CS330: Operating Systems

Semaphore, Classical problems

Semaphores

- Mutual exclusion techniques allows exactly one thread to access the critical section which can be restrictive
- Consider a scenario when a finite array of size N is accessed from a set of producer and consumer threads. In this case,
  - At most N concurrent producers are allowed if array is empty
  - At most N concurrent consumers are allowed if array is full
  - If we use mutual exclusion techniques, only one producer or consumer is allowed at any point of time

Operations on semaphore

```
struct semaphore{
                 int value;
                 spinlock_t *lock;
                 queue *waitQ;
               }sem_t;

// Operations

sem_init(sem_t *sem, int init_value);
sem_wait(sem_t *sem);
sem_post(sem_t *sem);
```

- Semaphores can be initialized by passing an initial value
- *sem_wait* waits (if required) till the value becomes +ve and returns after decrementing the value
- *sem_post* increments the value and wakes up a waiting context
- Other notations: P-V, down-up, wait-signal

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 in a multi-threaded process or across multiple processes
- If second argument is 0, the semaphore can be used from multiple threads
- Semaphores initialized with value = 1 (third argument) is called a binary semaphore and can be used to implement *blocking(waiting) locks*
- Initialize: sem_init(s, 0, 1)
  lock:sem_wait(s), unlock: sem_post(s)

Semaphore usage example: wait for child

```
child( ){
    ...
    sem_post(s);
    exit(0);
}
int main (void ){
        sem_init(s, 0);
        if(fork( ) == 0)
            child( );
        sem_wait(s);
}
```

- Assume that the semaphore is accessible from multiple processes, value initialized to zero
- If parent is scheduled after the child creation, it waits till child finishes
- If child is scheduled and exits before parent, parent does not wait for the semaphore

Semaphore usage example: ordering

```
A=0; B=0;

Thread-0 {
    A = 1;
    printf("B = %d\n", B);
}

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

- What are the possible outputs?

Semaphore usage example: ordering

```
A=0; B=0;

Thread-0 {
    A = 1;
    printf("B = %d\n", B);
}

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

- What are the possible outputs?
- (A = 1, B= 1), (A = 1, B = 0), (A = 0, B=1)
- How to guarantee A = 1, B= 1?

Semaphore usage example: ordering

```
sem_init(s1, 0);
A=0; B=0;
Thread - 0 {
    A = 1;
    sem_wait(s1);
    printf("B = %d\n", B);
}
Thread - 1 {
        B=1;
        sem_post(s1);
        printf("A = %d\n", A);
}
```

- What are the possible outputs?

Semaphore usage example: ordering

```
sem_init(s1, 0);
A=0; B=0;
Thread - 0 {
    A = 1;
    sem_wait(s1);
    printf("B = %d\n", B);
}
Thread - 1 {
        B=1;
        sem_post(s1);
        printf("A = %d\n", A);
}
```

- What are the possible outputs?
- (A = 1, B = 1), (A=0, B=1)
- How to guarantee A = 1, B= 1?

Ordering with two semaphores

```
sem_init(s1, 0);
sem_init(s2, 0)
A=0; B=0;
```

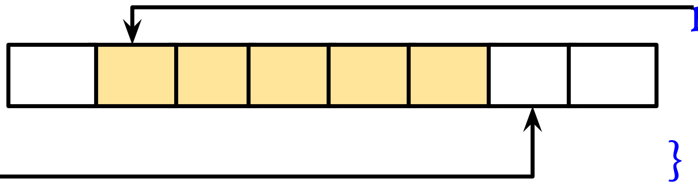
- Waiting for each other guarantees desired output

```
Thread - 0
{
    A = 1;
    sem_post(s1);
    sem_wait(s2);
    printf("%d\n", B);
}
```

```
Thread - 1
{
    B=1;
    sem_wait(s1);
    sem_post(s2);
    printf("%d\n", A);
}
```

Producer-consumer problem

```
DoProducerWork( ){
 while(1){
   item_t item = prod_p( );
   produce(item);
 }
}
```



```
DoConsumerWork( ){
 while(1){
   item_t item = consume( );
   cons_p(item);
 }
}
```

- A buffer of size N, one or more producers and consumers
- Producer produces an element into the buffer (after processing)
- Consumer extracts an element from the buffer and processes it
- Example: A multithreaded web server, network protocol layers etc.
- How to solve this problem using semaphores?

