CS 332/532 Systems Programming

Lecture 33

Semaphores

Professor : Mahmut Unan – UAB CS

Agenda

- Semaphores
- Thread synchronization using semaphores

Semaphore

- The fundamental principle is this: Two or more processes can cooperate by means of simple signals, such that a process can be forced to stop at a specified place until it has received a specific signal.
- Any complex coordination requirement can be satisfied by the appropriate structure of signals.
- For signaling, special variables called semaphores are used.
- To transmit a signal via semaphore s , a process executes the primitive semSignal(s) .
- To receive a signal via semaphore s , a process executes the primitive semWait(s) ; if the corresponding signal has not yet been transmitted, the process is suspended until the transmission takes place.

Semaphores

- While mutexes are one solution to ensure synchronization, semaphores provide an alternative and more generalized approach to establish synchronization among multiple threads.
- While mutexes provide a locking mechanism (both lock and unlock are performed by a single thread - *cooperative locks*), semaphores provide a signaling mechanism (two different threads cooperate with the wait/signal calls) to synchronize access to shared resources.
- Note that it is possible to implement a mutex using a binary semaphore.

Semaphore

A variable that has an integer value upon which only three operations are defined:

- There is no way to inspect or manipulate semaphores other than these three operations

1) A semaphore may be initialized to a nonnegative integer value
2) The semWait operation decrements the semaphore value
3) The semSignal operation increments the semaphore value

- The Pthread library provides the wait and signal calls that are described in the text book. However, since Linux uses the word signal for software interrupts, the wait and signal calls are called *sem_wait* and *sem_post*.
- The C APIs for *sem_wait* and *sem_post* are:

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

- We have to include the header file *semaphore.h* to compile and link with the pthread library using -*lpthread*.
- The *sem_wait* call decrements the semaphore variable *sem*.
- The *sem_wait* call returns immediately if the value of *sem* is greater than zero after decrementing the semaphore.
- If the current value of the semaphore is zero, then *sem_wait* call will block until the semaphore values goes above zero or it is interrupted by a signal handler.
- Similarly, the *sem_post* call increments the semaphore variable *sem* and if the value of *sem* goes above zero, it will then wake up the corresponding thread or process that was blocked in a *sem_wait* call.

- The *sem_init* function must be used to initialize a semaphore and the *sem_destroy* method to destroy a semaphore.
- The C API for the *sem_init* and *sem_destroy* are given below:

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned
int value);

int sem_destroy(sem_t *sem);
```

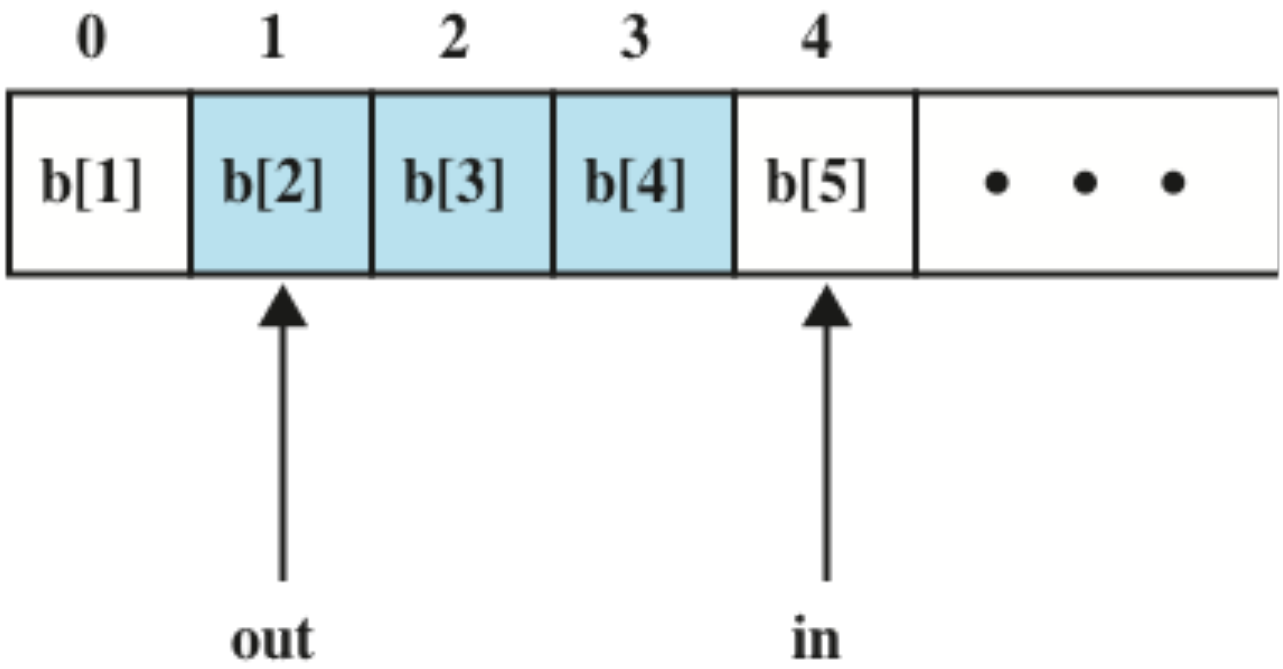
- The *sem_init* call will assign a semaphore *sem* with the initial value provided and the second parameter *pshared* specifies whether the semaphore is shared between threads in with a process or it is shared between processes.

- If the value of *pshared* is zero, then the semaphore is shared between threads and the semaphore must be declared in a shared address space so that all threads can access it.

- If the value of *pshared* is nonzero, then the semaphore will be shared between processes and the semaphore must be declared in a shared memory region

Producer/Consumer Problem

General Statement:	One or more producers are generating data and placing these in a buffer
	A single consumer is taking items out of the buffer one at a time
	Only one producer or consumer may access the buffer at any one time
The Problem:	Ensure that the producer won't try to add data into the buffer if its full, and that the consumer won't try to remove data from an empty buffer



Note: shaded area indicates portion of buffer that is occupied

Figure 5.11 Infinite Buffer for the Producer/Consumer Problem

- We will implement the bounded-buffer producer-consumer problem using semaphores here. In this exercise we will consider the case of a single producer and single consumer and use threads to create a producer and a consumer.
- We use the pseudo code from the textbook (Figure 5.16 on page 228) and replace semWait and semSignal with *sem_wait* and *sem_post*.
- This code is based on the examples provided in the classic book - UNIX Networking Programming, Volume 2 by W. Richard Stevens

prodcons1.c

