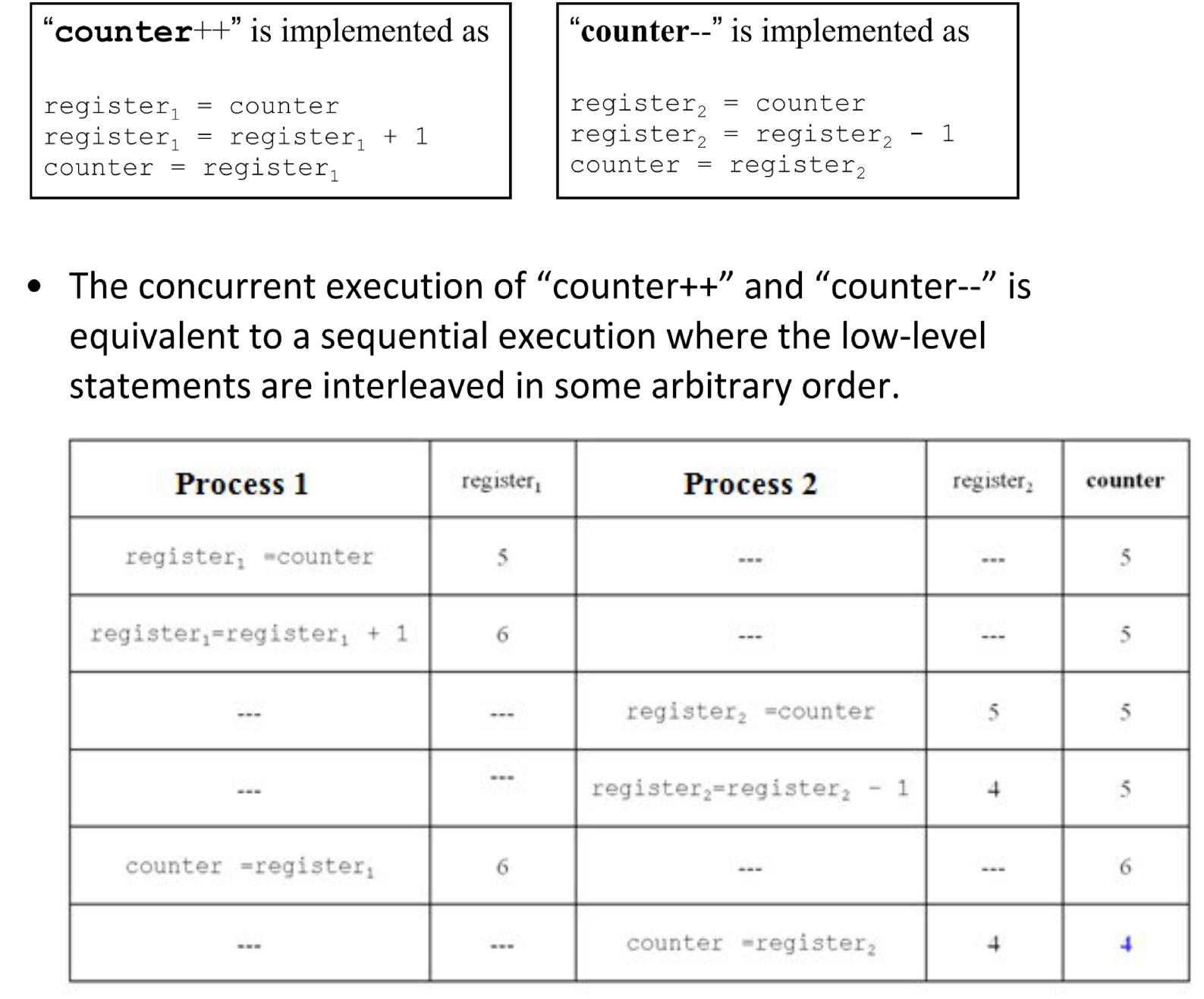
Semaphore for Synchronization

* *Race condition*:A situation that several tasks access and manipulate thesame data concurrently and the outcome of the execution depends on *the* *particular order in which the access take place*.
* Example:

– Suppose that the value of the variable **counter** = 5.

– Process 1 and process 2 execute the statements “**counter**++” and “**counter--**” concurrently.

– Following the execution of these two statements, the value of the variable **counter** may be 4, 5, or 6!



