Operating Systems

Semaphores, Classical problems

Debadatta Mishra, CSE, IITK

Why mutual exclusion is not sufficient?

- Locking techniques allows exactly one thread to access the critical section
- Consider a scenario when at most K concurrent accesses are alright
- Example:
    - A finite array of size N accessed from a set of producer and consumer threads
    - At most N concurrent producers are allowed if array is empty
    - At most N concurrent consumers are allowed if array is full
- Using mutual exclusion techniques
    - Any access to array must take a lock, one thread allowed at a time, not very efficient
    - Does read-write locks help?

Semaphores

```
typedef struct semaphore{
                    int value;
                    spinlock *LOCK;
                    Queue *waitQ;
                    ....
                }sem_t;   // Example
int wait (sem_t *s)                 int post (sem_t *s)
{                                   {
   s->value--;                          s->value++;
   Wait if s->value <= 0                wakeup one if one or more are waiting
}                                   }
```

- Generally, semaphores are initialized to a positive integer K
- Two operations: wait and post (other notations {wait, signal}, {P,V}, {down, up})

Unix semaphores

```
#include <semaphore.h>

main( ){
    sem_t  s;
    int K = 5;
    sem_init(&s, 0, K);
    sem_wait(&s);
    sem_post(&s);
}
```

- Can be used to synchronize threads of a single process if second argument is 0
- K is the value of the semaphore
- If K == 1, semaphore is called binary semaphores (mutex) as it can implement mutual exclusion
- lock → sem_wait(&s)
- unlock → sem_post(&s)

Semaphore implementation (buggy #1)

```
wait(sem_t *s)
{
   while(s->value <= 0);
   s->value--;
}
post(sem_t *s)
{
   s->value++;
}
```

- What is wrong with this implementation?
- Atomic decrement and increment, will it help?
- Disable preemption?

Semaphore implementation (inefficient)

```
wait(sem_t *s)
{
  lock(s->LOCK);
  while(s->value <= 0);
   s->value--;
  unlock(s->LOCK);
}
post(sem_t *s)
{
  atomic_inc(s->value);
}
```

- Busy wait, wastage of CPU cycles
- Why not put the context to sleep?

Semaphore implementation (buggy#2)

```
wait (sem_t *s)
{
  s->value--;
  if (s->value < 0){
     insert_tail(s->waitQ, self);
     self->state = WAITING;
     schedule( );
  }
}
```

```
post (sem_t *s)
{
     s->value++;
     if(s->value <= 0){
         p = remove_head(s->waitQ);
         p->state = READY;
     }
 }
```

- Why is this not correct?
- Mutual exclusion is needed, but where exactly?

Semaphore implementation (buggy#3)

```
wait (sem_t *s)
{
  lock(s->LOCK);
  s->value--;
  if (s->value < 0){
     insert_tail(s->waitQ, self);
     self->state = WAITING;
     schedule( );
  }
 unlock(s->LOCK);
}
```

```
post (sem_t *s)
{
  lock(s->LOCK);
  s->value++;
  if (s->value <= 0){
     p = remove_head(s->waitQ);
     p->state = READY;
   }
   unlock(s->LOCK);
}
```

- Why is this not correct?

Semaphore implementation (buggy#4)

```
wait (sem_t *s)
{
  lock(s->LOCK);
  s->value--;
  if (s->value < 0){
     insert_tail(s->waitQ, self);
     self->state = WAITING;
     unlock(s->LOCK);
     schedule( );
  }
 unlock(s->LOCK);
}
```

```
post (sem_t *s)
{
  lock(s->LOCK);
  s->value++;
  if (s->value <= 0){
     p = remove_head(s->waitQ);
     p->state = READY;
   }
   unlock(s->LOCK);
}
```

- Failure Scenario?

Semaphore implementation

```
wait (sem_t *s)
{
  lock(s->LOCK);
  s->value--;
  if (s->value < 0){
     insert_tail(s->waitQ, self);
     self->state = WAITING;
     unlock(s->LOCK);
     schedule( );
     return;
  }
 unlock(s->LOCK);
}
```

```
post (sem_t *s)
{
  lock(s->LOCK);
  s->value++;
  if (s->value <= 0){
     p = remove_head(s->waitQ);
     p->state = READY;
   }
   unlock(s->LOCK);
}
```

- Any assumptions for correctness?

Semaphore usage example: wait for child

```
sem_init(s, 0, 0);

child( ){
    ..........
    post(s);
}

parent( ){
        create_child( );
        wait(s);
}
```

- Semaphore value initialized to zero
- If parent gets scheduled after the child creation, waits till child finished
- If child gets scheduled before parent, parent does not wait

Semaphore usage example: ordering

```
A=0; B=0;

Th0 {
    A = 1;
    printf("%d\n", B);
}
Th1 {
        B=1;
        printf("%d\n", A);
}
```

- What are the possible outputs?
- How to ensure output (A = 1, B = 1)?

