***Synchronization between processes:***

* Semaphores
* Shared Memory

**SEMAPHORES**

A semaphore is a kernel-maintained atomic **integer** (JUST AN INTEGER) whose value is restricted to being greater than or equal to 0.

Various operations (i.e., system calls) can be performed on a semaphore, including:

* setting the semaphore to an absolute value;
* adding a number to the current value of the semaphore;
* subtracting a number from the current value of the semaphore;
  + YOU TRY TO SUBTRACT A VALUE FROM THE CURRENT VALUE AND ITS VALUE IS LESS THAN 0. SYSTEM BLOCKS UNTIL VALUE BECOME 0 OR HIGHER THAN 0.
* waiting for the semaphore value to be equal to 0.

The last two of these operations may cause the calling process to block.

When lowering a semaphore value, the kernel blocks any attempt to decrease the value below 0.

Similarly, waiting for a semaphore to equal 0 blocks the calling process if the semaphore value is not currently 0.

In both cases, the calling process remains blocked until some other process alters the semaphore to a value that allows the operation to proceed, at which point the kernel wakes the blocked process

Diagram

Description automatically generated

*Figure illustrates the use of a semaphore to synchronize two processes*

INCREASE IT (EXIT CRITICAL REGION), DECREASE IT (ENTER CR), WAIT FOR IT TO BECOME 0 (BLOCKED)

In terms of controlling the actions of a process, a semaphore has no meaning in and of itself.

Its meaning is determined only by the associations given to it by the processes using the semaphore.

Typically, processes agree on a convention that associates a semaphore with a shared resource, such as a region of shared memory.

Other uses of semaphores are also possible, such as synchronization between parent and child processes after fork().

In POSIX: SEM terminology, the wait and signal operations are called semaphore lock (subtract – lock if semaphore is 0 or negative) and semaphore unlock (add), respectively.

We can think of a semaphore as an integer value and a list of processes waiting for a signal operation.

Semaphore implementations use atomic operations (there wont be any context switch) of the underlying operating system to ensure correct execution

Suppose process 1 must execute statement a before process 2 executes statement b.

The semaphore sync enforces the ordering in the following pseudocode, provided that sync is initially 0.

A picture containing shape

Description automatically generated

a WILL BE DONE BEFORE b

What happens in the following pseudocode if semaphores S and Q are both initialized to 1?

Text, letter

Description automatically generated

FIRST WAIT TO Q OR S IS GONNA DECREASE THEIR VALUE, SECOND WAIT WILL BE BLOCKED.

RESULT OF THIS EXAMPLE CANNOT BE PREDICTED.

PROBABLY AT SOME POINT, BOTH OF THE PROCESSES WILL BE STUCK.

THIS IS BAD SYNCHRONIZATION.

In this course, we will focus on two types of semaphore implementations:

* POSIX Semaphores (or simply semaphore)
* SYSTEM V Semaphores (semaphore sets)

For a realtively simpler introduction to semaphore concept students are refered to old CSE 244 slides and notes.

POSIX Semaphores

POSIX: SEM specifies two types of semaphores:

* Named semaphores (FILE BASED SEMAPHORES): This type of semaphore has a name. By calling sem\_open() with the same name, unrelated processes can access the same semaphore.
* Unnamed semaphores (MEMORY BASED SEMAPHORES): This type of semaphore doesn’t have a name; instead, it resides at an agreedupon location in memory. Unnamed semaphores can be shared between processes or between a group of threads. BETWEEN RELATED PROCESSES, CHILD-PARENT ETC.

\*\*\*note that some systems might not have a full implementation of POSIX semaphores (i.e. Linux 2.4)

Opening a named semaphore

Text, letter

Description automatically generated

Returns pointer to a semaphore on success, or SEM\_FAILED on error

The sem\_open() function creates and opens a new named semaphore or opens an existing semaphore.

Regardless of whether we are creating a new semaphore or opening an existing semaphore, sem\_open() returns a pointer to a sem\_t value

POSIX Semaphores: Named Semaphores

Note that the results are undefined if we attempt to perform operations (sem\_post(), sem\_wait(), and so on) on a copy of the sem\_t variable pointed to by the return value of sem\_open().

In other words, the following use of sem2 is not permissible:

Shape

Description automatically generatedOUTCOME IS UNPREDICTABLE

When a child is created via fork(), it inherits references to all of the named semaphores that are open in its parent. After the fork(), the parent and child can use these semaphores to synchronize their actions.

Closing a Semaphore

When a process opens a named semaphore, the system records the association between the process and the semaphore.

The sem\_close() function terminates this association, releases any resources that the system has associated with the semaphore for this process, and decreases the count of processes referencing the semaphore.

