example is:



In the provided example, even though there are five items in the buffer initially, due to the interleaved execution, the final value of "count" is incorrectly determined as 4 or 6 instead of 5, indicating that there are only four items in the buffer.

We arrive at this state due to *race condition* thus leading to unpredictable outcomes based on the order of access. To prevent such issues, synchronization is necessary to ensure that only one process can manipulate the data at a time.

**Race condition** - occurs when multiple processes manipulate the same data concurrently, leading to unpredictable outcomes based on the order of access.

This need for synchronization is particularly vital in operating systems where various components access shared resources and in the context of multicore systems where multithreaded applications are common.

**Critical Section Problem**

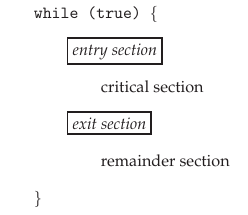
The critical-section problem pertains to systems with multiple processes, each possessing a *critical* section, wherein they access and update shared data. When one process is in its critical section, others must wait, ensuring exclusive access. The problem aims to create a synchronization protocol allowing processes to cooperatively share data. Processes request entry into their critical section via an *entry* section, followed by the critical section itself, possibly with an *exit* section, and finally, the remainder section.

**Critical Section** - Code segment where processes access and update shared data.

**Entry Section** - Code segment where a process requests entry into its critical section.

**Exit Section** - Optional code segment following the critical section for process exit.

**Remainder Section** - Code segment after the critical section, executing after process synchronization.



A solution to the critical section problem must satisfy the following:

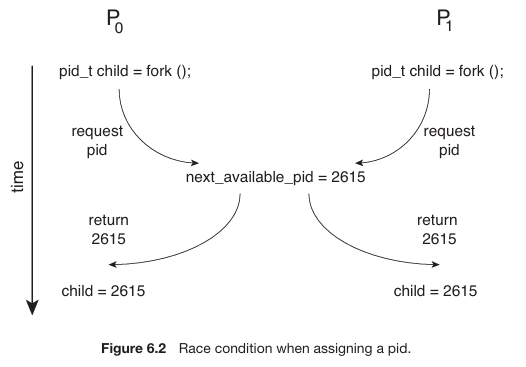
1. **Mutual Exclusion** – When a process is executing its critical section, then no other process is allowed to execute their critical section.
2. **Progress** - If no process is currently in its critical section and some processes want to enter theirs, only those processes not in their remainder sections can decide which enters next. This decision can't be delayed indefinitely.
3. **Bounded waiting** - There's a limit on number of times other processes can enter their critical sections between when one process requests entry and when that request is actually granted.

At any given time, many kernel-mode processes maybe active in operating system, which can make the code being executed/implemented prone to race condition. It can happen like the following:

1. **File Management** - Kernel code managing a list of open files is susceptible to race conditions when multiple processes attempt to open or close files simultaneously. If two processes attempt to modify the file list at the same time, it could lead to inconsistencies or errors.

2. **Process Creation** - When processes create child processes using the fork () system call, there's a race condition on the variable representing the next available process identifier (`pid`). Without proper synchronization, two separate processes might end up being assigned the same process identifier.

Other critical kernel data structures susceptible to race conditions include those related to memory allocation, process lists, and interrupt handling.



On a single core environment, race condition can be subdued via interrupt disabling. This will allow the sequence of instructions to be executed without preemption.

In a multiprocessor environment, disabling interrupts to handle critical sections becomes inefficient due to the time-consuming process of passing messages to all processors, reducing system efficiency, and impacting tasks like clock updates.

Operating systems typically employ two approaches to manage critical sections: preemptive and non-preemptive kernels.

|  |  |
| --- | --- |
| **Preemptive Kernel** | **Non-preemptive Kernel** |
| A preemptive kernel allows processes to be interrupted while running in kernel mode | A non-preemptive kernel does not permit interruption until a process exits kernel mode, blocks, or yields control. |

Non-preemptive kernels inherently avoid race conditions on kernel data structures as only one process operates in the kernel at a time. However, preemptive kernels, though susceptible to race conditions, offer advantages such as increased responsiveness and suitability for real-time programming not allowing a task to monopolize CPU or preempt a lower priority task (batch process) in favor of a higher one (real-time process). They need to be carefully design to mitigate race conditions, particularly challenging in multiprocessor environments where simultaneous execution on different CPU cores is possible (SMP).