Semaphore usage example: ordering

```
// s1->value = 0
A=0; B=0;

Th0{
    A = 1;
    wait(s1);
    printf("%d\n", B);
}
Th1{
        B=1;
        post(s1);
        printf("%d\n", A);
}
```

- What are the possible outputs?
- How to ensure output (A = 1, B = 1)?

Ordering with two semaphores

```
// s1->value = 0, s2->value = 0
A=0; B=0;

Th0{
    A = 1;
    post(s1);
    wait(s2);
    printf("%d\n", B);
}
```

```
// s1->value = 0, s2->value = 0
A=0; B=0;

Th1{
        B=1;
        wait(s1);
         post(s2);
         printf("%d\n", A);
}
```

Producer-consumer problem

- A buffer of size N, one or more producers and consumers
- Producer adds an element to the buffer
- Consumer extracts an element from the buffer
- Example: A multithreaded web server, network device queue etc.
- Solution using semaphores?

Producer-consumer problem (buggy #1)

```
item A[N], sem_t empty = {N}, used = {0}, pctr = 0, cctr = 0;
```

```
produce(item x)
{
   wait(empty);
   A[pctr] = x;
   pctr = (pctr + 1) % N;
   post(used);
}
```

```
consume()
{
   item x;
   wait(used);
   x = A[cctr];
   cctr = (cctr + 1) % N;
   post(empty);
   return x;
}
```

- Both produce and consume can be called concurrently
- What is the problem?

Producer-consumer problem (buggy #2)

```
item A[N], sem_t empty = {N}, used = {0}, pctr = 0, cctr = 0, mutex M;
```

```
produce(item x)
{
   lock(M);
   wait(empty);
   A[pctr] = x;
   pctr = (pctr + 1) % N;
   post(used);
   unlock(M);
}
```

```
consume()
{
   item x;
   lock(M)
   wait(used);
   x = A[cctr];
   cctr = (cctr + 1) % N;
   post(empty);
   unlock(M);
   return x;
}
```

- What is the problem?

Producer-consumer problem

```
item A[N], sem_t empty = {N}, used = {0}, pctr = 0, cctr = 0, mutex M;
```

```
produce(item x)
{
    wait(empty);
    lock(M);
   A[pctr] = x;
   pctr = (pctr + 1) % N;
   unlock(M);
   post(used);
}
```

```
consume()
{
    item x;
    wait(used);
    lock(M)
    x = A[cctr];
    cctr = (cctr + 1) % N;
    unlock(M);
    post(empty);
    return x;
}
```

- What if separate mutex is used for producer and consumer?

Producer-consumer problem (buggy?)

```
item A[N], sem_t empty = {N}, used = {0}, pctr = 0, cctr = 0, mutex P, C;
```

```
produce(item x)
{
    wait(empty);
    lock(P);
   A[pctr] = x;
   pctr = (pctr + 1) % N;
   unlock(P);
   post(used);
}
```

```
consume()
{
   item x;
   wait(used);
   lock(C)
   x = A[cctr];
   cctr = (cctr + 1) % N;
   unlock(C);
   post(empty);
   return x;
}
```

- Is there any issue?

Producer-consumer variant

```
produce(item x)
{
    wait(empty);
    lock(P);
    copy(A[pctr], x);
    pctr = (pctr + 1) % N;
    unlock(P);
    post(used);
}
```

```
consume()
{
    item x;
    wait(used);
    lock(C);
    copy( x, A[cctr]);
    cctr = (cctr + 1) % N;
    unlock(C);
    post(empty);
    return x;
}
```

- Copy inside the mutex not very efficient, can it be moved outside?

Producer-consumer variant (buggy?)

```
produce(item x)
{
    int lpos;
    wait(empty);
    lock(P);
    lpos = pctr;
    pctr = (pctr + 1) % N;
    unlock(P);
    copy(A[lpos], x);
    post(used);
}
```

```
consume()
{
    item x;  int lpos;
    wait(used);
    lock(C);
    lpos = cctr;
    cctr = (cctr + 1) % N;
    unlock(C);
    copy( x, A[lpos]);
    post(empty);
}
```

- Does this code work?

Producer-consumer variant

```
produce(item x)
{
    .........................
    pctr = (pctr + 1) % N;
    lock(A[lpos].lock);
    unlock(P);
    copy(A[lpos], x);
    unlock(A[lpos].lock);
    post(used);
}
```

```
consume()
{
    .....................
    cctr = (cctr + 1) % N;
    lock(A[lpos].lock);
    unlock(C);
    post(empty);
    copy( x, A[lpos]);
    unlock(A[lpos].lock);
}
```

- Why does this code work?
- Any benefits?

Condition variables

- *pthread_cond_wait (cond, mutex)*: Atomically releases the mutex and waits on a condition variable. Imp: *Does not perform any condition check.* Resumes execution holding the lock when pthread_cond_signal( ) is invoked

- *pthread_cond_signal (cond)*: Wakes up a waiting thread on condition *cond*, Ideally called holding the mutex.

