ECE437/CS481

M04B: PROCESS COORDINATION SEMAPHORES

CHAPTER 5.6

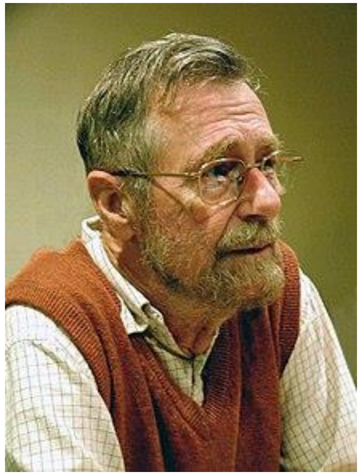
Xiang Sun

The University of New Mexico

Semaphore as Abstraction

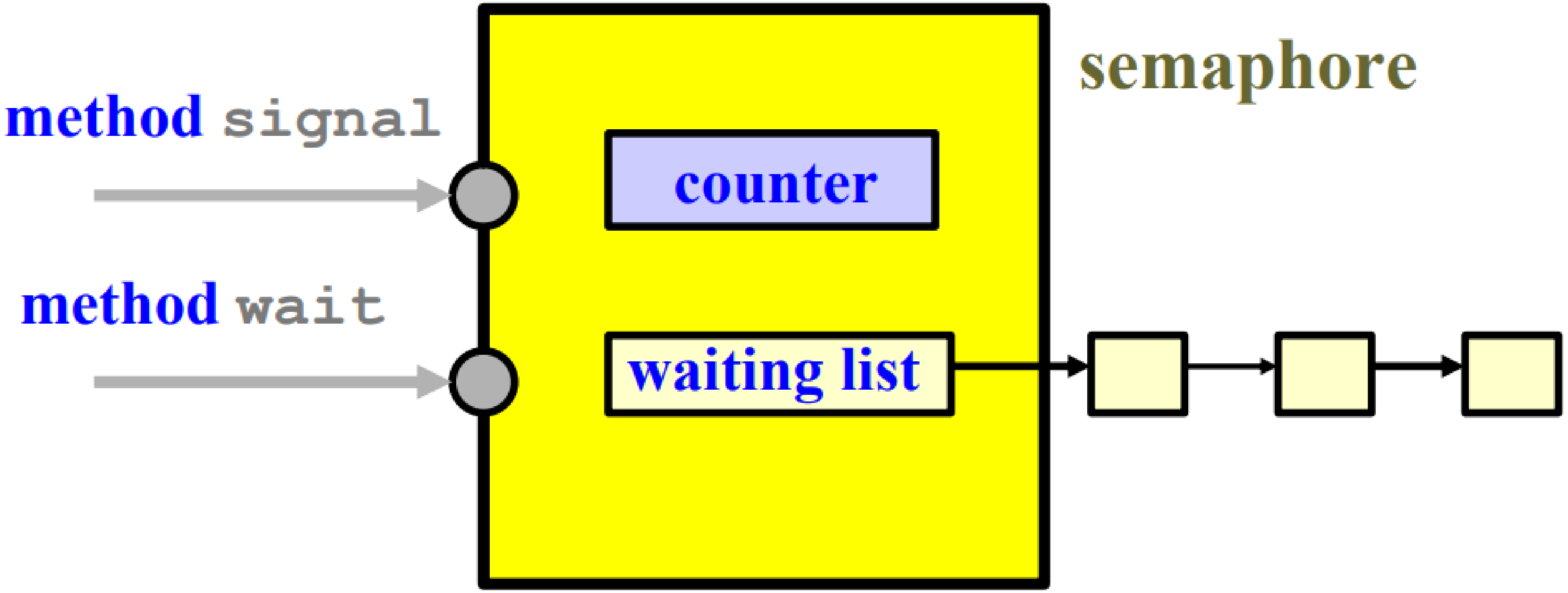
- **Motivation**
  - Using different schemes to handle CS in a low layer.
  - Hide the implementation details from users---user friendly
  - General solutions should be machine independent and relatively easy to use

- **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

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

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

Semaphore as Abstraction

signal() method in semaphore

- Increment semaphore (s) by 1.

```
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;
```

Process 1	Process 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?

