Operating System (OS) CS232

Concurrency: Semaphores

Dr. Muhammad Mobeen Movania
Dr. Muhammad Saeed

Outlines

- Semaphores in programming?
- Semaphores used for
  - mutual exclusion (as mutex locks)
  - Synchronization (as condition variables)
- Examples
  - Producer/Consumer Problem
  - Reader/Write Problem
  - Dining Philosopher Problem
- Summary

Semaphores

- Special synchronization construct
- Can be used as a lock or condition variable
- Provides two operations
  - `sem_wait()`
  - `sem_post()`

Semaphores – operations

```
int sem_wait(sem_t *s) {
    decrement the value of semaphore s by one
    wait if value of semaphore s is negative
}

int sem_post(sem_t *s) {
    increment the value of semaphore s by one
    if there are one or more threads waiting, wake one
}
```

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1);
```

Semaphore – accessing critical section

```
sem_t m;
sem_init(&m, 0, X); // initialize semaphore to X;

sem_wait(&m);
// critical section here
sem_post(&m);
```

- Initial value of X ??
- with X==1, this semaphore will behave as a lock!

Semaphore

- Named (files stored at /dev/shm/)
  - sem_open((const char *name, int oflag, mode_t mode, unsigned int **value**)
  - sem_close() //**value → 0,1,n**
  - sem_unlink()
- Unnamed (memory)
  - sem_init(sem_t *sem, int **pshared**, unsigned int **value**)
  - sem_destory() **pshared =0 (threads only) non-zero (multiple processes)**

- **sem_wait() / sem_trywait()**
- **sem_post()**
- **sem_getvalue()**

Mutual Exclusion using Named Semaphore

```
long balance = 0;               //  (Debit/Credit on balance)
sem_t *mutex;
```

```
int main(){
char* name = "/b_sem1";    //file at folder …  /dev/shm/sem.b_sem1
mutex = sem_open(name, O_CREAT, 0666, 1); //1-mutual exclusion,0-sync, n-counting
pthread_t t1, t2;   pthread_create(&t1, NULL, credit,(void *)100);  pthread_create(&t2, NULL, debit,50);
pthread_join(t1, NULL);  pthread_join(t2, NULL);
sem_close(mutex);
sem_unlink(name);
printf("Value of balance is :%ld\n", balance);
return 0;
}
```

```
void * credit(void * val){
sem_wait(mutex);
   balance = balance + val; //typecasting
 sem_post(mutex);
pthread_exit(NULL);
}
```

```
void * debit(void * val){
sem_wait(mutex);
   balance = balance - val; //typecasting
 sem_post(mutex);
pthread_exit(NULL);
}
```

Mutual Exclusion using Un-Named Semaphore

```
long balance = 0;            //  (Debit/Credit on balance)
sem_t mutex;
```

```
int main(){
//char* name = "/b_sem1";    mutex = sem_open(name, O_CREAT, 0666, 1);
sem_init(&mutex, 0, 1); // 0-threads, otherwise process ; 1-mutual exclusion,0-sync, n-counting
pthread_t t1, t2;   pthread_create(&t1, NULL, credit,(void *)100);  pthread_create(&t2, NULL, debit,50);
pthread_join(t1, NULL);  pthread_join(t2, NULL);
//sem_close(mutex);    sem_unlink(name);
sem_destroy(&mutex);
printf("Value of balance is :%ld\n", balance);
return 0;
}
```

```
void * credit(void * val){
sem_wait(&mutex);
   balance = balance + val; //typecasting
 sem_post(&mutex);
pthread_exit(NULL);
}
```

```
void * debit(void * val){
sem_wait(&mutex);
   balance = balance - val; //typecasting
 sem_post(&mutex);
pthread_exit(NULL);
}
```

Synchronization using Semaphores

```c
void * order_1(void *);
void * order_2(void *);
void * order_3(void *);
sem_t *semA, *semB;
```

```c
int main() {
    char* sem1 = "/sem1";
    char* sem2 = "/sem2";
    semA = sem_open(sem1, O_CREAT, 0666, 0);
    semB = sem_open(sem2, O_CREAT, 0666, 0);
    pthread_t t1, t2,t3;
    pthread_create(&t1, NULL, order_1, NULL)
    pthread_create(&t2, NULL, order_2, NULL)
    pthread_create(&t3, NULL, order_3, NULL)
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);

    sem_close(semA);
    sem_close(semB);
    sem_unlink(sem1);
    sem_unlink(sem2);
    printf("\n");
   return 0;
}
```