```c
/* Solution to the single Producer/Consumer problem using semaphores.
   This example uses a circular buffer to put and get the data
   (a bounded-buffer).
   Source: UNIX Network Programming, Volume 2 by W. Richard Stevens

   To compile: gcc -O -Wall -o <filename> <filename>.c -lpthread
   To run: ./<filename> <#items>

   To enable printing add -DDEBUG to compile:
   gcc -O -Wall -DDEBUG -o <filename> <filename>.c -lpthread
*/

/* include globals */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>     /* for POSIX threads */
#include <semaphore.h>   /* for POSIX semaphores */

```

```
18
19  #define  NBUFF          10
20
21  int      nitems;                                  /* read-only */
22
23  struct {                  /* data shared by producer and consumer */
24    int       buff[NBUFF];
25    sem_t     mutex, nempty, nstored;   /* semaphores, not pointers */
26  } shared;
27
28  void       *producer(void *), *consumer(void *);
29
30  /* end globals */
31
32  /* main program */
33  int main(int argc, char **argv)
34  {
35    pthread_t       tid_producer, tid_consumer;
36
37    if (argc != 2) {
38      printf("Usage: %s <#items>\n", argv[0]);
39      exit(-1);
40    }
41
```

```
41
42      nitems = atoi(argv[1]);
43
44      /* initialize three semaphores */
45      sem_init(&shared.mutex, 0, 1);
46      sem_init(&shared.nempty, 0, NBUFF);
47      sem_init(&shared.nstored, 0, 0);
48
49      /* create one producer thread and one consumer thread */
50      pthread_create(&tid_producer, NULL, producer, NULL);
51      pthread_create(&tid_consumer, NULL, consumer, NULL);
52
53      /* wait for producer and consumer threads */
54      pthread_join(tid_producer, NULL);
55      pthread_join(tid_consumer, NULL);
56
57      /* remove the semaphores */
58      sem_destroy(&shared.mutex);
59      sem_destroy(&shared.nempty);
60      sem_destroy(&shared.nstored);
61
62      return 0;
63  }
64  /* end main */
```

```c
/* producer function */
void *producer(void *arg)
{
  int i;

  for (i = 0; i < nitems; i++) {
    sem_wait(&shared.nempty);       /* wait for at least 1 empty slot */
    sem_wait(&shared.mutex);

    shared.buff[i % NBUFF] = i;     /* store i into circular buffer */
#ifdef DEBUG
    printf("wrote %d to buffer at location %d\n", i, i % NBUFF);
#endif

    sem_post(&shared.mutex);
    sem_post(&shared.nstored);      /* 1 more stored item */
  }
  return (NULL);
}
/* end producer */
```

```c
/* consumer function */
void *consumer(void *arg)
{
  int i;

  for (i = 0; i < nitems; i++) {
    sem_wait(&shared.nstored);       /* wait for at least 1 stored item */
    sem_wait(&shared.mutex);

    if (shared.buff[i % NBUFF] != i)
      printf("error: buff[%d] = %d\n", i, shared.buff[i % NBUFF]);
#ifdef DEBUG
    printf("read %d from buffer at location %d\n",
            shared.buff[i % NBUFF], i % NBUFF);
#endif

