Chapitre VI SEMAPHORES

In computer science, a **semaphore** is a variable or abstract data type used to control access to a common resource by multiple processes in a concurrent system such as a multitasking operating system. A semaphore is simply a variable. This variable is used to solve critical section problems and to achieve process synchronization in the multi processing environment. A trivial semaphore is a plain variable that is changed (for example, incremented or decremented, or toggled) depending on programmer-defined conditions.

A useful way to think of a semaphore as used in the real-world system is as a record of how many units of a particular resource are available, coupled with operations to adjust that record *safely* (i.e., to avoid race conditions) as units are acquired or become free, and, if necessary, wait until a unit of the resource becomes available.

Semaphores are a useful tool in the prevention of race conditions; however, their use is by no means a guarantee that a program is free from these problems. Semaphores which allow an arbitrary resource count are called **counting semaphores**, while semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called **binary semaphores** and are used to implement locks.

Library analogy

Suppose a library has 10 identical study rooms, to be used by one student at a time. Students must request a room from the front desk if they wish to use a study room. If no rooms are free, students wait at the desk until someone relinquishes a room. When a student has finished using a room, the student must return to the desk and indicate that one room has become free.

In the simplest implementation, the clerk at the front desk knows only the number of free rooms available, which they only know correctly if all of the students actually use their room while they've signed up for them and return them when they're done. When a student requests a room, the clerk decreases this number. When a student releases a room, the clerk increases this number. The room can be used for as long as desired, and so it is not possible to book rooms ahead of time.

In this scenario the front desk count-holder represents a counting semaphore, the rooms are the resource, and the students represent processes/threads. The value of the semaphore in this scenario is initially 10, with all rooms empty. When a student requests a room, they are granted access, and the value of the semaphore is changed to 9. After the next student comes, it drops to 8, then 7 and so on. If someone requests a room and the current value of the semaphore is 0,[2] they are forced to wait until a room is freed (when the count is increased from 0). If one of the rooms was released, but there are several students waiting, then any method can be used to select the one who will occupy the room (like FIFO or flipping a coin). And of course, a student needs to inform the clerk about releasing their room only after really leaving it, otherwise, there can be an awkward situation when such student is in the process of leaving the room (they are packing their textbooks, etc.) and another student enters the room before they leave it.

Important observations

When used to control access to a pool of resources, a semaphore tracks only *how many* resources are free; it does not keep track of *which* of the resources are free. Some other mechanism (possibly involving more semaphores) may be required to select a particular free resource.

The paradigm is especially powerful because the semaphore count may serve as a useful trigger for a number of different actions. The librarian above may turn the lights off in the study hall when there are no students remaining, or may place a sign that says the rooms are very busy when most of the rooms are occupied.

The success of the protocol requires applications to follow it correctly. Fairness and safety are likely to be compromised (which practically means a program may behave slowly, act erratically, hang or crash) if even a single process acts incorrectly. This includes:

- requesting a resource and forgetting to release it;
- releasing a resource that was never requested;
- holding a resource for a long time without needing it;
- using a resource without requesting it first (or after releasing it).

Even if all processes follow these rules, *multi-resource deadlock* may still occur when there are different resources managed by different semaphores and when processes need to use more than one resource at a time, as illustrated by the dining philosophers problem.

Semantics and implementation[edit]

Counting semaphores are equipped with two operations, historically denoted as P and V (see § Operation names for alternative names). Operation V increments the semaphore *S*, and operation P decrements it.

The value of the semaphore *S* is the number of units of the resource that are currently available. The P operation wastes time or sleeps until a resource protected by the semaphore becomes available, at which time the resource is immediately claimed. The V operation is the inverse: it makes a resource available again after the process has finished using it. One important property of semaphore *S* is that its value cannot be changed except by using the V and P operations.