Buggy #1

```
item_t A[n], pctr=0, cctr = 0;
sem_t empty = sem_init(n), used = sem_init(0);
```

```
produce(item_t item){
    sem_wait(&empty);
    A[pctr] = item;
    pctr = (pctr + 1) % n;
    sem_post(&used);
}
```

```
item_t consume( ) {
    sem_wait(&used);
    item_t item = A[cctr];
    cctr = (cctr + 1) % n;
    sem_post(&empty);
    return item;
}
```

- This solution does not work. What is the issue?

Buggy #1

```
item_t A[n], pctr=0, cctr = 0;
sem_t empty = sem_init(n), used = sem_init(0);
```

```
produce(item_t item){
    sem_wait(&empty);
    A[pctr] = item;
    pctr = (pctr + 1) % n;
    sem_post(&used);
}
```

```
item_t consume( ) {
    sem_wait(&used);
    item_t item = A[cctr];;
    cctr = (cctr + 1) % n;
    sem_post(&empty);
    return item;
}
```

- This solution does not work. What is the issue?
- The counters (pctr and cctr) are not protected, can cause race conditions

Buggy #2

```
item_t A[n], pctr=0, cctr = 0;  lock_t *L = init_lock( );
sem_t empty = sem_init(n), used = sem_init(0);
```

```
produce(item_t item){
    lock(L);   sem_wait(&empty);
    A[pctr] = item;
    pctr = (pctr + 1) % n;
    sem_post(&used);  unlock(L);
}
```

```
item_t consume( ) {
    lock(L);  sem_wait(&used);
    item_t item = A[cctr];;
    cctr = (cctr + 1) % n;
    sem_post(&empty); unlock(L);
    return item;
}
```

- What is the problem?

Buggy #2

```
item_t A[n], pctr=0, cctr = 0;  lock_t *L = init_lock( );
sem_t empty = sem_init(n), used = sem_init(0);
```

```
produce(item_t item){
    lock(L);   sem_wait(&empty);
    A[pctr] = item;
    pctr = (pctr + 1) % n;
    sem_post(&used);  unlock(L);
}
```

```
item_t consume( ) {
    lock(L);  sem_wait(&used);
    item_t  item = A[cctr];;
    cctr = (cctr + 1) % n;
    sem_post(&empty); unlock(L);
    return item;
}
```

- What is the problem?
- Consider empty = 0 and producer has taken lock before the consumer. This results in a deadlock, consumer waits for L and producer for empty

A working solution

```
item_t A[n], pctr=0, cctr = 0;  lock_t *L = init_lock( );
sem_t empty = sem_init(n), used = sem_init(0);
```

```
produce(item_t item){
    sem_wait(&empty); lock(L);
    A[pctr] = item;
    pctr = (pctr + 1) % n;
    unlock(L); sem_post(&used);
}
```

```
item_t consume( ) {
    sem_wait(&used); lock(L)
    item_t item = A[cctr];
    cctr = (cctr + 1) % n;
    unlock(L); sem_post(&empty);
    return item;
}
```

- The solution is deadlock free and ensures correct synchronization, but very much serialized (inside produce and consume)
- What if we use separate locks for producer and consumer?

Solution with separate mutexes

```
item_t A[n], pctr=0, cctr = 0;  lock_t *P = init_lock( ), *C=init_lock( );
sem_t empty = sem_init(n), used = sem_init(0);
```

```
produce(item_t item){
    sem_wait(&empty); lock(P);
    A[pctr] = item;
    pctr = (pctr + 1) % n;
    unlock(P); sem_post(&used);
}
```

```
item_t consume( ) {
    sem_wait(&used); lock(C)
    item_t item = A[cctr];
    cctr = (cctr + 1) % n;
    unlock(C); sem_post(&empty);
    return item;
}
```

- Does this solution work?
- Homework: Assume that item is a large object and copy of item takes long time. How can we perform the copy operation without holding the lock?