A picture containing shape

Description automatically generated

Returns 0 on success, -1 on error

Closing a semaphore does not delete it. For that purpose, we need to use sem\_unlink().

BY CALLING sem\_open AGAIN, YOU CAN REACH SEMAPHORE VALUE AGAIN.

SEMAPHORE CONTAINS VALUE INSIDE KERNEL.

Removing a Named Semaphore

The sem\_unlink() function removes the semaphore identified by name and marks the semaphore to be destroyed once all processes cease using it.

A picture containing diagram

Description automatically generated

Returns 0 on success, -1 on error

BEFORE UNLINKING A SEMAPHORE, BE SURE THAT NO PROCESSES ARE BLOCKED BY SEMAPHORE.

Semaphore Operations

***Waiting on a Semaphore***

Text

Description automatically generated

Returns 0 on success, -1 on error

The sem\_wait() function decrements (decreases by 1) the value of the semaphore referred to by sem.

If the semaphore currently has a value greater than 0, sem\_wait() returns immediately. If the value of the semaphore is currently 0, sem\_wait() blocks until the semaphore value rises above 0; at that time, the semaphore is then decremented and sem\_wait() returns.

The sem\_trywait() function is a nonblocking version of sem\_wait().

IF YOU DON’T WANT YOUR PROCESS TO BE BLOCKED AND SEE IF ANY OTHER PROCESSES WAIT, YOU CAN USE sem\_trywait.

WAIT THERE FOR SPECIFIC TIME 🡪 sem\_timedwait

***Posting a Semaphore***

Shape

Description automatically generated

Returns 0 on success, -1 on error

SIGNAL IN SHORT.

WAIT-SIGNAL / LOCK-UNLOCK / WAIT-POST

IF THEY ARE MORE THAN 1 PROCESS WAITING FOR SEMAPHORE, YOU HAVE NO IDEA WHICH PROCESS WILL BE UNBLOCKED.

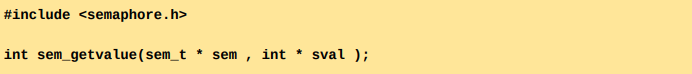
The sem\_post() function increments (increases by 1) the value of the semaphore referred to by sem.

If the value of the semaphore was 0 before the sem\_post() call, and some other process (or thread) is blocked waiting to decrement the semaphore, then that process is awoken, and its sem\_wait() call proceeds to decrement the semaphore.

If multiple processes (or threads) are blocked in sem\_wait(), then, if the processes are being scheduled under the default time-sharing policy, it is indeterminate which one will be awoken and allowed to decrement the semaphore.

Retrieving the Current Value of a Semaphore

The sem\_getvalue() function returns the current value of the semaphore referred to by sem in the int pointed to by sval.



Returns 0 on success, -1 on error

If one or more processes (or threads) are currently blocked waiting to decrement the semaphore’s value, then the value returned in sval depends on the implementation.

There are two possibilities:

* 0 or
* a negative number whose absolute value is the number of waiters blocked in sem\_wait() (On Linux the former behavior is adapted)

**POSIX Semaphores: Unnamed Semaphores**

Unnamed semaphores (also known as memory-based semaphores) are variables of type sem\_t that are stored in memory allocated by the application.

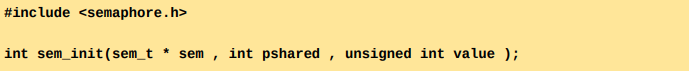
The semaphore is made available to the processes or threads that use it by placing it in an area of memory that they share.

Operations on unnamed semaphores use the same functions that are used to operate on named semaphores. In addition, two further functions are required:

* The sem\_init() function initializes a semaphore and informs the system of whether the semaphore will be shared between processes or between the threads of a single process.
* The sem\_destroy(sem) function destroys a semaphore

Initializing an Unnamed Semaphore

The sem\_init() function initializes the unnamed semaphore pointed to by sem to the value specified by value.



Returns 0 on success, -1 on error

value WILL BE THE INITIAL VALUE

The pshared argument indicates whether the semaphore is to be shared between threads or between processes.

* If pshared is 0, then the semaphore is to be shared between the threads of the calling process. In this case, sem is typically specified as the address of either a global variable or a variable allocated on the heap
* If pshared is nonzero, then the semaphore is to be shared between processes. In this case, sem must be the address of a location in a region of shared memory (a POSIX shared memory object, a shared mapping created using mmap(), or a System V shared memory segment).
  + SINCE IT IS UNNAMED, YOU CAN ONLY USE IT BETWEEN PARENT AND CHILD PROCESSES OR BETWEEN CHILD PROCESSES

Take a peek into the future (first the thread function definition)

SEMAPHORE EXAMPLE BETWEEN PARENT PROCESS AND THREADS GENERATED BY IT

Text, letter

Description automatically generated



CS 🡪 CRITICAL SECTION

Text, letter

Description automatically generated



BOTH OF THE THREADS ARE DOING THE SAME FUNCTION AND THEY USE THE SEMAPHORE INITIALIZED IN MAIN PROCESS HERE. CODE SHOULD HAVE DESTROYED THE SEMAPHORE WHEN IT IS DONE WITH IT.

