Chapter 7

Synchronization Examples

Yunmin Go

School of CSEE

HANDONG GLOBAL UNIVERSITY

Agenda

- Semaphores
- Bounded Buffer
- Reader-Writer Locks
- Dining Philosophers

Semaphores

- **Semaphore**: synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
  - Semaphore **`S`** is integer variable
  - Can only be accessed via two atomic operations
  - `wait()` operation

```
wait(S) {
    while (S <= 0);    // waits for the lock
    S--;               // holds the lock
}
```

  - `signal()` operation

```
signal(S) {
    S++;               // release the lock
}
```

Semaphores Usage

- Counting semaphore
  - Integer value **S** can range over an unrestricted range
  - Initialized to # of available resources

- Binary semaphore
  - Integer value can range only between 0 and 1
  - Same as mutex lock

- Can solve various synchronization problems

POSIX Semaphores

- Named semaphores
  - Multiple unrelated processes can easily use a common semaphore
  - Creating and initializing the lock

```
#include <semaphore.h>
sem_t *sem;              // global declaration

// Create the semaphore and initialize it to 1
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

  - Acquiring and releasing the lock

```
sem_wait(sem);   // acquire the semaphore

/* critical section */

sem_post(sem);   // release the semaphore
```

https://man7.org/linux/man-pages/man3/sem_open.3.html
https://man7.org/linux/man-pages/man3/sem_wait.3p.html
https://man7.org/linux/man-pages/man3/sem_post.3.html

POSIX Semaphores

- `sem_wait()`

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

  - If the value of the semaphore was *one* or *higher* when called `sem_wait()`, return right away.
  - It will cause the caller to suspend execution waiting for a subsequent post.
  - When negative, the value of the semaphore is equal to the number of waiting threads.

POSIX Semaphores

- sem_post()

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

  - Simply increments the value of the semaphore.
  - If there is a thread waiting to be woken, wakes one of them up.

POSIX Semaphores

- Unnamed semaphores
  - Creating and initializing the lock

```
#include <semaphore.h>
sem_t sem;              // global declaration

// Create the semaphore and initialize it to 1
sem_init(&sem, 0, 1);
```

  - Acquiring and releasing the lock

```
sem_wait(&sem);  // acquire the semaphore

/* critical section */

sem_post(&sem);  // release the semaphore
```

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

- *pshared*
  - If *pshared* has the value 0, then the semaphore is shared between the threads of a process.
  - If *pshared* is nonzero, then the semaphore is shared between processes.

https://man7.org/linux/man-pages/man3/sem_init.3.html

Binary Semaphores

- Using a semaphore as a lock
  - The initial value should be 1.

```
sem_t m;
sem_init(&m, 0, X); // init X to 1

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

<Two Threads Using A Semaphore>

Val	Thread 0	State	Thread 1	State
1		Run		Ready
1	call sem_wait()	Run		Ready
0	sem_wait() returns	Run		Ready
0	(crit sect begin)	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	call sem_wait()	Run
-1		Ready	decr sem	Run
-1		Ready	(sem<0)→sleep	Sleep
-1		Run	*Switch→T0*	Sleep
-1	(crit sect end)	Run		Sleep
-1	call sem_post()	Run		Sleep
0	incr sem	Run		Sleep
0	wake(T1)	Run		Ready
0	sem_post() returns	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	sem_wait() returns	Run
0		Ready	(crit sect)	Run
0		Ready	call sem_post()	Run
1		Ready	sem_post() returns	Run

Semaphores for Ordering

- Semaphores are also useful to order events in a concurrent program

```
sem_t s;

void *child(void *arg) {
    printf("child\n");
    sem_post(&s);  // signal here: child is done
    return NULL;
}

int main(int argc, char *argv[]) {
    sem_init(&s, 0, X); // what should X be?
    printf("parent: begin\n");
    pthread_t c;
    pthread_create(c, NULL, child, NULL);
    sem_wait(&s);      // wait here for child
    printf("parent: end\n");
    return 0;
}
```

What should the initial value of semaphore be?

Semaphores for Ordering

- Thread trace: Case 1
  - The parent call `sem_wait()` before the child has called `sem_post()`.

