ECE437/CS481

M04B: PROCESS COORDINATION SEMAPHORES

CHAPTER 5.6

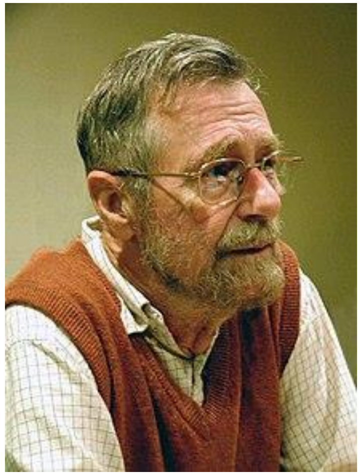
Xiang Sun

The University of New Mexico

Semaphore as Abstraction

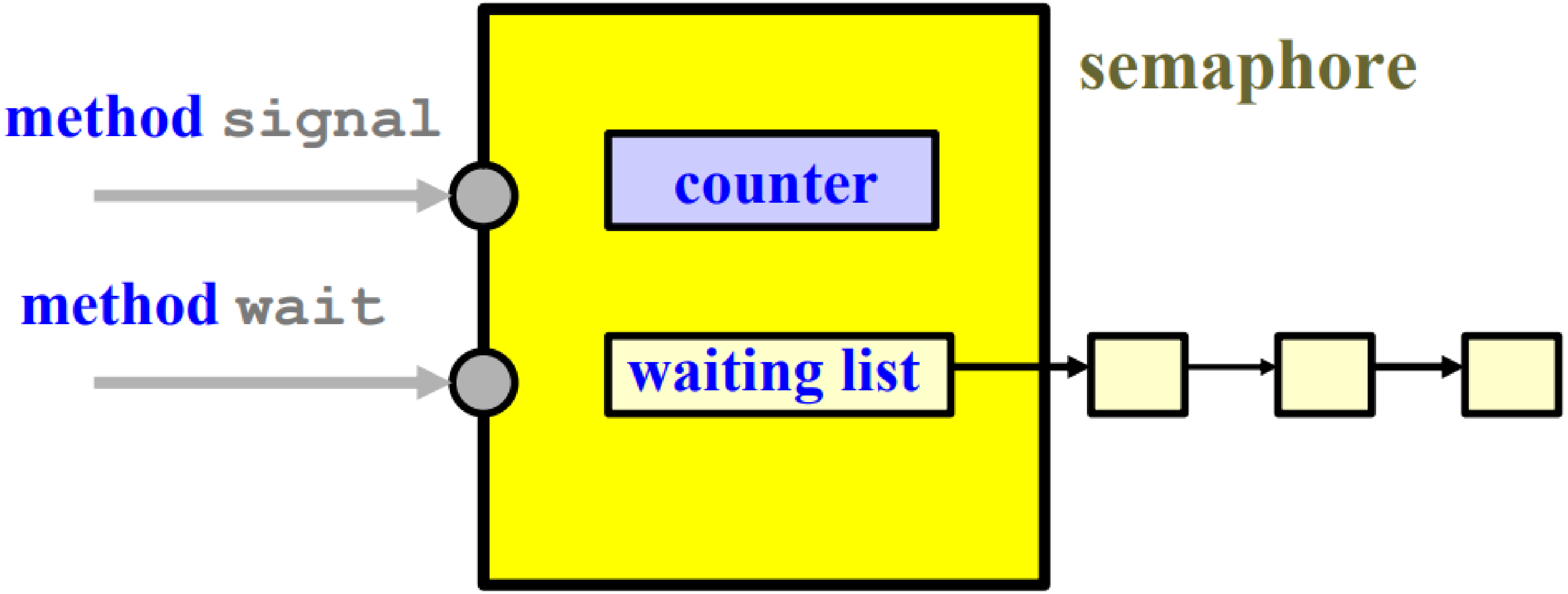
- Motivation
  - Using different schemes to handle critical section in a low layer.
  - Hide the implementation details from users---user friendly
  - General solutions should be machine independent and relatively easy to use

- OS mechanisms for synchronization---Semaphore
  - Introduced by Edsger W. Dijkstra
    - A Dutch computer scientist
    - ACM Turing Award winner
    - A professor retired at UT Austin



What is semaphore?