Shared memory synchronization

* There are two essential needs for synchronization between multiple processes executing on shared memory

– **Establishing an order between two events**

* + - E.g. in the server and client case, we want to make sure the server finishes writing before the client reads

– **Mutually exclusive access to a certain resource**

* + - Such as a data structure, a file, etc
    - E.g. Two people deposit to the same account

“deposit +=100”. We want to make sure that the increment happens one at a time. Why? (Let us look draw a time line showing possible interleaving of events)

* A semaphore can be used for both purposes
* An ordinary while loop (busy wait loop) is not safe for ensuring mutual exclusion

– Two processes may both think they have successfully set the lock and, so, have the exclusive access

* + - Again, we can draw a time line showing possible interleaving of events that may lead to failed mutual exclusion

– A semaphore is guaranteed to be able to have the correct view of the locking status

The concept of semaphores

* Semaphores may be *binary* (0/1), or *counting*
* Every semaphore variable, *s*, It is initialized to

some positive value

* 1 for a binary semaphore
* N > 1 for a counting semaphore

Binary semaphores

* **A binary semaphore , s , is used** for mutual exclusion andwake up sync

1 == unlocked

0 == locked

* *s*, is associated with two operations:
* **P(s)**

– Tests *s*; if positive, resets s to 0 and proceed; otherwise, put the executing process to the back of a waiting queue for s

* **V(s)**

– Set s to 1 and wake up a process in the waiting queue for s

Counting semaphores

* **A counting semaphore, s, is used** for producer/consumer sync

n == the count of available resources

0 == no resource (locking consumers out)

* *s*, is associated with two operations:
* **P(s)**

– Tests s, if positive, decrements s and proceed

– otherwise, put the executing process to the back of a waiting queue for s

* **V(s)**

– Increments *s; wakes up a process, if any, in the waiting queue* *for s*

Critical Sections

* We like to think of locking a concurrent data structure
* In current practice, however, locks (incl. binary semaphores) are typically used to lock a segment of program statements (or instructions)
* Such a program segment is called a *critical* *section*

– *A critical section is a program segment that may* *modify shared data structures*

– *It should be executed by one process at any given time*

* With a binary semaphore

– If multiple processes are locked out of a critical section

* + - As soon as the critical section is unlocked, only one process is allowed in
    - The other processes remain locked out
* Implementation of semaphores is fair to processes
  + - A first‐come‐first‐serve queue

Unix Semaphores

* There are actually at least two implementations
* UNIX System V has an old implementation

– Analogous to shared memory system calls

– Calls to semget(), semat(), semctl(), etc

– Not as easy to use as POSIX implementation

* We will use POSIX implementation in

this course

POSIX semaphore system calls

* #include <semaphore.h>
* POSIX semaphores come in two forms: named semaphores and unnamed semaphores.

Using unnamed semaphores

* Unnamed semaphores are also called memory‐ based semaphores

– Named semaphores are “file‐based”

* An unnamed semaphore does not have a name.

– It is placed in a region of memory that is shared between multiple threads (a thread‐shared semaphore) or processes (a process‐shared semaphore).

* A process‐shared semaphore must be placed in a shared memory region

System calls

* Before being used, an unnamed semaphore must be initialized using ***sem\_init****(3)*. It can then be operated on using ***sem\_post****(3)* and ***sem\_wait****(3)*.
* When the semaphore is no longer required, and before the memory in which it is located is deallocated, the semaphore should be destroyed using ***sem\_destroy****(3)*.
* Compile using ‐lrt

Recall that shared memory segments must be removed before program exits

* “An unnamed semaphore should be destroyed with sem\_destroy() before the memory in which it is located is deallocated.”
* “ Failure to do this can result in resource leaks on some implementations.”

**int sem\_init(sem\_t \*sem, int pshared, unsigned int value);**

* #include <semaphore.h>
* **sem\_init**() initializes the unnamed semaphore atthe address pointed to by *sem*. The *value* argument specifies the initial value for the semaphore.
* If pshared has the value 0, then the semaphore is shared between the threads of a process
* If pshared is nonzero, then the semaphore is shared between processes, and should be located in a region of shared memory

**int sem\_wait(sem\_t \*sem);**

* **sem\_wait**() decrements (locks) the semaphorepointed to by *sem*.
* If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately.
* If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call.

**int sem\_post(sem\_t \*sem);**

* **sem\_post**() increments (unlocks) thesemaphore pointed to by *sem*.
* If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a ***sem\_wait****(3)* call will be woken up

**int sem\_destroy(sem\_t \*sem);**

* **Destroys** the unnamed semaphore at the addresspointed to by *sem*. Only a semaphore that has been initialized by ***sem\_init****(3)* should be destroyed using **sem\_destroy**().
* Destroying a semaphore that other processes or threads are currently blocked on (in ***sem\_wait****(3)*) produces undefined behavior.
* Using a semaphore that has been destroyed produces undefined results, until the semaphore has been reinitialized using ***sem\_init****(3)*.

