Synchronization Primitives pt. 2(ch. 30+31+32)

Operating Systems
Based on: Three Easy Pieces by Arpaci-Dusseaux

Moshe Sulamy

Tel-Aviv Academic College

Semaphores

- What value should a semaphore be initialized to?
  - General rules: number of resources
  - Lock: 1 $\rightarrow$ can be locked after initialization
  - Ordering: 0 $\rightarrow$ nothing to give away at the start

Producer / Consumer

- How can we implement a **bounded buffer** with semaphores?

```c
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;
}
```

First Attempt

- What is the problem?

```
sem_t empty; // initialized to MAX
sem_t full;  // initialized to 0

void produce(int value) {
    sem_wait(&empty);
    put(value);
    sem_post(&full);
}
int consume() {
    sem_wait(&full);
    int tmp = get();
    sem_post(&empty);
    return tmp;
}
```

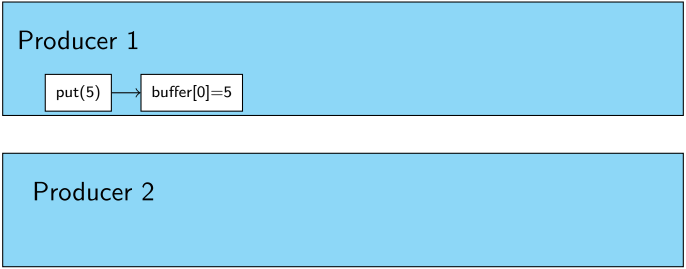
First Attempt

Producer 1

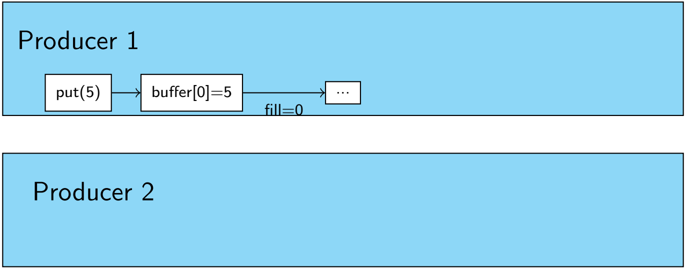
Producer 2

First Attempt

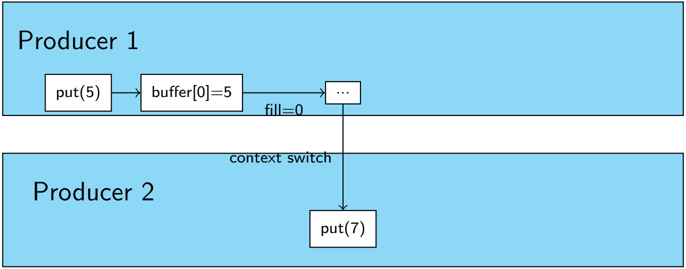
First Attempt



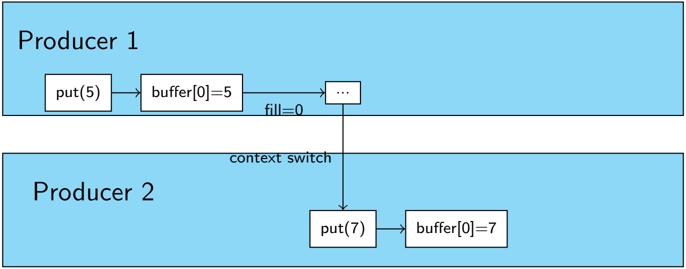
First Attempt



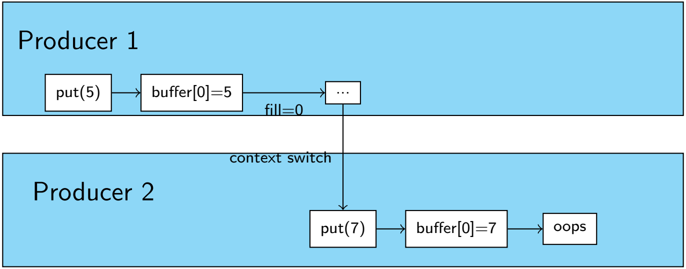
First Attempt



First Attempt



First Attempt



With Mutual Exclusion

- What is the problem?

```
sem_t mutex; // binary semaphore
sem_t empty; // initialized to MAX
sem_t full;  // initialized to 0

void produce(int value) {
    sem_wait(&mutex);
    sem_wait(&empty);
    put(value);
    sem_post(&full);
    sem_post(&mutex);
}
int consume() {
    sem_wait(&mutex);
    sem_wait(&full);
    int tmp = get();
    sem_post(&empty);
    sem_post(&mutex);
    return tmp;
}
```

With Mutual Exclusion

- What is the problem? **deadlock!**
  - Producer waits on empty, holds mutex, consumer can't consume

```
sem_t mutex; // binary semaphore
sem_t empty; // initialized to MAX
sem_t full;  // initialized to 0

void produce(int value) {
    sem_wait(&mutex);
    sem_wait(&empty);
    put(value);
    sem_post(&full);
    sem_post(&mutex);
}
int consume() {
    sem_wait(&mutex);
    sem_wait(&full);
    int tmp = get();
    sem_post(&empty);
    sem_post(&mutex);
    return tmp;
}
```