**Peterson’s Solution**

A software based solution where two processes namely P0 and P1 alternate execution between their critical and remainder sections.

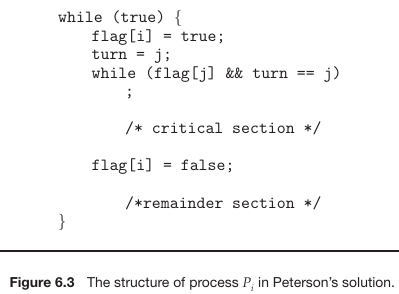
Modern computer architectures may not ensure the correctness of Peterson's solution due to the way they execute basic machine-language instructions like load and store.

Pi = P0, Pj = P1 -> **for the remainder of this section**

Peterson’s solution requires two shared data items:



The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process Pi is allowed to execute in its critical section. The flag array is used to indicate if a process is ready to enter its critical section. For example, if flag[i] is true, Pi is ready to enter its critical section.



The breakdown of the above code is as follows:

* Process `Pi` sets its flag to true to indicate its desire to enter the critical section.
* `Pi` then sets the `turn` variable to the identifier of the other process `Pj`, allowing `Pj` to enter if it wishes.
* If both `Pi` and `Pj` attempt to enter simultaneously, they may both set `turn` to their respective identifiers (`i` and `j`) at nearly the same time.
* However, due to the nature of shared variables, only one of these assignments will succeed, while the other will be overwritten immediately.
* The eventual value of `turn` determines which process (`Pi` or `Pj`) is granted entry into the critical section.
* The process whose identifier remains in turn gains priority and can enter the critical section first, ensuring mutual exclusion.

To prove mutual exclusion (Property 1), we observe the behavior of processes Pi and Pj in their critical sections:

* Each process Pi enters its critical section only if either `flag[j]` is `false` or `turn` equals `i`.
* If both processes could be executing in their critical sections simultaneously, then `flag[0]` and `flag[1]` would both be `true`.
* This implies that both processes could not have executed their `while` statements at the same time because `turn` can only be either `0` or `1`, but not both.
* Consequently, one of the processes, say Pj, must have successfully executed the `while` statement, while Pi had to execute at least one additional statement (`turn == j`).
* At this point, `flag[j]` is `true` and `turn` equals `j`, indicating that Pj is in its critical section.
* Mutual exclusion is preserved since `flag[j]` remains `true` and `turn` remains `j` as long as Pj is in its critical section, ensuring that only one process can execute in the critical section at a time.

To prove progress (Property 2) and bounded waiting (Property 3), we analyze the behavior of processes Pi and Pj:

* Process Pi can only be prevented from entering the critical section if it is stuck in the `while` loop with the condition `flag[j] == true` and `turn == j`.
* If Pj is not ready to enter the critical section, `flag[j]` will be `false`, allowing Pi to enter its critical section.
* If Pj has set `flag[j] ` to `true` and is also executing in its `while` statement, then either `turn == i` or `turn == j`.
* If `turn == i`, Pi will enter the critical section.
* If `turn == j`, Pj will enter the critical section. However, once Pj exits its critical section, it will reset `flag[j] ` to `false`, allowing Pi to enter its critical section.
* If Pj resets `flag[j] ` to `true`, it must also set `turn` to `i`. Thus, since Pi does not change the value of the variable `turn` while executing the `while` statement, Pi will enter the critical section (progress) after at most one entry by Pj (bounded waiting).

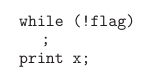
As mentioned before, Peterson’s solution will not work on modern computer architecture because compilers reorder read and write instructions without dependencies in order to improve system efficiency.

In a single-threaded application, reordering instructions doesn't affect program correctness, as long as the final outcome remains consistent with expectations.

However, in a multithreaded application with shared data, instruction reordering can lead to inconsistent or unexpected results. This is because multiple threads accessing shared data may depend on the sequence of instructions, and any deviation from expected order could cause unexpected behavior.