    sem_post(&shared.mutex);
    sem_post(&shared.nempty);        /* 1 more empty slot */
  }
  return (NULL);
}
/* end consumer */
```

- **gcc -O -Wall -DDEBUG prodcons1.c -lpthread**
  $ **./a.out 20**
  wrote 0 to buffer at location 0
  wrote 1 to buffer at location 1
  wrote 2 to buffer at location 2
  wrote 3 to buffer at location 3
  wrote 4 to buffer at location 4
  wrote 5 to buffer at location 5
  wrote 6 to buffer at location 6
  wrote 7 to buffer at location 7
  wrote 8 to buffer at location 8
  read 0 from buffer at location 0
  read 1 from buffer at location 1
  read 2 from buffer at location 2
  read 3 from buffer at location 3
  read 4 from buffer at location 4
  read 5 from buffer at location 5
  read 6 from buffer at location 6
  read 7 from buffer at location 7
  read 8 from buffer at location 8
  wrote 9 to buffer at location 9
  wrote 10 to buffer at location 0
  wrote 11 to buffer at location 1
  wrote 12 to buffer at location 2
  wrote 13 to buffer at location 3
  wrote 14 to buffer at location 4
  wrote 15 to buffer at location 5
  wrote 16 to buffer at location 6
  wrote 17 to buffer at location 7
  wrote 18 to buffer at location 8
  read 9 from buffer at location 9
  read 10 from buffer at location 0
  read 11 from buffer at location 1
  read 12 from buffer at location 2
  read 13 from buffer at location 3
  read 14 from buffer at location 4
  read 15 from buffer at location 5
  read 16 from buffer at location 6
  read 17 from buffer at location 7
  read 18 from buffer at location 8
  wrote 19 to buffer at location 9
  read 19 from buffer at location 9

- The source code is self-explanatory, we will focus on key sections of the code here.
- First, we define global variables such as the number of items (*nitems*) the producer will produce and the consumer will consume.
- Then we create a shared region, called *shared*, that will be shared between the producer and consumer threads.
- It contains the buffer shared by the producer and consumer and the three semaphores: one for the mutex lock (*mutex*), one for number of empty slots (*nempty*), and one for the number of slots filled (*nstored*) (these correspond to semaphores *s*, *e*, and *n*, respectively, from the textbook).

```
int      nitems;          /* read-only */
struct {
/* data shared by producer and consumer */
   int       buff[NBUFF];
    sem_t     mutex, nempty, nstored;
      /* semaphores */
} shared;
```

- In the main function, we read the number of items to be produced/consumed as a command-line argument and initialize the three semaphores using *sem_init* as per the pseudocode from the textbook.

```
nitems = atoi(argv[1]);

/* initialize three semaphores */
sem_init(&shared.mutex, 0, 1);
sem_init(&shared.nempty, 0, NBUFF);
sem_init(&shared.nstored, 0, 0);
```

Semaphores

- Generalization of the `semWait` and `semSignal` primitives
  - No other process may access the semaphore until all operations have completed

Consists of:

- Current value of the semaphore
- Process ID of the last process to operate on the semaphore
- Number of processes waiting for the semaphore value to be greater than its current value
- Number of processes waiting for the semaphore value to be zero

Semaphores

Solaris provides classic counting semaphores with the following primitives:

- sema_p() Decrements the semaphore, potentially blocking the thread
- sema_v() Increments the semaphore, potentially unblocking a waiting thread
- sema_tryp() Decrements the semaphore if blocking is not required

Semaphores

- User level:
  - Linux provides a semaphore interface corresponding to that in UNIX SVR4

– Internally:
  - Implemented as functions within the kernel and are more efficient than user-visable semaphores

– Three types of kernel semaphores:
  – Binary semaphores
  – Counting semaphores
  – Reader-writer semaphores

Traditional Semaphores	
void sema_init(struct semaphore *sem, int count)	Initializes the dynamically created semaphore to the given count
void init_MUTEX(struct semaphore *sem)	Initializes the dynamically created semaphore with a count of 1 (initially unlocked)
void init_MUTEX_LOCKED(struct semaphore *sem)	Initializes the dynamically created semaphore with a count of 0 (initially locked)
void down(struct semaphore *sem)	Attempts to acquire the given semaphore, entering uninterruptible sleep if semaphore is unavailable
int down_interruptible(struct semaphore *sem)	Attempts to acquire the given semaphore, entering interruptible sleep if semaphore is unavailable; returns -EINTR value if a signal other than the result of an up operation is received
int down_trylock(struct semaphore *sem)	Attempts to acquire the given semaphore, and returns a nonzero value if semaphore is unavailable
void up(struct semaphore *sem)	Releases the given semaphore
Reader-Writer Semaphores	
void init_rwsem(struct rw_semaphore, *rwsem)	Initializes the dynamically created semaphore with a count of 1
void down_read(struct rw_semaphore, *rwsem)	Down operation for readers
void up_read(struct rw_semaphore, *rwsem)	Up operation for readers
void down_write(struct rw_semaphore, *rwsem)	Down operation for writers
void up_write(struct rw_semaphore, *rwsem)	Up operation for writers

Table 6.5

Linux Semaphores

(Table can be found on page 293 in textbook)