- A semaphore is an object that consists of a counter, a waiting list of processes and two methods (e.g., functions): signal() and wait()



Semaphore as Abstraction

❑ wait() method in semaphore

- ➢ Wait for semaphore (s) to become positive and then decrement

```
sem_t  s;

wait(s){
 while (s <= 0)
 {}//do no-operation, just wait in the list;
 s--;
}
```

- ✓ Semaphore (s) is an integer value
- ✓ wait(s) is an atomic operation, i.e., uninterruptible operation

Semaphore as Abstraction

- signal() method in semaphore

  - Increment semaphore (s) by 1.

```
sem_t  s;

signal(s){
    s++;
}
```

    - ✓ signal() is an atomic operation, i.e., uninterruptible operation

Applications of Semaphore

❑ Protect critical section (CS)

```
semaphore S = 1;
int  count = 0;
```

Thread 1	Thread 2	
while (1) {	while (1) {	
//do something	//do something	
wait(S);	wait(S);	Entry
count++;	count--;	Critical sections
signal(S);	signal(S);	Exit
//do something	//do something	
}	}	

```
wait(s){
 while (s <= 0)
 {}//do no-operation, just wait in the list;
 s--;
}

signal(s){
     s++;
}
```

- ✓ S is a binary semaphore.
- ✓ What if the initial value of S=0?

Applications of Semaphore

❑ Enforce process synchronization

- ✓ Define S as a binary semaphore.
- ✓ Initially, S=0.
- ✓ Problem requirements: start A_{13} after completion of A_{01}.

Thread 1	Thread 2
A01;	A11;
signal(S);	A12;
A02;	wait(S)
A03;	A13;

Applications of Semaphore

❑ Notification

```
semaphore S1 = 1, S2=0
int  count = 0;

 Thread 1                          Thread 2
while (1) {                        while (1) {
  //do something                     //do something
  wait(S1);         notify           wait(S2);
  printf(“1”);                       printf(“2”);
  signal(S2);       notify           signal(S1);
  //do something                     //do something
}                                  }
```

- ✓ Thread 1 uses signal(S2) to notify Thread 2, indicating “I am done. Please go ahead.”
- ✓ The output is 1 2 1 2 1 2….
- ✓ What if both S1 and S2 are initialized to 0?

Applications of Semaphore

- Counting Semaphore—represent a resource with multiple instances

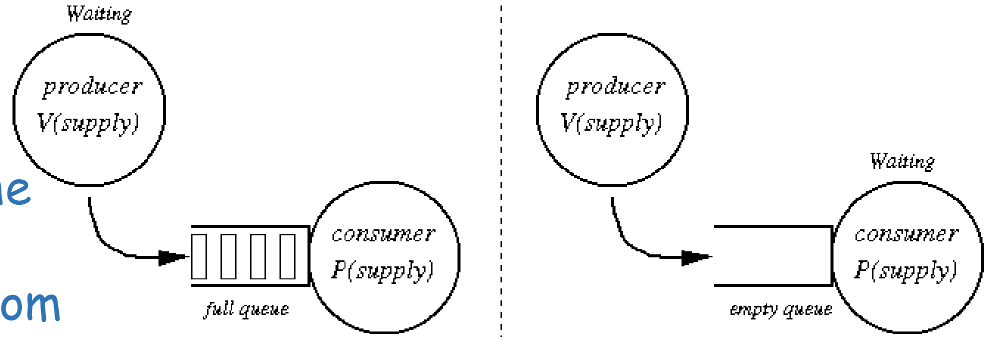
```
semaphore S = 3;

        Thread 1                    Thread 2
while (1) {                  while (1) {
   //do something               //do something
   wait(S);                     wait(S);              Entry
        At most 3 processes can be here!!
   signal(S);                   signal(S);            Exit
   //do something               //do something
}                            }
```

- After three processes pass through wait(), the red section is locked until a process calls signal().
- Solving producer/consumer problem

Applications of Semaphore

❑ Producer/Consumer (bounded-buffer) problem



Producer and Consumer

- Assumption:
  - The producer's job is to generate data item, put it into the buffer, and start again.
  - The consumer is to consume data item (i.e., removing it from the buffer).
- Requirements:
  - To make sure that the producer won't try to add a data item into the buffer if it's full and that the consumer won't try to remove a data item from an empty buffer.
  - If the producer is adding a data item into the buffer, the consumer is not allowed to remove a data item from the buffer.
  - If the consumer is removing a data item from the buffer, the producer is not allowed to add a data item into the buffer.
- The Producer/Consumer problem can be extended to the scenario, where multiple threads/processes try to read/write the same buffer.