THREADLER GLOBAL VARIABLELARIN HEPSİNE HAKİM. HANGİ EXECUTIONIN ÖNCE OLACAĞINA HİÇBİR PROCESS KARAR VEREMEZ. SADECE CSLERİ AYIRIYORUZ.

Destroying an Unnamed Semaphore

The sem\_destroy() function destroys the semaphore sem, which must be an unnamed semaphore that was previously initialized using sem\_init(). It is safe to destroy a semaphore only if no processes or threads are waiting on it.



Returns 0 on success, -1 on error

After an unnamed semaphore segment has been destroyed with sem\_destroy(), it can be reinitialized with sem\_init().

An unnamed semaphore should be destroyed before its underlying memory is deallocated. (!!! bevare of leaks !!!)

WHEN WE ARE DONE WITH SEMAPHORES, WE DESTROY IT.

WE DON’T HAVE TO UNLOCK IT BC IT WAS CREATED INSIDE THE INITIAL PARENT PROCESS ANYWAY.

IF YOU DON’T DESTROY THE SEMAPHORE, LOCATED PLACE IN KERNEL DATABASE THAT IS RESERVED FOR THAT INITIAL PROCESS AND UNTIL THE INIT PROCESS UNDERSTANDS YOUR PROGRAM IS FINISHED, IT WILL NOT RELEASE THAT RESOURCE ALLOCATED INSIDE THE KERNEL. MEMORY LEAK!

**Comparisons with Other Synchronization Techniques**

**POSIX semaphores versus System V semaphores**

In general the POSIX IPC interface is simpler and more consistent with the traditional UNIX file model, and POSIX IPC objects are reference counted, which simplifies the task of determining when to delete an IPC object. Aside from these:

* The POSIX semaphore interface is much simpler than the System V semaphore interface. This simplicity is achieved without loss of functional power.
* POSIX named semaphores eliminate the initialization problem associated with System V semaphores (sem. sets)

YOU HAVE TO BE CAREFUL WHILE DOING SYNCHRONIZATION BETWEEN MULTIPLE PROCESSES. ONLY THING IS THIS WITH POSIX.

**POSIX semaphores versus Pthreads mutexes**

POSIX semaphores and Pthreads mutexes can both be used to synchronize the actions of threads within the same process, and their performance is similar.

However, mutexes are usually preferable, because the ownership property of mutexes enforces good structuring of code. By contrast, one thread can increment a semaphore that was decremented by another thread. This flexibility can lead to poorly structured synchronization designs.

There is one circumstance in which mutexes can’t be used in a multi-threaded application and semaphores may therefore be preferable.

Because it is async-signal safe , the sem\_post() function can be used from within a signal handler to synchronize with another thread. This is not possible with mutexes, because the Pthreads functions for operating on mutexes are not async-signal-safe (SIGNAL GELDİĞİNDE YENİDEN BAŞLATMIYOR.)(!! ?? !!). However, because it is usually preferable to deal with asynchronous signals by accepting them using sigwaitinfo() (or similar), rather than using signal handlers, this advantage of semaphores over mutexes is seldom required.

PTHREAD MUTEXES ARE EASIER TO USE.

SOME OF THE OPERATIONS OF PTHREAD MUTEXES ARE NOT ATOMIC. IT CAN BE DEAL WITH SIGNAL MANIPULATION.

**SHARED MEMORY**

Shared memory allows two or more processes to share the same region (usually referred to as a segment) of physical memory.

Since a shared memory segment becomes part of a process’s user-space memory, no kernel intervention is required for IPC. All that is required is that one process copies data into the shared memory; that data is immediately available to all other processes sharing the same segment.

This provides fast IPC by comparison with techniques such as pipes or message queues, where the sending process copies data from a buffer in user space into kernel memory and the receiving process copies in the reverse direction.

The fact that IPC using shared memory is not mediated by the kernel implies that some method of synchronization is required so that processes don’t simultaneously access the shared memory

PROCESSES DON’T HAVE TO DO SYSTEM CALLS.

WHEN YOU ASSIGN IT, ALLOCATE SIZE ON IT, IT IS SAME FOR BOTH PROCESSES. YOU CAN EASILY SEND DATA BETWEEN THOSE PROCESSES. YOU HAVE TO DO SYNCHRONIZATION.