An example where the shared data items are:  

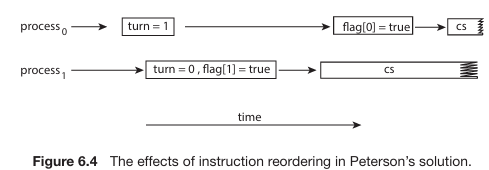

Suppose Thread 1 performs:



Whereas Thread 2 performs:



The reordering by the processor may print 0. If Thread 2’s instructions were reordered where flag was set true first then x would be printed 0. Similarly Thread 1’s instructions may be reordered rendering x as 0 before even loading flag.



By rearranging the order of the first two instructions inside the while loop in Figure 6.3; both processes could potentially enter their respective critical sections simultaneously.

**Hardware Support for Synchronization**

Previously described solution was *software* based solution as it required help from OS and hardware level instructions for mutual exclusion. Now, hardware based solutions will be discussed.

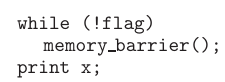
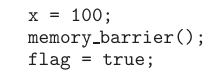
**Memory barriers**

Memory models in computer architecture determine how memory modifications are visible to other processors, categorized into *strongly* ordered and *weakly* ordered models.

**Memory model** - A memory model in computer architecture determines how memory modifications made by one processor are visible to other processors in a multi-processing environment. Determines what guarantees will be provided to the application program.

|  |  |
| --- | --- |
| **Strongly ordered** | **Weakly ordered** |
| In the context of memory models, strongly ordered systems ensure that memory modifications made by one processor are immediately visible to all other processors in a multi-processing environment. | In the context of memory models, weakly ordered systems allow for the possibility that memory modifications made by one processor may not be immediately visible to other processors in a multi-processing environment. |

To ensure visibility of memory modifications in shared-memory multiprocessors, memory barrier instructions are provided. Memory barriers enforce completion of all loads and stores before subsequent operations, preventing instruction reordering issues.

The first example ensures that flag is loaded before x’s value. The second one ensures that x’s value is assigned before setting flag to true.

In scenarios like Peterson's solution for mutual exclusion, memory barriers can be strategically placed to ensure correct execution. Memory barriers are low-level operations primarily utilized by kernel developers for specialized code.

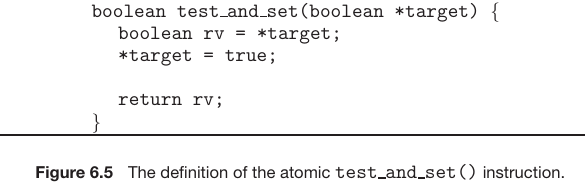
**Hardware Instructions**

This section discusses hardware instructions for atomic operations, specifically focusing on the test and set () and compare and swap () instructions. These instructions provide atomicity, ensuring that operations occur as one uninterruptible unit.

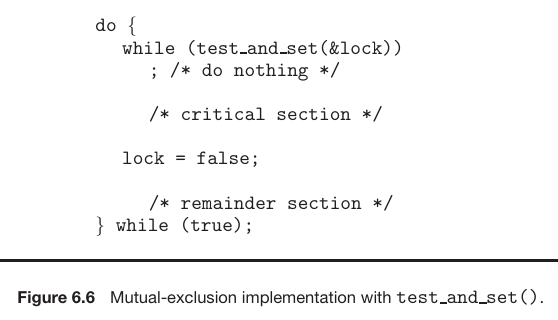
**Atomically** – Operations are performed as a single uninterruptable unit.

**Atomic Operations** - Operations in computer science that are uninterruptible. Atomic operations are guaranteed to be executed as a single unit without interruption, ensuring that they are either completed entirely or not at all.

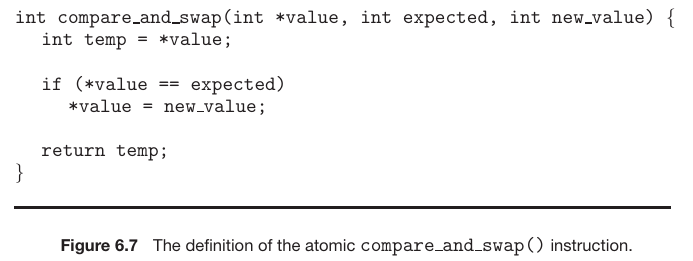
They are utilized to implement mutual exclusion, crucial for critical sections in multi-threaded systems. The test and set () instruction sets a Boolean variable to true atomically and is used to implement mutual exclusion by continuously attempting to acquire a lock.