A simple way to understand wait (P) and signal (V) operations is:

- wait: Decrements the value of semaphore variable by 1. If the new value of the semaphore variable is negative, the process executing wait is blocked (i.e., added to the semaphore's queue). Otherwise, the process continues execution, having used a unit of the resource.

- signal: Increments the value of semaphore variable by 1. After the increment, if the pre-increment value was negative (meaning there are processes waiting for a resource), it transfers a blocked process from the semaphore's waiting queue to the ready queue.

Many operating systems provide efficient semaphore primitives that unblock a waiting process when the semaphore is incremented. This means that processes do not waste time checking the semaphore value unnecessarily.

The counting semaphore concept can be extended with the ability to claim or return more than one "unit" from the semaphore, a technique implemented in Unix. The modified V and P operations are as follows, using square brackets to indicate atomic operations, i.e., operations which appear indivisible from the perspective of other processes:

```
function V(semaphore S, integer I):
    [S ← S + I]

function P(semaphore S, integer I):
    repeat:
        [if S ≥ I:
        S ← S − I
        break]
```

However, the remainder of this section refers to semaphores with unary V and P operations, unless otherwise specified.

To avoid starvation, a semaphore has an associated queue of processes (usually with FIFO semantics). If a process performs a P operation on a semaphore that has the value zero, the process is added to the semaphore's queue and its execution is suspended. When another process increments the semaphore by performing a V operation, and there are processes on the queue, one of them is removed from the queue and resumes execution. When processes have different priorities the queue may be ordered by priority, so that the highest priority process is taken from the queue first.

If the implementation does not ensure atomicity of the increment, decrement and comparison operations, then there is a risk of increments or decrements being forgotten, or of the semaphore value becoming negative. Atomicity may be achieved by using a machine instruction that is able to read, modify and write the semaphore in a single operation. In the absence of such a hardware instruction, an atomic operation may be synthesized through the use of a software mutual exclusion algorithm. On uniprocessor systems, atomic operations can be ensured by temporarily suspending preemption or disabling hardware interrupts. This approach does not work on multiprocessor systems where it is possible for two programs sharing a semaphore to run on different processors at the same time. To solve this problem in a multiprocessor system a locking variable can be used to control access to the semaphore. The locking variable is manipulated using a test-and-set-lock command.

Example

Trivial example

Consider a variable *A* and a boolean variable *S*. *A* is only accessed when *S* is marked true. Thus, *S* is a semaphore for *A*.

One can imagine a stoplight signal (*S*) just before a train station (*A*). In this case, if the signal is green, then one can enter the train station. If it is yellow or red (or any other color), the train station cannot be accessed.

Login queue

Consider a system that can only support ten users (S=10). Whenever a user logs in, P is called, decrementing the semaphore *S* by 1. Whenever a user logs out, V is called, incrementing *S* by 1 representing a login slot that has become available. When *S* is 0, any users wishing to log in must wait until $S > 0$ and the login request is enqueued onto a FIFO queue; mutual exclusion is used to ensure that requests are enqueued in order. Whenever *S* becomes greater than 0 (login slots available), a login request is dequeued, and the user owning the request is allowed to log in.

Producer–consumer problem

In the producer–consumer problem, one process (the producer) generates data items and another process (the consumer) receives and uses them. They communicate using a queue of maximum size *N* and are subject to the following conditions:

- the consumer must wait for the producer to produce something if the queue is empty;
- the producer must wait for the consumer to consume something if the queue is full.

The semaphore solution to the producer–consumer problem tracks the state of the queue with two semaphores: `emptyCount`, the number of empty places in the queue, and `fullCount`, the number of elements in the queue. To maintain integrity, `emptyCount` may be lower (but never higher) than the actual number of empty places in the queue, and `fullCount` may be lower (but never higher) than the actual number of items in the queue. Empty places and items represent two kinds of resources, empty boxes and full boxes, and the semaphores `emptyCount` and `fullCount` maintain control over these resources.