Example

```
cond_t C; mutex M;   BOOL condition;
```

```
Th1( )
{
  while(1){
      ................
       lock(M);
       while(condition != true)
             cond_wait (C, M);
      unlock(M);
     ................
}
```

```
Th2( )
{
  while(1){
      ................
      lock(M);
      condition = true;
      cond_signal(C);
      unlock(M);
      ....................
}
```

- What if Th2( ) does not acquire lock?
- What if cond_signal( ) called after unlock(M)?

An example implementation (src: pthread man page)

```
cond_t {  mutex CM, int val, waitQ};
```

```
cond_wait(cond_t C, mutex M)
{
   int value = C->value;
   unlock(M);
   lock(C->CM);
   If (value == C->value){
       addQ(C->waitQ, self);
       unlock(C->CM);
       schedule( );
   }else   {unlock(C->CM);}
   lock(M);
 }
```

```
cond_signal(cond_t C)
{
     lock(C->CM);
     C->value++;
     if(!empty(C->waitQ)){
         p=pickQ(C->waitQ);
         p->state = running;
      }
     unlock(C->CM);
  }
```

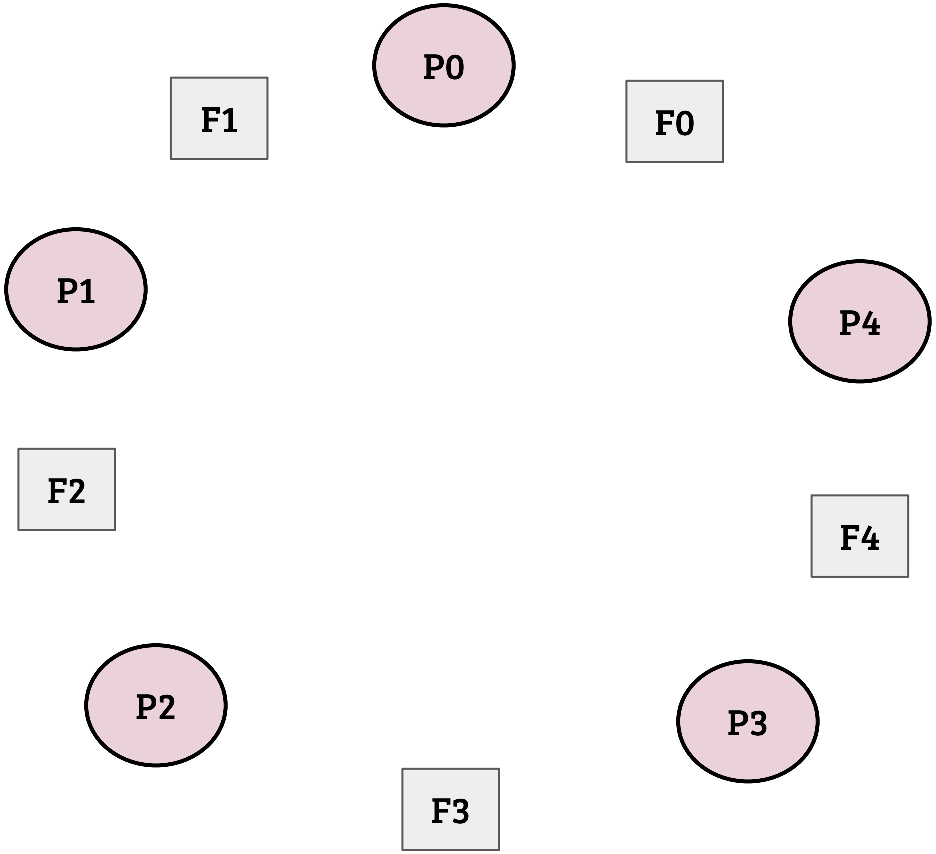
Deadlock: example

```
ACCOUNT {
                long ID;
                mutex M;
                long balance;
                . . .
           };
```

```
Txn_transfer(ACCOUNT *S, ACCOUNT *D,  $)
{
    lock (S->M);
    lock (D->M);
    ....................
    ....................
    unlock (D->M);
    unlock(S->M);
}
```

- Where is the deadlock?
- Example: {T1: Txn_transfer(&CSE, & IITK);}, {T2: Txn_transfer(&IITK, &CSE);
- Solution?

Dining philosophers




```
P(ID )
{
   while (1) {
      think( );
      getforks( );
      eat( );
      putforks( );
   }
}
```

Conditions for deadlock

- Mutual exclusion → Exclusive access to the resource
- Hold-and-wait → Hold one lock and wait for other
- No resource preemption → Locks can not be forcibly removed from threads holding them
- Circular wait → A cycle of threads requesting locks held by others. Specifically, a cycle in the directed graph $G(V, E)$ where V is the set of processes, $(v1, v2) \in E$ if $v1$ is waiting for a lock held by $v2$

All of the above conditions should be satisfied for a deadlock to occur