Applications of Semaphore

❑ Producer/Consumer (bounded-buffer) problem

- Solving the problem by applying semaphore
  - Define one binary semaphore to achieve mutual exclusion in updating buffer.
  - Define two counting semaphores: one is to keep track of the number of occupied entries in the buffer, and the other is to keep track of the number of unoccupied entries in the buffer.

```
Semaphores: mutex, empty, full;

mutex = 1; // for mutual exclusion
empty = N; // number empty buffer entries
full = 0; // number full buffer entries */
```

Applications of Semaphore

❑ Producer/Consumer (bounded-buffer) problem

➢ Solving the problem by applying semaphore

semaphore mutex = 1, empty = N, full = 0;

	Producer	Consumer
	while (1) {	while (1) {
T1	wait(empty);	wait(full);
T2	wait(mutex);	wait(mutex);
T3	//produce item	//consume item
T4	signal(mutex);	signal(mutex);
T5	signal(full);	signal(empty);
	}	}

❑ Producer/Consumer (bounded-buffer) problem

- If we set up the binary semaphore before counting semaphores, i.e.,

semaphore mutex = 1, empty = N, full = 0;

	Producer	Consumer
	while (1) {	while (1) {
T1	wait(mutex);	wait(mutex)
T2	wait(empty);	wait(full);
T3	//produce item	//consume item
T4	signal(full);	signal(empty);
T5	signal(mutex)	signal(mutex)
	}	}

Will this solution solve the producer/consumer problem?

Implementation of Semaphores

- Semaphores can achieve mutual exclusive as long as signal() and wait() are atomic operations (i.e., cannot be interrupted).

- However, signal() and wait() are not atomic.

```
wait(s){
 while (s <= 0)
 //do no-operation, just wait in the list;
 s--;
}
```

```
signal(s){
    s++;
}
```

> If two processes execute wait() and signal() on the same semaphore s at the same time, semaphore s may not be synchronized.

Implementation of Semaphores

- In the symmetrical multiprocessing (SMP) architecture, a process accessing to memory location excludes other processes accessing to the same location.
    - A machine instruction can perform more than one action atomically on the same memory location.
        - ✓ Test_and_Set
        - ✓ Comp_and_Swap
        - ✓ Fetch_and_Add
        - ✓ ...
    - The execution of such machine instruction is also mutually exclusive, even with multiple CPU/cores.

Implementation of Semaphores

❑ Test_and_Set Instruction

It's logic function

```
Test_and_Set (int *x) {
    int temp = *x;
    *x = TRUE;
    return temp;
  }
```

Old value of x	New value of x	return value
TRUE	TRUE	TRUE
FALSE	TRUE	FALSE

- Two actions: read and write (same memory location, i.e., x)
- Two possible cases when instruction Test_and_Set is applied to x

Implementation of Semaphores

❑ Implementing Semaphores by Test_and_Set Instruction

Old value of x	New value of x	return value	
TRUE	TRUE	TRUE	→ Other process is executing wait()
FALSE	TRUE	FALSE	→ No process is executing wait()

```
Bool lock0=false; //lock free

wait(s){
  while (1){
   while (Test_and_Set(&lock0))
    {}; //do nothing, just wait;
   if (s>0) {
     s--;
     lock0=false; //set lock free
     break;}
    else {lock0=false; //set lock free}
  }
}
```

```
signal(s){
     while (Test_and_Set(&lock0))
      {}; //do nothing, just wait;
      s++;
      lock0=false; //set lock free
}
```

Busy-waiting and Non-busy-waiting Semaphore

❑ Busy-waiting Semaphore

- Recall that

```
Bool lock0=false; //lock free
wait(s){
  while (1){
    while (Test_and_Set(lock0))
      {}; //do nothing, just wait;
      if (s>0) {
        s--; lock0=false; //set lock free
        break;}
      else {lock0=false; //set lock free}}
```

```
signal(s){
    while (Test_and_Set(lock0))
     {}; //do nothing, just wait;
     s++;
     lock0=false; //set lock free
}
```

❑ Drawbacks of busy-waiting semaphore

- If s<=0, processes continuously checking the value of s, thus wasting processor resources.