POSIX Shared Memory

POSIX shared memory allows the user to share a mapped region between unrelated processes without needing to create a corresponding mapped file.

To use a POSIX shared memory object, we perform two steps:

1. Use the shm\_open() function to open an object with a specified name. The shm\_open() function is analogous to the open() system call. It either creates a new shared memory object or opens an existing object.
   1. IF OTHER PROCESS KNOWS THE SPECIFIED NAME, IT CAN ALSO SHARE SAME MEMORY SPACE WITH YOU.
   2. THEN YOU HAVE TO INITIALIZE THE SIZE OF IT.
   3. PASS FILE DESCRIPTOR TO A SPECIFIC VALUE ON YOUR SYSTEM.
2. Pass the file descriptor obtained in the previous step in a call to mmap() that specifies MAP\_SHARED in the flags argument. This maps the shared memory object into the process’s virtual address space.

SHM\_OPEN() OPENS THE SHARED MEMORY LOCATION.

USE MMAP() TO CREATE THE FILE DESCRIPTOR.

Creating Shared Memory Objects

The shm\_open() function creates and opens a new shared memory object or opens an existing object. The arguments to shm\_open() are analogous to those for open().

Text

Description automatically generated

Returns file descriptor on success, or –1 on error

THERE IS NO SIZE WHILE OPENING THE MEMORY SPACE.

When a new shared memory object is created, it initially has zero length. This means that, after creating a new shared memory object, we normally call ftruncate() to set the size of the object before calling mmap() (MAP DESCRIPTOR TO A VARIABLE). Following the mmap() call, we may also use ftruncate() to expand or shrink the shared memory object as desired.

***Example Program : > $ pshm\_create -c /demo\_shm 10000***

Text, letter

Description automatically generated

Text, letter

Description automatically generated

Using Shared Memory Objects

The programs given on the next two slides demonstrate the use of a shared memory object to transfer data from one process to another. The first program copies the string contained in its second command-line argument into the existing shared memory object named in its first command-line argument.

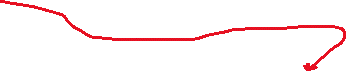
Before mapping the object and performing the copy, the program uses ftruncate() to resize the shared memory object to be the same length as the string that is to be copied.

IF SOME OTHER PROCESSES WANT TO USE SHARED MEMORY, MMAP CAN BE USED BY THEM TO ACCESS THE SHARED MEMORY LOCATION.

***Example Program: writing to a shared memory object***

Text

Description automatically generated



OPENS THE SHARED MEMORY. ADJUST SIZE OF IT. ASSIGN VALUE OF IT TO ADDR. COPY CONTENTS OF ARGV[2] TO SHARED MEMORY SPACE.

***Example Program: reading from a shared memory object***

Text

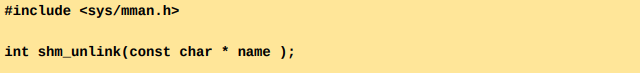
Description automatically generated with medium confidence

OPENS THE SHARED MEMORY. DON’T DO ANY SIZE STUFF. MAPPING IS DONE HERE TOO BUT WITH READ ONLY (NOT USE PROT\_WRITE).

PRINTS THE CONTENT OF ADDR (SHARED MEMORY) TO STDOUT.

Removing Shared Memory Objects

When a shared memory object is no longer required, it should be removed using shm\_unlink().



Returns 0 on success, or –1 on error

The shm\_unlink() function removes the shared memory object specified by name. Removing a shared memory object doesn’t affect existing mappings of the object (which will remain in effect until the corresponding processes call munmap() or terminate), but prevents further shm\_open() calls from opening the object. Once all processes have unmapped the object, the object is removed, and its contents are lost.

THE ONE WHO CREATED SHARED MEMORY OBJECT HAS TO REMOVE CONTENTS OF IT. CREATOR ALSO HAS TO BE SURE THAT NO OTHER PROCESSES ARE TRYING TO REACH SHARED MEMORY BC THEY WONT REACH AND RETURN WITH ERROR CODE.

**OVERALL**

Shared Memory provide fast IPC, and applications typically must use a semaphore (or other synchronization primitive) to synchronize access to the shared region.

Once the shared memory region has been mapped into the process’s virtual address space, it looks just like any other part of the process’s memory space.

The system places the shared memory regions within the process virtual address space.

Assuming that we don’t attempt to map a shared memory region at a fixed address, we should ensure that all references to locations in the region are calculated as offsets (rather than pointers), since the region may be located at different virtual addresses within different processes.

IN SOME OCCASIONS, YOU USE SHARED MEMORY BETWEEN THREADS TOO.

SHARED MEMORY IS NICE WAY OF SHARING DATA BETWEEN 2 RELATED OR UNRELATED PROCESSES.