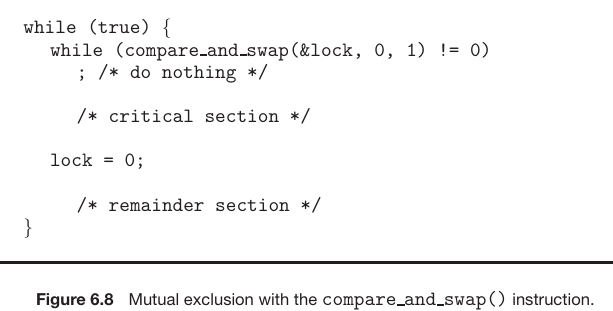
The test\_and\_set () function sets a Boolean variable (\*target) to true atomically and returns its previous value. In the provided example, it is used to implement mutual exclusion by continuously attempting to acquire a lock (lock) until it is successfully acquired.



Similarly, the compare and swap () instruction swaps the content of two words atomically and is utilized to implement mutual exclusion by setting a global variable to indicate whether a process has acquired the lock.

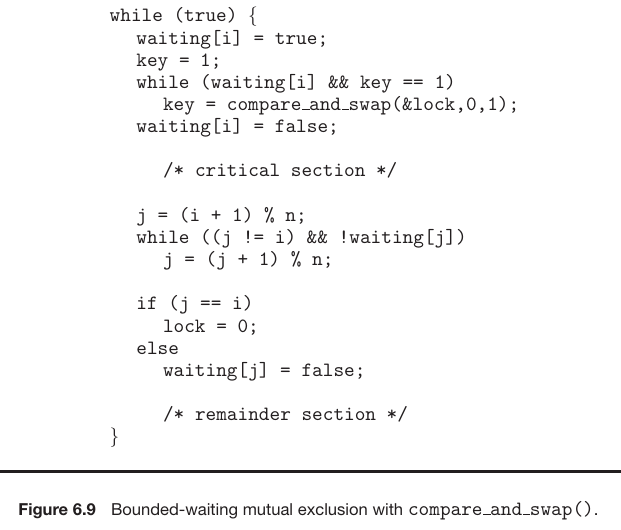


The compare\_and\_swap () function swaps the content of two words atomically based on a condition. It takes three arguments: the address of the variable to modify (\*value), the expected value (expected), and the new value (new\_value). In the example, it is utilized to implement mutual exclusion by attempting to set a global variable (lock) to 1 only if its current value is 0. If successful, the process enters its critical section, and upon exiting, it resets the lock to 0, allowing other processes to enter their critical sections.



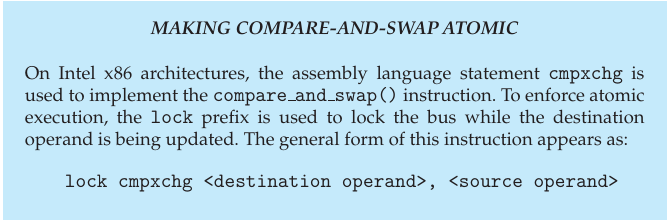
While the above implementation addresses mutual exclusion, it doesn’t address bounded waiting requirement.

 -> shared data items



The provided algorithm utilizes the compare and swap () instruction to ensure mutual exclusion, progress, and bounded waiting in a concurrent system. It involves maintaining an array of Boolean values (`waiting`) and an integer variable (`lock`). Each process, denoted as Pi, follows a series of steps including setting its waiting flag, attempting to acquire the lock using compare and swap (), and entering its critical section based on certain conditions.

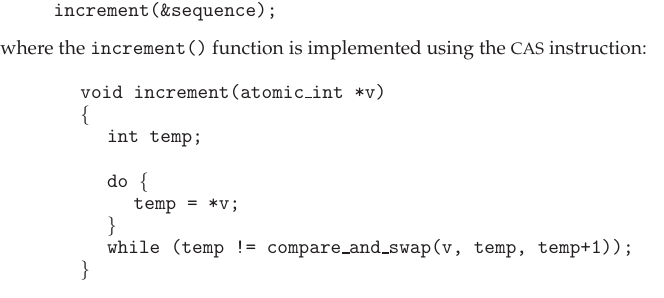
The algorithm ensures mutual exclusion by allowing only one process to acquire the lock at a time, progress by enabling waiting processes to proceed when the current process exits the critical section, and bounded waiting by providing a fair ordering for processes to enter the critical section within a finite number of turns. The algorithm relies on atomic operations to maintain synchronization and is designed to meet the critical-section requirements of mutual exclusion, progress, and bounded waiting in concurrent systems.