The binary semaphore `useQueue` ensures that the integrity of the state of the queue itself is not compromised, for example by two producers attempting to add items to an empty queue simultaneously, thereby corrupting its internal state. Alternatively a mutex could be used in place of the binary semaphore.

The `emptyCount` is initially *N*, `fullCount` is initially 0, and `useQueue` is initially 1.

The producer does the following repeatedly:

```
produce:
    P(emptyCount)
    P(useQueue)
    putItemIntoQueue(item)
    V(useQueue)
    V(fullCount)
```

The consumer does the following repeatedly

```
consume:
    P(fullCount)
    P(useQueue)
    item ← getItemFromQueue()
    V(useQueue)
    V(emptyCount)
```

Below is a substantive example:

1. A single consumer enters its critical section. Since `fullCount` is 0, the consumer blocks.
2. Several producers enter the producer critical section. No more than *N* producers may enter their critical section due to `emptyCount` constraining their entry.
3. The producers, one at a time, gain access to the queue through `useQueue` and deposit items in the queue.
4. Once the first producer exits its critical section, `fullCount` is incremented, allowing one consumer to enter its critical section.

Note that `emptyCount` may be much lower than the actual number of empty places in the queue, for example in the case where many producers have decremented it but are waiting their turn on `useQueue` before filling empty places. Note that `emptyCount + fullCount ≤ N` always holds, with equality if and only if no producers or consumers are executing their critical sections

In computing, the **producer–consumer problem**[1][2] (also known as the **bounded-buffer problem**) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

The solution for the producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer. The solution can be reached by means of inter-process communication, typically using semaphores. An inadequate solution could result in a deadlock where both processes are waiting to be awakened. The problem can also be generalized to have multiple producers and consumers.

Inadequate implementation

To solve the problem, some programmer might come up with a solution shown below. In the solution two library routines are used, `sleep` and `wakeup`. When sleep is called, the caller is blocked until another process wakes it up by using the wakeup routine. The global variable `itemCount` holds the number of items in the buffer.

```
int itemCount = 0;

procedure producer()
{
    while (true)
    {
        item = produceItem();

        if (itemCount == BUFFER_SIZE)
        {
            sleep();
        }

        putItemIntoBuffer(item);
        itemCount = itemCount + 1;

        if (itemCount == 1)
        {
            wakeup(consumer);
        }
    }
}
```

```
procedure consumer()
{
    while (true)
    {

        if (itemCount == 0)
        {
            sleep();
        }

        item = removeItemFromBuffer();
        itemCount = itemCount - 1;

        if (itemCount == BUFFER_SIZE - 1)
        {
            wakeup(producer);
        }

        consumeItem(item);
    }
}
```

The problem with this solution is that it contains a race condition that can lead to a deadlock. Consider the following scenario:

1. The `consumer` has just read the variable `itemCount`, noticed it's zero and is just about to move inside the `if` block.
2. Just before calling sleep, the consumer is interrupted and the producer is resumed.
3. The producer creates an item, puts it into the buffer, and increases `itemCount`.
4. Because the buffer was empty prior to the last addition, the producer tries to wake up the consumer.
5. Unfortunately, the consumer wasn't yet sleeping, and the wakeup call is lost. When the consumer resumes, it goes to sleep and will never be awakened again. This is because the consumer is only awakened by the producer when `itemCount` is equal to 1.
6. The producer will loop until the buffer is full, after which it will also go to sleep.

Since both processes will sleep forever, we have run into a deadlock. This solution therefore is unsatisfactory.

An alternative analysis is that if the programming language does not define the semantics of concurrent accesses to shared variables (in this case `itemCount`) with use of synchronization, then the solution is unsatisfactory for that reason, without needing to explicitly demonstrate a race condition.