Val	Parent	State	Child	State
0	`create(Child)`	Run	*(Child exists, can run)*	Ready
0	call `sem_wait()`	Run		Ready
-1	`decr sem`	Run		Ready
-1	`(sem<0)→sleep`	Sleep		Ready
-1	*Switch→Child*	Sleep	`child runs`	Run
-1		Sleep	call `sem_post()`	Run
0		Sleep	`inc sem`	Run
0		Ready	`wake(Parent)`	Run
0		Ready	`sem_post()` returns	Run
0		Ready	*Interrupt→Parent*	Ready
0	`sem_wait()` returns	Run		Ready

Semaphores for Ordering

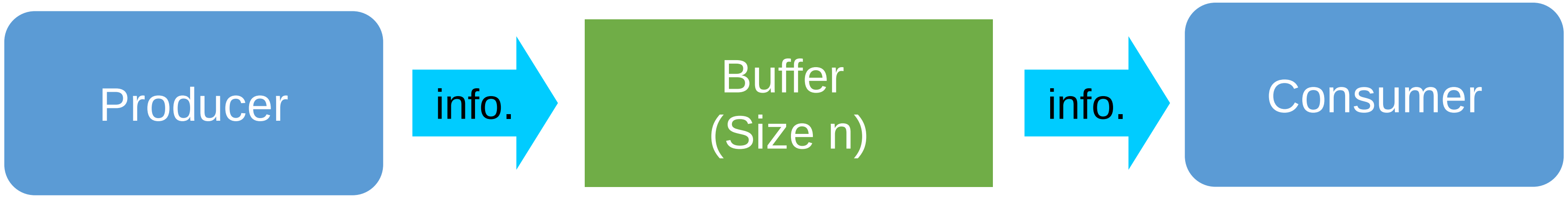
- Thread trace: Case 2
  - The child runs to completion before the parent call `sem_wait()`.

Val	Parent	State	Child	State
0	`create(Child)`	Run	*(Child exists; can run)*	Ready
0	*Interrupt→Child*	Ready	`child runs`	Run
0		Ready	call `sem_post()`	Run
1		Ready	`inc sem`	Run
1		Ready	`wake(nobody)`	Run
1		Ready	`sem_post()` returns	Run
1	`parent runs`	Run	*Interrupt→Parent*	Ready
1	call `sem_wait()`	Run		Ready
0	`decrement sem`	Run		Ready
0	`(sem≥0)→awake`	Run		Ready
0	`sem_wait()` returns	Run		Ready

Agenda

- Semaphores
- Bounded Buffer
- Reader-Writer Locks
- Dining Philosophers

The Bounded-Buffer Problem



- If the buffer is full, the producer must wait until the consumer deletes an item
  - Producer needs an empty space
  - **# of empty slot** is represented by a semaphore ***empty***
- If the buffer is empty, the consumer must wait until the producer adds an item
  - Consumer needs an item
  - **# of item** is represented by a semaphore ***full***

Bounded-Buffer Problem with Semaphore

```
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);
    }
}

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

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

void put(int value) {
    buffer[fill] = value;    // line f1
    fill = (fill + 1) % MAX;// line f2
}

int get() {
    int tmp = buffer[use];  // line g1
    use = (use + 1) % MAX;  // line g2
    return tmp;
}
```

- Imagine that MAX is greater than 1.
  - If there are multiple producers and consumers, race condition can happen.
  - It means that the old data there is overwritten.
  - The filling of a buffer and incrementing of the index into the buffer is a critical section.

A Solution: Adding 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);   // line P0 (NEW LINE)
        sem_wait(&empty);   // line P1
        put(i);             // line P2
        sem_post(&full);    // line P3
        sem_post(&mutex);   // line P4 (NEW LINE)
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);   // line C0 (NEW LINE)
        sem_wait(&full);    // line C1
        int tmp = get();    // line C2
        sem_post(&empty);   // line C3
        sem_post(&mutex);   // line C4 (NEW LINE)
        printf("%d\n", tmp);
    }
}
```

- Imagine two thread: one producer and one consumer.
  - The consumer acquire the `mutex` (`C0`).
  - The consumer calls `sem_wait()` on the full semaphore (`C1`).
  - The consumer is blocked and yield the CPU.
    - The consumer still holds the mutex!
  - The producer calls `sem_wait()` on the binary `mutex` semaphore (`P0`).
  - The producer is now stuck waiting too. a classic deadlock!

A Working Solution

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

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);   // line P0
        sem_wait(&mutex);   // line P1.5 (lock)
        put(i);             // line P2
        sem_post(&mutex);   // line P2.5 (unlock)
        sem_post(&full);    // line P3
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);    // line C1
        sem_wait(&mutex);   // line C1.5 (lock)
        int tmp = get();    // line C2
        sem_post(&mutex);   // line C2.5 (unlock)
        sem_post(&empty);   // line C3
        printf(“%d\n”, tmp);
    }
}
```