**Atomic Variables**

Atomic variables provide atomic operations on basic data types like integers and Booleans, helping ensure mutual exclusion in situations prone to data races, such as when incrementing a counter. These atomic variables are often implemented using compare and swap () operations.

**Atomic Variables** – Variables that provide atomic operations, ensuring that updates to the variable occur as a single uninterruptible unit, typically implemented using compare and swap () operations.



While atomic variables address race conditions in some scenarios, they may not entirely solve race conditions in all situations, as demonstrated with the bounded-buffer problem. Two consumers who are looping waiting for count > 0 (count is atomic) and the buffer is empty. When an item is produced, they both will try to consume it simultaneously.

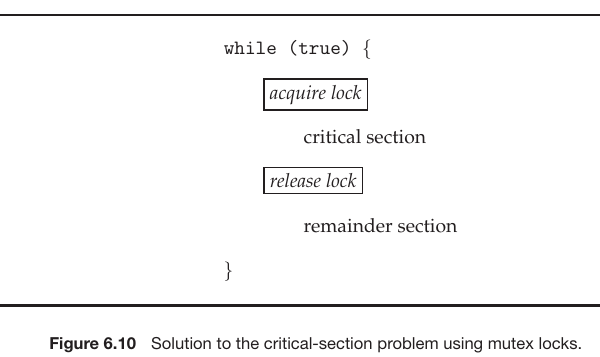
However, atomic variables are widely used in operating systems and concurrent applications for single updates of shared data.

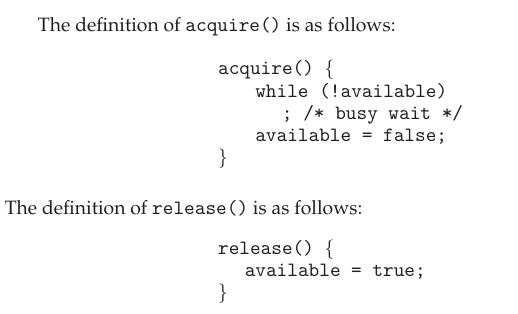
**Mutex Locks**

Mutex lock is a higher-level software tool used by operating-system designers to solve the critical-section problem, preventing race conditions in concurrent programming. It is accessed through functions such as acquire () and release (), where acquire () locks the critical section and release () unlocks it.

The mutex lock utilizes a Boolean variable called available, which indicates whether the lock is available. If available, acquire () succeeds in locking the critical section; otherwise, the process attempting to acquire the lock is blocked until it's released.

**Mutex lock** – A higher-level software tool used for mutual exclusion in concurrent programming to prevent race conditions.





These instructions must be performed *atomically,* thus can be implemented using compare and swap (CAS).

**Lock contention**

**Locks** – Mechanisms used in concurrent programming to control access to shared resources, ensuring mutual exclusion among threads.

Locks in concurrent programming are categorized as either contended or uncontended.

|  |  |
| --- | --- |
| **Contended** | **Uncontended** |
| A lock is considered contended if a thread is blocked while attempting to acquire it. | A lock is considered uncontended if the lock is available for immediate acquisition. |

**High Contention** – Type of contended lock, occurs when there is a significant number of threads contending for the same lock simultaneously, leading to potential performance degradation.

**Low Contention** – Type of contented lock where only a few threads are contending for the lock, resulting in less competition and potential performance improvements.

The described implementation of mutex locks (fig 6.10) involves *busy waiting*, where a process continuously loops in acquire () call while another process holds the lock. This approach is inefficient in multiprogramming systems as it wastes CPU cycles and prevents other processes from utilizing the CPU.

**Busy waiting** – A synchronization technique where a process continuously polls or checks for a condition until it becomes true is busy waiting.

The described mutex lock implementation is also referred to as a spin lock because the process "spins" while waiting for the lock. Spinlocks avoid context switches, which can be time-consuming, making them preferable in certain circumstances, especially on multi-core systems where one thread spins for a lock while the other executes its critical section on another core.