Using semaphores

Semaphores solve the problem of lost wakeup calls. In the solution below we use two semaphores, `fillCount` and `emptyCount`, to solve the problem. `fillCount` is the number of items already in the buffer and available to be read, while `emptyCount` is the number of available spaces in the buffer where items could be written. `fillCount` is incremented and `emptyCount` decremented when a new item is put into the buffer. If the producer tries to decrement `emptyCount` when its value is zero, the producer is put to sleep. The next time an item is consumed, `emptyCount` is incremented and the producer wakes up. The consumer works analogously.

```
semaphore fillCount = 0; // items produced
semaphore emptyCount = BUFFER_SIZE; // remaining space

procedure producer()
{
    while (true)
    {
        item = produceItem();
        down(emptyCount);
        putItemIntoBuffer(item);
        up(fillCount);
    }
}

procedure consumer()
{
    while (true)
    {
        down(fillCount);
        item = removeItemFromBuffer();
        up(emptyCount);
        consumeItem(item);
    }
}
```

The solution above works fine when there is only one producer and consumer. With multiple producers sharing the same memory space for the item buffer, or multiple consumers sharing the same memory space, this solution contains a serious race condition that could result in two or more processes reading or writing into the same slot at the same time. To understand how this is possible, imagine how the procedure `putItemIntoBuffer()` can be implemented. It could contain two actions, one determining the next available slot and the other writing into it. If the procedure can be executed concurrently by multiple producers, then the following scenario is possible:

1. Two producers decrement `emptyCount`
2. One of the producers determines the next empty slot in the buffer
3. Second producer determines the next empty slot and gets the same result as the first producer
4. Both producers write into the same slot

To overcome this problem, we need a way to make sure that only one producer is executing `putItemIntoBuffer()` at a time. In other words, we need a way to execute a critical section with mutual exclusion. The solution for multiple producers and consumers is shown below.

```
mutex buffer_mutex; // similar to "semaphore buffer_mutex = 1", but different (see notes below)
semaphore fillCount = 0;
semaphore emptyCount = BUFFER_SIZE;

procedure producer()
{
    while (true)
    {
        item = produceItem();
        down(emptyCount);
        down(buffer_mutex);
        putItemIntoBuffer(item);
        up(buffer_mutex);
        up(fillCount);
    }
}

procedure consumer()
```

```
{
    while (true)
    {
        down(fillCount);
        down(buffer_mutex);
        item = removeItemFromBuffer();
        up(buffer_mutex);
        up(emptyCount);
        consumeItem(item);
    }
}
```

Notice that the order in which different semaphores are incremented or decremented is essential: changing the order might result in a deadlock. It is important to note here that though mutex seems to work as a semaphore with value of 1 (binary semaphore), but there is difference in the fact that mutex has ownership concept. Ownership means that mutex can only be "incremented" back (set to 1) by the same process that "decremented" it (set to 0), and all other tasks wait until mutex is available for decrement (effectively meaning that resource is available), which ensures mutual exclusivity and avoids deadlock. Thus using mutexes improperly can stall many processes when exclusive access is not required, but mutex is used instead of semaphore.

Critical section

In concurrent programming, concurrent accesses to shared resources can lead to unexpected or erroneous behavior, so parts of the program where the shared resource is accessed need to be protected in ways that avoid the concurrent access. This protected section is the **critical section** or **critical region.** It cannot be executed by more than one process at a time. Typically, the critical section accesses a shared resource, such as a data structure, a peripheral device, or a network connection, that would not operate correctly in the context of multiple concurrent accesses.[1]

Need for critical sections

Different codes or processes may consist of the same variable or other resources that need to be read or written but whose results depend on the order in which the actions occur. For example, if a variable *x* is to be read by process A, and process B has to write to the same variable *x* at the same time, process A might get either the old or new value of *x*.