```c
void * order_3(void * arg){
    sem_wait(semB);
    fprintf(stderr, "must be done 3rd\n");
}
void * order_2(void * arg){
    sem_wait(semA);
    fprintf(stderr, "must be done 2nd\n");
    sem_post(semB);
}
void * order_1(void * arg){
    fprintf(stderr,"must be done 1st\n ");
   sem_post(semA);
}
```

```
must be done 1st
 must be done 2nd
must be done 3rd
```

More Examples

- As Reading Assignment
  - Producer/Consumer Problem (Already Discussed)
  - Reader/Write Problem
  - Dining Philosopher Problem

Semaphore – Producer-consumer problem

```
int buffer[MAX];
int fill = 0;
int use  = 0;

void put(int value) {
    buffer[fill] = value;     //
    fill = (fill + 1) % MAX;  //
}

int get() {
    int tmp = buffer[use];    //
    use = (use + 1) % MAX;    //
    return tmp;
}
```

```
int main(int argc, char *argv[])
    // ...
    sem_init(&empty, 0, MAX); //
    sem_init(&full, 0, 0);    //
    // ...
}
```

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);         // Line P1
        put(i);                   // Line P2
        sem_post(&full);          // Line P3
    }
}

void *consumer(void *arg) {
    int i, tmp = 0;
    while (tmp != -1) {
        sem_wait(&full);          // Line C1
        tmp = get();              // Line C2
        sem_post(&empty);         // Line C3
        printf("%d\n", tmp);
    }
}
```

Semaphores – Producer Consumer with mutual exclusion

```
sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++)
        sem_wait(&mutex);
        sem_wait(&empty);
        put(i);
        sem_post(&full);
        sem_post(&mutex);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++)
        sem_wait(&mutex);
        sem_wait(&full);
        int tmp = get();
        sem_post(&empty);
        sem_post(&mutex);
        printf("%d\n", tmp);
    }
}
```

```
int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX are empty
    sem_init(&full, 0, 0);    // 0 are full
    sem_init(&mutex, 0, 1);   // mutex=1: a lock
    // ...
}
```

Semaphore – Producer consumer (correct)

```
sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);        /
        sem_wait(&mutex);        /
        put(i);                  /
        sem_post(&mutex);        /
        sem_post(&full);         /
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);         /
        sem_wait(&mutex);        /
        int tmp = get();         /
        sem_post(&mutex);        /
        sem_post(&empty);        /
        printf("%d\n", tmp);
    }
}
```

```
int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX are empty
    sem_init(&full, 0, 0);    // 0 are full
    sem_init(&mutex, 0, 1);   // mutex=1: a lock
    // ...
}
```

Readers-Writers Problem

- Let’s say a concurrent operations happen on a global linked-list
- Of two kinds:
  - Write(), modify the list
  - Read(), simply read list data

- Multiple threads
- Can we have multiple reads in parallel? How?

```
typedef struct _rwlock_t {
  sem_t lock;       // binary sema
  sem_t writelock; // allow ONE w
  int   readers;   // #readers in
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
  rw->readers = 0;
  sem_init(&rw->lock, 0, 1);
  sem_init(&rw->writelock, 0, 1);
}
```

```
void rwlock_acquire_writelock(rwlock_t *rw) {
  sem_wait(&rw->writelock);
}

void rwlock_release_writelock(rwlock_t *rw) {
  sem_post(&rw->writelock);
}
```

```
void rwlock_acquire_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers++;
  if (rw->readers == 1) // first reader gets writelock
    sem_wait(&rw->writelock);
  sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers--;
  if (rw->readers == 0) // last reader lets it go
    sem_post(&rw->writelock);
  sem_post(&rw->lock);
}
```