**Spinlock** – A type of mutex lock implementation where a process repeatedly checks for lock availability without blocking, also known as busy-waiting.

**Short Duration**

Spinlocks are frequently preferred on multi-processor systems when the lock is anticipated to be held for a brief period. The decision to use a spinlock is based on the understanding that waiting for a lock entails *two* context switches: one to transition the thread to a waiting state and another to resume the waiting thread once the lock is accessible. Consequently, spinlocks are advisable for locks expected to be held for a shorter duration than two context switches.

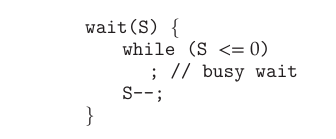
**Semaphores**

Semaphores provide a more sophisticated way for processes to synchronize their activities compared to mutex locks.

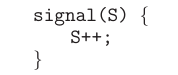
**Semaphore –** An integer variable used for synchronization that supports *atomic* operations for process coordination and synchronization.

Semaphores are synchronization tools introduced by Dutch computer scientist Edsger Dijkstra. A semaphore, denoted as S, is an integer variable that supports two atomic operations: wait () and signal ().

The wait () operation, originally termed P, checks if the value of the semaphore is greater than zero. If not, it waits until it becomes greater than zero. Once the semaphore's value is positive, it decrements the value of S. Both operations are *atomic*.



The signal () operation, originally called V, increments the value of the semaphore atomically.

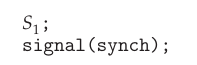


**Semaphore Usage**

The OS utilizes two main types of semaphores; *counting* and *binary*

|  |  |
| --- | --- |
| **Counting** | **Binary** |
| Counting semaphores allow the value to range over an unrestricted domain and are used to control access to a finite number of resources. Each process wishing to use a resource decrements the semaphore count through a wait () operation and increments it through a signal () operation upon releasing the resource. When the semaphore count reaches 0, all resources are in use, and processes wishing to use them will block until the count becomes greater than 0 | Binary semaphores, on the other hand, can only have values of 0 or 1 and behave similarly to mutex locks, providing mutual exclusion. They are often used in systems where mutex locks are not available. |

Semaphores can also be used to solve various synchronization problems. For example, consider two concurrently running processes, P1 with statement S1 and P2 with statement S2. If we require that S2 be executed only after S1 has completed, we can implement this using a common semaphore named synch, initialized to 0. In process P1, after statement S1, we insert the statement:



In process P2, we have:



Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal (synch), which is after statement S1 has been executed.

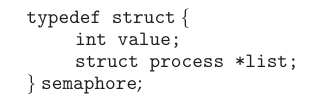
**Semaphore Implementation**

All synchronization methods described so far suffer from busy waiting. To overcome this we can modify the *wait* and *signal* functions as follows:

When a process executes the wait () operation on a semaphore and finds that the semaphore value is not positive, it must wait. Instead of engaging in busy waiting, where the process continuously checks the semaphore value, the process can suspend itself. The suspend operation involves placing the process into a waiting queue associated with the semaphore, and the process's state is switched to the waiting state. Control is then transferred to the CPU scheduler, which selects another process to execute.

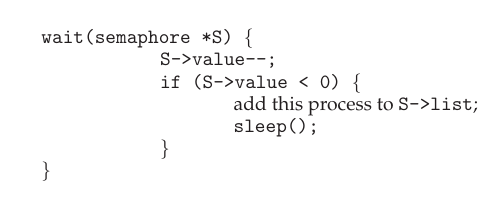
When a process is suspended and waiting on a semaphore S, it can be restarted when another process executes a signal () operation on that semaphore. The process is restarted by a wakeup () operation, which transitions the process from the waiting state to the ready state. Upon waking up, the process is placed in the ready queue, ready for execution. Whether the CPU is switched from the currently running process to the newly ready process depends on the CPU-scheduling algorithm in use.

A semaphore is defined as a structure containing an integer value and a list of processes waiting on it.

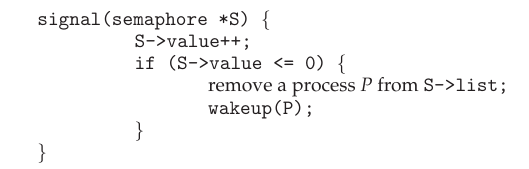


When a process must wait on a semaphore, it is added to the list of waiting processes.