Process A:

```
// Process A
 .
 .
 b = x + 5;               // instruction executes at time = Tx
 .
```

Process B:

```
// Process B
.
.
x = 3 + z;               // instruction executes at time = Tx
.
```

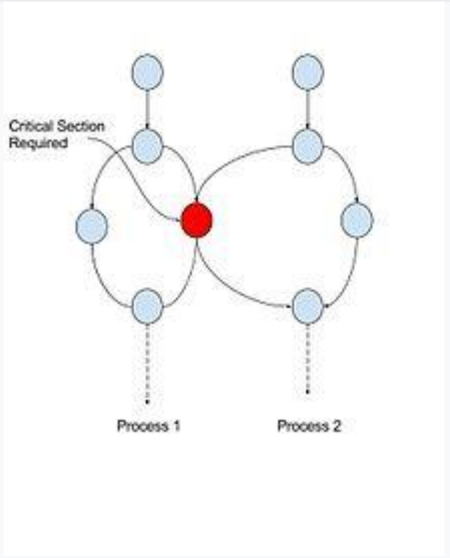


Fig 1: Flow graph depicting need for critical section

In cases like these, a critical section is important. In the above case, if A needs to read the updated value of *x*, executing Process A and Process B at the same time may not give required results. To prevent this, variable *x* is protected by a critical section. First, B gets the access to the section. Once B finishes writing the value, A gets the access to the critical section and variable *x* can be read.

By carefully controlling which variables are modified inside and outside the critical section, concurrent access to the shared variable are prevented. A critical section is typically used when a multi-threaded program must update multiple related variables without a separate thread making conflicting changes to that data. In a related situation, a critical section may be used to ensure that a shared resource, for example, a printer, can only be accessed by one process at a time.

Implementation of critical sections

The implementation of critical sections vary among different operating systems.

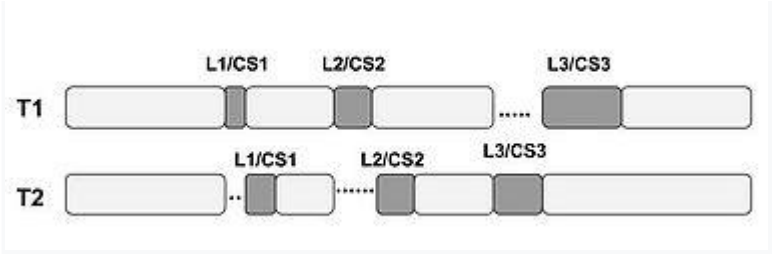
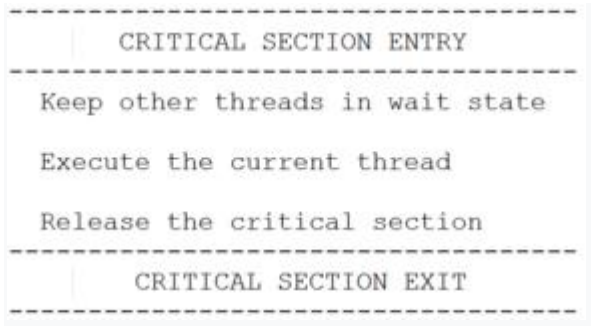


Fig 2: Locks and critical sections in multiple threads

A critical section will usually terminate in finite time,[2] and a thread, task, or process will have to wait for a fixed time to enter it (bounded waiting). To ensure exclusive use of critical sections some synchronization mechanism is required at the entry and exit of the program.

Critical section is a piece of a program that requires mutual exclusion of access.