❑ Enforce process synchronization

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

Process 0	Process 1
A01;	A11;
signal(S);	A12;
A02;	wait(S)
A03;	A13;

Process 0	Process 1
A_{01}	A_{11}
A_{02}	A_{12}
A_{03}	A_{13}

❑ Notification

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

Process 1

```
while (1) {
  //do something
  wait(S1);
  printf("1");
  signal(S2);
  //do something
}
```

notify

Process 2

```
while (1) {
  //do something
  wait(S2);
  printf("2");
  signal(S1);
  //do something
}
```

notify

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

Applications of Semaphore

- Counting Semaphore—represent a resource with multiple instances

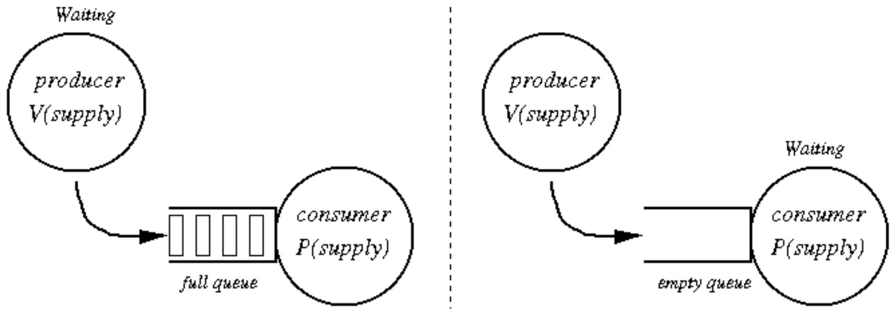
```
semaphore S = 3;

Process 1                     Process 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 data item into the buffer if it's full and that the consumer won't try to remove data item from an empty buffer.
  - If the producer is adding data item into the buffer, the consumer won't allow to remove data item from the buffer.
  - If the consumer is remove data item from the buffer, the producer won't allow to add 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 number of data items in the buffer, and the other is to keep track of number of unoccupied slots in the buffer

```
Semaphores: mutex, empty, full;

mutex = 1; // for mutual exclusion
empty = N; // number empty buf entries
full = 0; // number full buf 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);
	}	}

Applications of Semaphore

❑ 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++;
}
```

> signal() and wait() have critical sections, i.e., 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.

❑ 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., var x)
- Two possible cases when instruction Test_and_Set is applied to var x

❑ 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 (s <= 0)
   {};//do no-operation, just wait in the list;
 while (Test_and_Set(lock0))
   {}; //do nothing, just wait;
  s--;
  lock0=false; //set lock free
}
```

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

Implementation of Semaphores

❑ Comp_and_Swap Instruction

It's logic function

```
Comp_and_Swap (int *x,
               int old, int new) {
    int temp = *x;
    if (temp == old)
        *x = new;
    return temp;
}
```

Old value of x	New value of x	return value
== old	new	old
!= old	no change	old value of x

- Two actions: compare and swap
- Two possible cases when instruction Comp_and_Swap is applied to var x

- Implementing Semaphores by Comp_and_Swap (int *x, int old, int new) Instruction
  - If old=0, new=1

Old value of x	New value of x	Return value	
1	1	1	Other process is executing wait()
0	1	0	No process is executing wait()

```
Bool lock0=0; //lock free

wait(s){
while (s <= 0)
   {};//do no-operation, just wait in the list;
while (Comp_and_Swap(lock0,0,1))
   {}; //do nothing, just wait;
  s--;
  lock0=0; //set lock free
}
```

```
signal(s){
    while (Comp_and_Swap(lock0,0,1))
     {}; //do nothing, just wait;
     s++;
     lock0=0; //set lock free
}
```

Busy-waiting and Non-busy-waiting Semaphore

❑ Busy-waiting Semaphore

- Recall that

```
Bool lock0=false; //lock free
wait(s){
while (s <= 0)
   {};//do no-operation, just wait in the list;
while (Test_and_Set(lock0))
   {}; //do nothing, just wait;
   s--;
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 (i.e., another process is accessing the CS), the process should check the value of s and wait periodically, thus wasting processor time.
- Process starvation is possible when a process leaves a critical section and more than one process is waiting
- Priority inversion
  - ✓ if a low priority process, P_A, is within the CS and another higher priority process, P_B, come and try 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 ---- deadlock.

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;
 s--;
 if(s <= 0){
       add this process into queue q;
       block();}
  lock0=false; //set lock free
}
```

```
signal(s){
     while (Test_and_Set(lock0))
      {}; //do nothing, just wait;
      s++;
      if (s<=0){
         remove a process P from queue q;
         wakeup(P);
      lock0=false; //set lock free
}
```

POSIX semaphores

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

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

- To destroy a semaphore

```
sem_destoy(sem_t *mutex);
```

POSIX semaphores

❑ 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

- 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
```