With Mutual Exclusion

- Solution: use mutex around the critical section

```
sem_t mutex; // binary semaphore
sem_t empty; // initialized to MAX
sem_t full;  // initialized to 0

void produce(int value) {
    sem_wait(&empty);
    sem_wait(&mutex);
    put(value);
    sem_post(&mutex);
    sem_post(&full);
}
int consume() {
    sem_wait(&full);
    sem_wait(&mutex);
    int tmp = get();
    sem_post(&mutex);
    sem_post(&empty);
    return tmp;
}
```

Reader-Writer Locks

- More flexible locking primitive
  - e.g., concurrent operations: inserts and lookups
  - Insert changes state $\rightarrow$ traditional critical section
  - Lookup reads data structure $\rightarrow$ many at once (if no insert)
- **Reader-writer lock**
  - Four operations: acquire/release read/write lock

Reader-Writer Locks

- A **single writer** can acquire the lock
- Once a reader acquires a **read lock**:
  - **More readers** are allowed to acquire the read lock
  - A writer waits until all readers are finished

Reader-Writer Locks

- A **single writer** can acquire the lock
- Once a reader acquires a **read lock**:
  - **More readers** are allowed to acquire the read lock
  - A writer waits until all readers are finished

```
typedef struct _rwlock_t {
    sem_t lock;          // binary semaphore
    sem_t writelock;     // used to allow ONE writer
    int readers;         // count of readers in CS
} rwlock_t;

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

Reader-Writer Locks

```c
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);     // CS for readers
    rw->readers++;
    if (rw->readers == 1)
        sem_wait(&rw->writelock); // first reader grabs writelock
    sem_post(&rw->lock);
}
void rwlock_release_readlock(rwlock_t* rw) {
    sem_wait(&rw->lock);     // CS for readers
    rw->readers--;
    if (rw->readers == 0)
        sem_post(&rw->writelock); // last reader releases writelock
    sem_post(&rw->lock);
}
```

Reader-Writer Locks

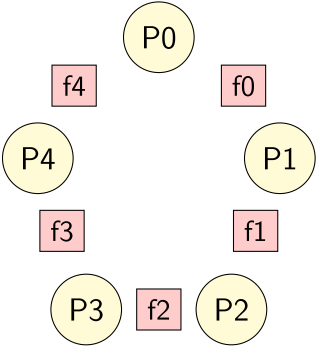
- What is the problem?

Reader-Writer Locks

- What is the problem? **fairness**
  - Easy to **starve** writer
  - How to prevent readers from starving writers?

Dining Philosophers

- Five philosophers around a table
  - Single fork between each pair
  - Philosophers **think** and **eat**
  - Two forks to eat (left and right)



Dining Philosophers

- As code:

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

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

Dining Philosophers

- Use a semaphore for each fork:

```
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)]);
}
```

- The problem?

Dining Philosophers

- Use a semaphore for each fork:

```
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)]);
}
```

- The problem? **deadlock!**
  - Each philosopher grabs fork on their left
  - All waiting for their right

- Solution: break the dependency

```
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)]);
    }
}
```

Thread Throttling

- For example: hundreds of threads work in parallel
- Section of code allocates a lot of memory
  - All threads at the same time $\rightarrow$ exceeds physical memory
  - Machine will start thrashing (swapping to and from the disk)

- Solution?

Thread Throttling

- For example: hundreds of threads work in parallel
- Section of code allocates a lot of memory
  - All threads at the same time $\rightarrow$ exceeds physical memory
  - Machine will start thrashing (swapping to and from the disk)

- Solution? **semaphore**!
  - Initialized to max threads we wish to enter code section
  - Surrounds code section, limits concurrent threads in it

Implementing Semaphores

- Doesn't maintain invariant: negative value $\rightarrow$ # waiting threads
  - Easier, matches the Linux implementation

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

void zem_init(zem_t* s, int value) {
    s->value = value;
    pthread_cond_init(&s->cond);
    pthread_mutex_init(&s->lock);
}
void zem_wait(zem_t* s) {
    pthread_mutex_lock(&s->lock);
    while (s->value <= 0)
        pthread_cond_wait(&s->cond, &s->lock);
    s->value--;
    pthread_mutex_unlock(&s->lock);
}
void zem_post(zem_t* s) {
    pthread_mutex_lock(&s->lock);
    s->value++;
    pthread_cond_signal(&s->cond);
    pthread_mutex_unlock(&s->lock);
}
```

Summary

- **Condition variables**
  - Thread waits until a certain condition
  - `wait()`, `signal()`
  - **Hold lock** while signaling
  - Check value **in a loop**
- **Semaphore**
  - Integer value
  - Decrement on acquire, wait if negative, increment on release
- **Read-write lock**
  - **Single writer** or **multiple readers**
- Dining philosophers
  - Think and eat
- Producer / consumer (bounded buffer)