Dining Philosophers Problem

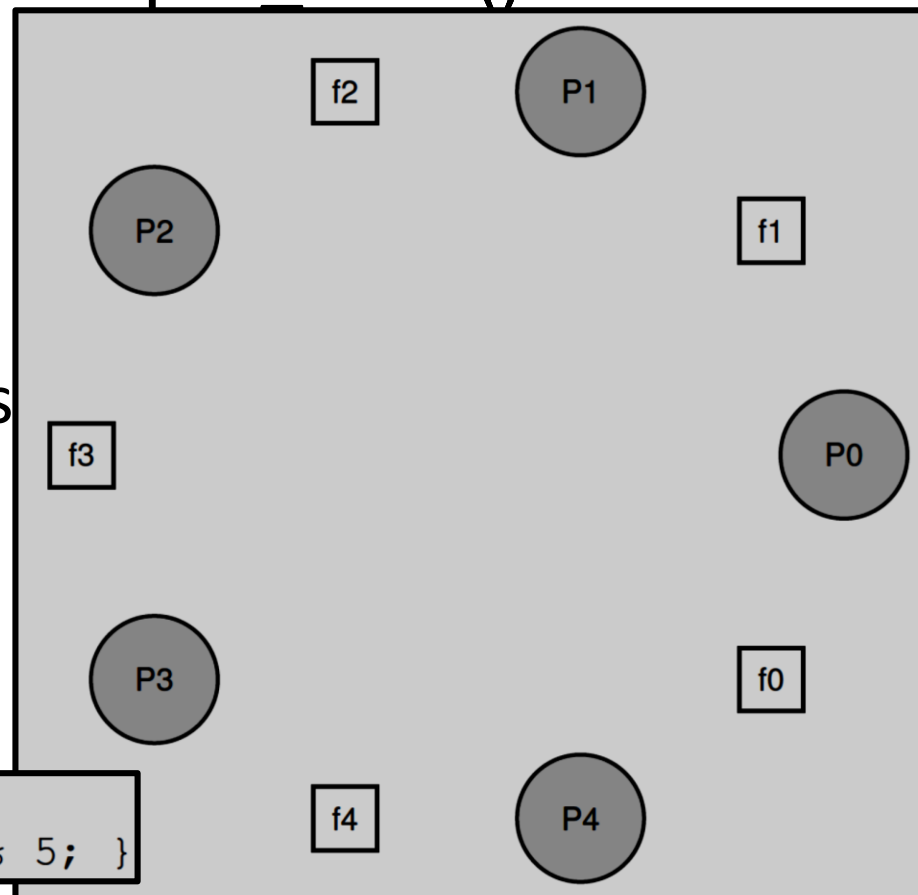
- N philosophers sitting on a table
- N forks b/w them. One b/w each pair.
- A philosopher can do two things:
  - Philosophize (no need of forks)
  - Eat (needs both forks: left and right)

- A philosopher can only do one of the two things at a time

```
while (1) {
    think();
    get_forks(p);
    eat();
    put_forks(p);
}
```

Dining Philosophers Problem … contd.

- Implement get_forks() and put_forks() such that:
  - There's no deadlock
  - No philosopher starves
  - As much concurrency as possible




```
int left(int p)  { return p; }
int right(int p) { return (p + 1) % 5; }
```

Dining Philosophers - implementation

- Five semaphores (one for each fork)
- Initialize each semaphore to 1

```
sem_t forks[5].
```

```
void get_forks(int p) {
    sem_wait(&forks[left(p)]);
    sem_wait(&forks[right(p)]);
}

void put_forks(int p) {
    sem_post(&forks[left(p)]);
    sem_post(&forks[right(p)]);
}
```

Dining Philosophers – breaking deadlock

- Break the dependency!
- Let at least one philosopher acquire forks in different order!

```
void get_forks(int p) {
    if (p == 4) {
        sem_wait(&forks[right(p)]);
        sem_wait(&forks[left(p)]);
    } else {
        sem_wait(&forks[left(p)]);
        sem_wait(&forks[right(p)]);
    }
}
```

Semaphores – an implementation

```c
typedef struct __Zem_t {
    int value;
    pthread_cond_t cond;
    pthread_mutex_t lock;
} Zem_t;

// only one thread can call this
void Zem_init(Zem_t *s, int value) {
    s->value = value;
    Cond_init(&s->cond);
    Mutex_init(&s->lock);
}
```

```c
void Zem_wait(Zem_t *s) {
    Mutex_lock(&s->lock);
    while (s->value <= 0)
        Cond_wait(&s->cond, &s->lock);
    s->value--;
    Mutex_unlock(&s->lock);
}

void Zem_post(Zem_t *s) {
    Mutex_lock(&s->lock);
    s->value++;
    Cond_signal(&s->cond);
    Mutex_unlock(&s->lock);
}
```

Summary

- Semaphores are a simple, powerful and flexible primitive for writing concurrent programs
- Semaphores provide a general solution that can operate as a lock or condition variable