- Deadlock
  - ✓ if a low priority process, P_A, is within the CS and another higher priority process, P_B, comes and tries to access the CS. In this case, P_B will obtain the processor to wait for P_A to exit the CS, while P_A is blocked since the processor is executing P_B.

Busy-waiting and Non-busy-waiting Semaphore

❑ Non-busy-waiting Semaphore

- Introduce the queueing system into semaphore

```
typedef struct {
int count;
queue q; /* queue of threads/processes waiting on this semaphore */
} Semaphore;
```

❑ Wait() and Signal() in Non-busy-waiting Semaphore

```
Bool lock0=false; //lock free
wait(s){
  while (Test_and_Set(lock0))
    {}; //do nothing, just wait;
    if (s>0) {
      s--; lock0=false; //set lock free
      break;}
    else {
    add this process into queue q;//block this process
    lock0=false; //set lock free}}
```

```
signal(s){
    while (Test_and_Set(lock0))
     {}; //do nothing, just wait;
     if (queue q=empty){
       s++;}
     else{
       remove a process P from queue q;
       wakeup(P);//put process P into ready queue}
     lock0=false; //set lock free}
```

❑ POSIX semaphores

- Include semaphore.h
- Compile the code by linking with -lrt
- To wait/lock a semaphore

```
int sem_wait(sem_t *sem);
```

- To release or signal a semaphore

```
int sem_post(sem_t *sem)
```

- To initialize a semaphore

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

- To destroy a semaphore

```
sem_destoy(sem_t *sem);
```

❑ POSIX semaphores--Producer/Consumer (bounded-buffer) problem

```c
#include <stdlib.h>
#include <pthread.h>
#include <stdio.h>
#include <semaphore.h>
#include <unistd.h>
#define NUM 5 //queue length

int queue[NUM];

sem_t full, empty, mutex;

void *producer(void *arg)
{
    int p = 0;
    while(1){
        sem_wait(&empty);
        sem_wait(&mutex);
        queue[p] = rand() % 1000 + 1;
        printf("produce %d\n", queue[p]);
        p = (p + 1) % NUM;
        sem_post(&mutex);
        sem_post(&full);
        sleep(rand()%5);
    }
}
```

```c
void *consumer(void *arg)
{
    int c = 0, i;
    while(1) {
        sem_wait(&full);
        sem_wait(&mutex);
        for(i=0; i < NUM; i++) {
            printf("%d ", queue[i]);
        }
        putchar('\n');

        printf("consume %d\n", queue[c]);
        queue[c] = 0;
        sem_post(&mutex);
        sem_post(&empty);
        c = (c+1)%NUM;
        sleep(rand()%5);
    }
}
```

```c
int main()
{
    pthread_t pid, cid;

    sem_init(&empty, 0, NUM);
    sem_init(&full, 0, 0);
    sem_init(&mutex, 0, 1);
    pthread_create(&pid, NULL, producer, NULL);
    pthread_create(&cid, NULL, consumer, NULL);
    pthread_join(pid, NULL);
    pthread_join(cid, NULL);
    sem_destroy(&full);
    sem_destroy(&empty);
    return 0;
}
```

- POSIX semaphores--Producer/Consumer (bounded-buffer) problem

```
shaun@shaun-VirtualBox:~/OS_code/Semaphore$ ./prod_cons
produce 384
384 0 0 0 0
consume 384
produce 916
0 916 0 0 0
consume 916
produce 387
0 0 387 0 0
consume 387
produce 422
0 0 0 422 0
consume 422
produce 691
0 0 0 0 691
consume 691
produce 927
produce 427
927 427 0 0 0
consume 927
produce 212
0 427 212 0 0
consume 427
0 0 212 0 0
consume 212
^C
```