The wait () operation decrements the semaphore's value and, if it becomes negative, adds the current process to the list and suspends it using the sleep () operation.



The signal () operation increments the semaphore's value and, if it was negative before incrementing, removes a process from the waiting list and resumes its execution using the wakeup () operation.



The sleep () and wakeup () operations are provided by the operating system as basic system calls, allowing processes to suspend and resume their execution, respectively.

In this implementation, semaphore values can be negative, unlike the classical definition where they are always non-negative. If a semaphore's value is negative, it indicates the number of processes waiting on it. This fact results from switching the order of the decrement and the test in the implementation of the wait () operation.

The list of waiting processes can be managed by adding a link field to each process control block (PCB). Each semaphore contains an integer value and a pointer to a list of PCBs, allowing for easy management of waiting processes.

To ensure bounded waiting, processes can be added and removed from the list using a FIFO queue structure, where the semaphore contains both head and tail pointers to the queue. However, any queuing strategy can be employed as long as it ensures correctness.

Semaphore operations must be executed atomically to prevent race conditions. In a single-processor environment, this can be achieved by inhibiting interrupts during the execution of wait () and signal () operations.

In a multicore environment, interrupts must be disabled on every processing core to prevent interleaving of instructions from different processes. This can be challenging and may impact performance, necessitating alternative techniques such as compare-and-swap or spinlocks allowing wait and signal to execute atomically.

Despite efforts to reduce busy waiting, it still exists in this implementation, albeit limited to the critical sections of wait () and signal () operations. These critical sections are typically short, minimizing the duration of busy waiting. However, in applications with long or frequently occupied critical sections, busy waiting remains inefficient.

**Liveness**

**Liveness** – It refers to a set of properties that a system must satisfy to ensure that processes make progress during their execution life cycle.

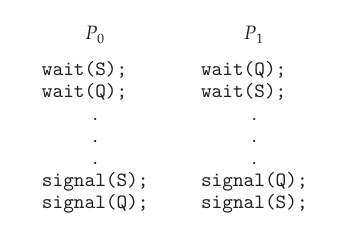
This section discusses the consequences of using synchronization tools to manage critical sections in concurrent programming. It highlights the risk of processes waiting indefinitely to enter critical sections, which violates the progress and bounded-waiting criteria outlined for solutions to the critical-section problem.

This indefinite waiting result in a "*liveness* failure," where processes fail to make progress during their execution lifecycle. Such failures can manifest in various forms, all of which adversely affect system performance and responsiveness. One example provided is an infinite loop, where a process may wait indefinitely, potentially causing a *deadlock*.

**Deadlock**

A deadlock is a condition in concurrent systems where two or more processes are unable to proceed because each is waiting for the other to release a resource, resulting in a standstill and halting progress indefinitely.

Using a semaphore with a waiting queue can lead to deadlock, where multiple processes are stuck indefinitely waiting for an event triggered by only one of them, typically the execution of a signal() operation. This deadlock state halts progress in the system, rendering the processes unable to proceed.



Suppose that P0 executes wait (S) and then P1 executes wait (Q) .When P0 executes wait (Q), it must wait until P1 executes signal (Q). Similarly, when P1 executes wait (S), it must wait until P0 executes signal (S). Since these signal () operations cannot be executed, P0 and P1 are deadlocked.

We say that a set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set. The “events” with which we are mainly concerned here are the acquisition and release of resources such as mutex locks and semaphores.

**Priority Inversion**

Priority inversion refers to a situation in concurrent systems where a lower-priority process holds a resource needed by a higher-priority process, causing the higher-priority process to wait.

A scheduling challenge arises when a higher-priority process needs to access kernel data currently being used by a lower-priority process, potentially causing the higher-priority process to wait. This situation is further complicated if the lower-priority process is preempted by another higher-priority process. This scenario, known as priority inversion, occurs in systems with multiple priorities.

Priority inversion is typically addressed using a priority-inheritance protocol, where processes accessing resources needed by higher-priority processes temporarily inherit the higher priority until they are finished with the resources. This protocol ensures that critical resources are promptly released to higher-priority processes, preventing delays caused by lower-priority processes.