Examples

* We first look at a bad example in which the unnamed semaphore is not placed in the shared memory (test1.c)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  | **int shmid1;** |  |
|  |  |  | **int \*shm1, \*s;** |  |
| **// compile with ‐lrt** |  |  | **if ((shmid1 = shmget(IPC\_PRIVATE,** |  |
| **#include <semaphore.h>** |  |  | **SHMSIZE, 0666)) < 0) {** |  |
|  |  | **perror("shmget");** |  |
| #include <stdio.h> |  |  |  |
|  |  | **exit(1);** |  |
| #include <errno.h> |  |  |  |
|  |  | **}** |  |
| #include <stdlib.h> |  |  |  |
|  |  | **if ((shm1 = shmat(shmid1, NULL, 0)) == (int** |  |
| #include <unistd.h> |  |  | **\*) ‐1) {** |  |
| #include <sys/types.h> |  |  | **perror("shmat");** |  |
| #include <sys/shm.h> |  |  | **exit(1);** |  |
| #include <sys/wait.h> |  |  | **}** |  |
|  |  | **\*shm1 = 0;** |  |
| #define SHMSIZE 1024 |  |  |  |
|  |  | **ptr = shm1;** |  |
| **int main(int argc, char \*\*argv)** | | |  |
|  |  |
| { | • | In this example, the semaphore is not | |  |
| int i,nloop=10,\*ptr; |  |
| **sem\_t mutex;** |  | placed in the shared memory. | |  |
|  | • | Therefore, it is ineffective for mutual | |  |

exclusion synchronization

/\* create, initialize semaphore \*/

**if( sem\_init(&mutex,1,1) < 0)**

{

perror("semaphore

initilization");

exit(0);

}

if (fork() == 0) { /\* child process\*/ **sem\_wait(&mutex);**

for (i = 0; i < nloop; i++) { printf("child: %d\n", (\*ptr)++); **sleep(5);**

}

**sem\_post(&mutex);** exit(0);

}

/\* back to parent process \*/ sem\_wait(&mutex);

for (i = 0; i < nloop; i++) { printf("parent: %d\n",

(\*ptr)++);

**sleep(5);**

}

sem\_post(&mutex); wait(int \*) 0); shmctl(shmid1, IPC\_RMID, (struct shmid\_ds \*) 0); exit(0);

}

* The mutex is supposed to ensure that each process prints its entire data w/o mixing with the other process’ data
* But it fails to do so

* Next, we look at an even worse example:

– We want to let parent process prints its entire data first

– So we let child process wait for the process to give it a go‐ ahead

– Initialize the mutex variable to 0 and wait for the parent process to change it to 1.

* But we didn’t put the mutex variable in the shared memory
* The child process never wakes up!
* We need to manually kill the child process and free the shared memory

}

/\* back to parent process \*/

for (i = 0; i < nloop; i++) printf("parent: %d\n", (\*ptr)++);

**sem\_post**(&mutex);exit(0);

#include ….. **// stuck.c**

int main(int argc, char \*\*argv)

{

int i,nloop=10,\*ptr;

**sem\_t mutex;**

**……**

**if( sem\_init(&mutex,1,1) < 0) \*/**

**if( sem\_init(&mutex,1,0) < 0) { …..**

**}**

**if (fork() == 0) { /\* child process\*/ sem\_wait(&mutex);**

**for (i = 0; i < nloop; i++) printf("child: %d\n", (\*ptr)++); exit(0)**

**}**

* Finally, we will correct the errors by placing the semaphore in the shared memory
* We also need to remember to destroy the unnamed semaphore before removing the shared memory segment.
* Be careful with the timing for destroying the semaphore

– Make sure there should not be waiting processes

**// nonstuck.c sem\_t \*p\_mutex;**

**……**

**if ((shmid2 = shmget(IPC\_PRIVATE, SHMSIZE, 0666)) < 0) { perror("shmget");**

**exit(1);**

**}**

**p\_mutex = (sem\_t \*) shmat(shmid2, NULL, 0); if (p\_mutex == (sem\_t \*) ‐1) {**

**perror("mutex shmat fails "); exit(1);**

**}**

**if( sem\_init(p\_mutex,1,0) < 0)**

**{**

**perror("semaphore initilization");**

**exit(1);**

**}**

**if (fork() == 0) { /\* child process\*/**

**sem\_wait(p\_mutex);** **// cont’d on next page**

// nonstuck.c cont’d

if (fork() == 0) { /\* child process\*/ **sem\_wait(p\_mutex);**

for (i = 0; i < nloop; i++) printf("child: %d\n", (\*ptr)++); **sem\_destroy(p\_mutex);**

**shmctl(shmid2, IPC\_RMID, (struct shmid\_ds \*) 0); shmctl(shmid1, IPC\_RMID, (struct shmid\_ds \*) 0);** exit(0);

}

/\* back to parent process \*/ for (i = 0; i < nloop; i++) printf("parent: %d\n", (\*ptr)++); **sem\_post(p\_mutex);**

exit(0);

* We can make a similar change to test1.c
* We will see that now each process will print its entire data without interleaving with other processes
* Which process writes first will be unknown in advance