Contents

Semaphores

Definition

A semaphore is an object

with an integer value that we can manipulate with two routines;

routines

sem wait() sem post()1.

the initial value

determines its behavior, we must initialize it to some value,

how to use

sem wait()

will either
return right away (because the value of the semaphore was one or higher when we called sem wait()), or it will cause the caller to suspend execution waiting for a subsequent post.

multiple calling threads
may call into sem wait(), and thus all be queued waiting to be woken.

sem post()

does not
wait for some particular condition

it simply
increments the value of the semaphore and then, if there is a thread waiting to be woken, wakes one of them up.

the value of the semaphore, when negative,

is equal to the number of waiting threads

Binary Semaphores (Locks)

we simply surround the critical section of interest with a sem wait()/sem post() pair.

initial value of the semaphore

the initial value should be 1.

trace

Val	Thread 0	State	Thread 1	State
1		Run		Ready
1	call `sem_wait()`	Run		Ready
0	`sem_wait()` returns	Run		Ready
0	`(crit sect begin)`	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	call `sem_wait()`	Run
-1		Ready	`decr sem`	Run
-1		Ready	`(sem<0)→sleep`	Sleep
-1		Run	*Switch→T0*	Sleep
-1	`(crit sect end)`	Run		Sleep
-1	call `sem_post()`	Run		Sleep
0	`incr sem`	Run		Sleep
0	`wake(T1)`	Run		Ready
0	`sem_post()` returns	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	`sem_wait()` returns	Run
0		Ready	`(crit sect)`	Run
0		Ready	call `sem_post()`	Run
1		Ready	`sem_post()` returns	Run

Figure 31.5: **Thread Trace: Two Threads Using A Semaphore**

Semaphores For Ordering

example,

a thread
may wish to wait for a list to become non-empty,

so it can delete an element from it.

In this pattern of usage,

we often find one thread **waiting** for something to happen, and another thread making that something happen and then **signaling** that it has happened, thus waking the waiting thread.

We are thus using the semaphore
as an ordering primitive (similar to our use of condition variables

example

Val	Parent	State	Child	State
0	`create(Child)`	Run	*(Child exists, can run)*	Ready
0	call `sem_wait()`	Run		Ready
-1	`decr sem`	Run		Ready
-1	`(sem<0)→sleep`	Sleep		Ready
-1	*Switch→Child*	Sleep	`child runs`	Run
-1		Sleep	call `sem_post()`	Run
0		Sleep	`inc sem`	Run
0		Ready	`wake(Parent)`	Run
0		Ready	`sem_post()` returns	Run
0		Ready	*Interrupt→Parent*	Ready
0	`sem_wait()` returns	Run		Ready

Figure 31.7: **Thread Trace: Parent Waiting For Child (Case 1)**

The Producer/Consumer (Bounded Buffer) Problem

ASIDE: SETTING THE VALUE OF A SEMAPHORE

what's the general rule for semaphore initialization?

One simple way to think about it,

consider the number of resources you are willing to give away immediately after initialization.

With the lock,

it was 1, because you are willing to have the lock locked (given away) immediately after initialization.

With the ordering case,

it was 0, because there is nothing to give away at the start; only when the child thread is done is the resource created,

A Working Solution

To solve this problem, we simply must reduce the scope of the lock. simply move the mutex acquire and release to be just around the critical section;

Reader-Writer Locks

problem

imagine a number of concurrent list operations, While inserts change the state of the list lookups simply read the data structure; as long as we can guarantee that no insert is on-going, we can allow many lookups to proceed concurrently.

The code

once a reader has acquired a read lock, more readers will be allowed to acquire the read lock too; however, any thread that wishes to acquire the write lock will have to wait until all readers are finished;

should be used with some caution.

They often add more overhead and thus do not end up speeding up performance as compared to just using simple and fast locking primitives

Thread Throttling

problem

how can a programmer prevent “too many” threads from doing something at once and bogging the system down?

Answer:
decide upon a threshold for “too many”, and then use a semaphore to limit the number of threads concurrently executing the piece of code in question.

A simple semaphore can solve this problem.

By initializing the value of the semaphore to the maximum number of threads you wish to enter the memory-intensive region at once, and then putting a sem wait() and sem post() around the region,