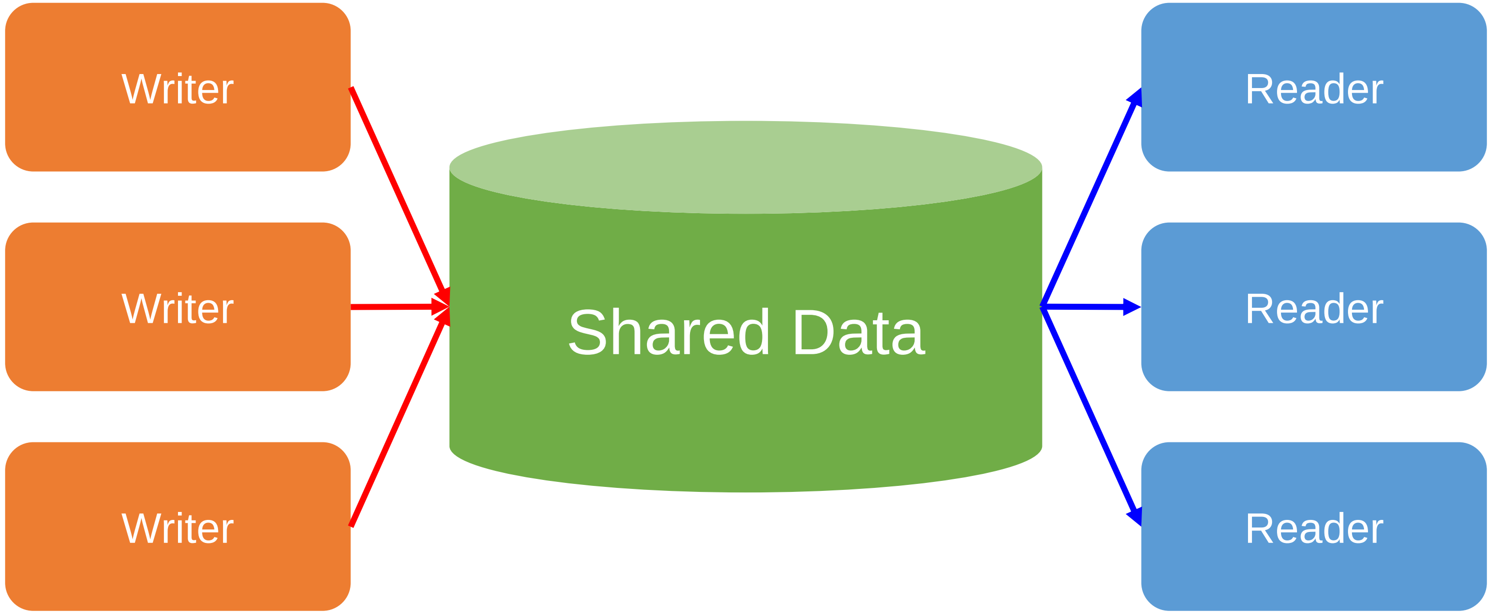
```
int main(int argc, char *argv[]) {
    // …
    sem_init(&empty, 0, MAX); // MAX buffers are empty
    sem_init(&full, 0, 0);    // ... and 0 are full
    sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock
    // …
 }
```

Agenda

- Semaphores
- Bounded Buffer
- Reader-Writer Locks
- Dining Philosophers

The Readers-Writers Problem

- There are multiple readers and writers to access a shared data
  - Readers can access database simultaneously.
  - When a writer is accessing the shared data, no other thread can access it.