As shown in Fig 2 in the case of mutual exclusion (Mutex), one thread blocks a critical section by using locking techniques when it needs to access the shared resource and other threads have to wait to get their turn to enter into the section. This prevents conflicts when two or more threads share the same memory space and want to access a common resource.



```
-----------------------------------------
       CRITICAL SECTION ENTRY
-----------------------------------------
  Keep other threads in wait state

  Execute the current thread

  Release the critical section
-----------------------------------------
        CRITICAL SECTION EXIT
-----------------------------------------
```

Fig 3: Pseudo code for implementing critical section

The simplest method to prevent any change of processor control inside the critical section is implementing a semaphore. In uni processor systems, this can be done by disabling interrupts on entry into the critical section, avoiding system calls that can cause a context switch while inside the section, and restoring interrupts to their previous state on exit. Any thread of execution entering any critical section anywhere in the system will, with this implementation, prevent any other thread, including an interrupt, from being granted processing time on the CPU—and therefore from entering any other critical section or, indeed, any code whatsoever—until the original thread leaves its critical section.

This brute-force approach can be improved upon by using Semaphores. To enter a critical section, a thread must obtain a semaphore, which it releases on leaving the section. Other threads are prevented from entering the critical section at the same time as the original thread, but are free to gain control of the CPU and execute other code, including other critical sections that are protected by different semaphores. Semaphore locking also has a time limit to prevent a deadlock condition in which a lock is acquired by a single process for an infinite time stalling the other processes which need to use the shared resource protected by the critical session.

Uses of critical sections

Kernel-level critical sections

Typically, critical sections prevent thread and process migration between processors and the preemption of processes and threads by interrupts and other processes and threads.

Critical sections often allow nesting. Nesting allows multiple critical sections to be entered and exited at little cost.

If the scheduler interrupts the current process or thread in a critical section, the scheduler will either allow the currently executing process or thread to run to completion of the critical section, or it will schedule the process or thread for another complete quantum. The scheduler will not migrate the process or thread to another processor, and it will not schedule another process or thread to run while the current process or thread is in a critical section.

Similarly, if an interrupt occurs in a critical section, the interrupt information is recorded for future processing, and execution is returned to the process or thread in the critical section.[4] Once the critical section is exited, and in some cases the scheduled quantum completed, the pending interrupt will be executed. The concept of scheduling quantum applies to "round-robin" and similar scheduling policies.

Since critical sections may execute only on the processor on which they are entered, synchronization is only required within the executing processor. This allows critical sections to be entered and exited at almost zero cost. No inter-processor synchronization is required. Only instruction stream synchronization[5] is needed. Most processors provide the required amount of synchronization by the simple act of interrupting the current execution state. This allows critical sections in most cases to be nothing more than a per processor count of critical sections entered.

Performance enhancements include executing pending interrupts at the exit of all critical sections and allowing the scheduler to run at the exit of all critical sections. Furthermore, pending interrupts may be transferred to other processors for execution.

Critical sections should not be used as a long-lasting locking primitive. Critical sections should be kept short enough so that it can be entered, executed, and exited without any interrupts occurring from the hardware and the scheduler.

Kernel-level critical sections are the base of the software lockout issue.

Critical sections in data structures

In parallel programming, the code is divided into threads. The read-write conflicting variables are split between threads and each thread has a copy of them. Data structures like linked lists, trees, hash tables etc. have data variables that are linked and cannot be split between threads and hence implementing parallelism is very difficult. To improve the efficiency of implementing data structures multiple operations like insertion, deletion, search need to be executed in parallel. While performing these operations, there may be scenarios where the same element is being searched by one thread and is being deleted by another. In such cases, the output may be erroneous. The thread searching the element may have a hit, whereas the other thread may delete it just after that time. These scenarios will cause issues in the program running by providing false data. To prevent this, one method is that the entire data-structure can be kept under critical section so that only one operation is handled at a time. Another method is locking the node in use under critical section, so that other operations do not use the same node. Using critical section, thus, ensures that the code provides expected outputs.

Critical sections in computer networking

Critical sections are also needed in computer networking. When the data arrives at network sockets, it may not arrive in an ordered format. Let's say program 'X' running on the machine needs to collect the data from the socket, rearrange it and check if anything is missing. While this program works on the data, no other program should access the same socket for that particular data. Hence, the data of the socket is protected by a critical section so that program 'X' can use it exclusively.