Reader-Writer Locks

- Imagine a number of concurrent list operations, including inserts and simple lookups.
  - insert:
    - Change the state of the list
    - A traditional critical section makes sense.
  - lookup:
    - Simply read the data structure.
    - As long as we can guarantee that no insert is on-going, we can allow many lookups to proceed concurrently.

This special type of lock is known as a reader-write lock.

Reader-Writer Locks

- Only a single writer can acquire the lock.
- Once a reader has acquired a read lock,
  - More readers will be allowed to acquire the read lock too.
  - A writer will have to wait until all readers are finished.

```
typedef struct _rwlock_t {
    sem_t lock;      // binary semaphore (basic lock)
    sem_t writelock; // used to allow ONE writer or MANY readers
    int readers;     // count of readers reading in critical section
} 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

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

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

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

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

Reader-Writer Locks

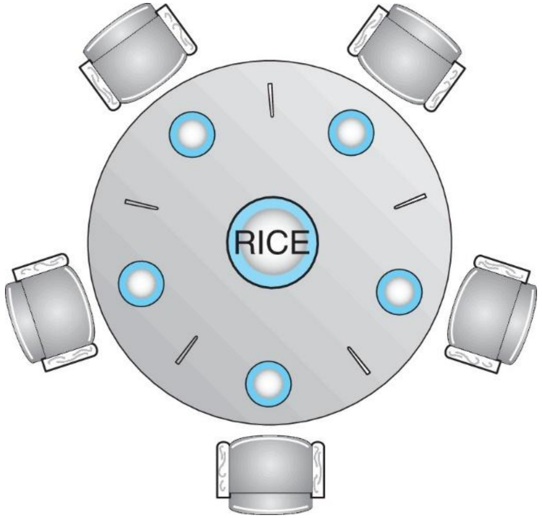
- The reader-writer locks have **fairness problem**.
  - It would be relatively easy for reader to starve writer.
  - How to prevent more readers from entering the lock once a writer is waiting?

Agenda

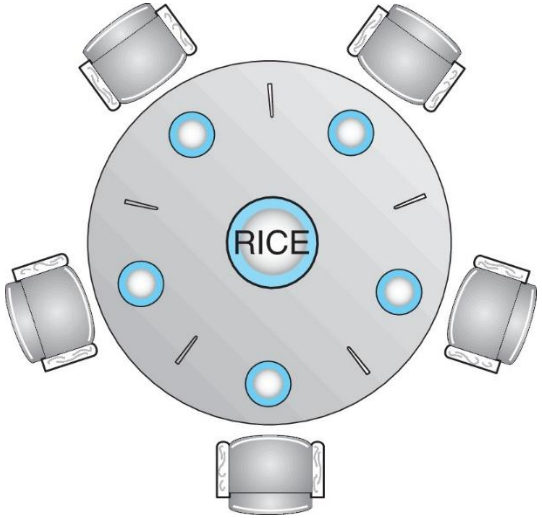
- Semaphores
- Bounded Buffer
- Reader-Writer Locks
- Dining Philosophers

The Dining Philosophers Problem

- Problem definition
  - 5 philosophers sitting on a circular table
    - Thinking or eating
  - 5 bowls, 5 single chopsticks
  - No interaction with colleagues
  - To eat, the philosopher should pick up two chopsticks closest to her
  - A philosopher can pick up only one chopstick at a time
  - When she finish eating, she release chopsticks
- Solution should be deadlock-free and starvation-free



The Dining Philosophers Problem



- Shared data
  - Bowl of rice (data set)
  - Semaphore `chopstick[5]` initialized to 1
- A possible solution, but deadlock can occur

```
while (true){
    sem_wait(chopstick[i]);          // pick up left chopstick
    sem_wait(chopstick[(i+1) % 5]);  // pick up right chopstick

    /* eat for a while */

    sem_post(chopstick[i]);          // release left chopstick
    sem_post(chopstick[(i+1) % 5]);  // release right chopstick

    /* think for a while */
}
```

  - If each philosopher happens to grab the chopstick on their left before any philosopher can grab the chopstick on their right. Each will be stuck holding one chopstick and waiting for another, forever.

A Solution: Breaking The Dependency

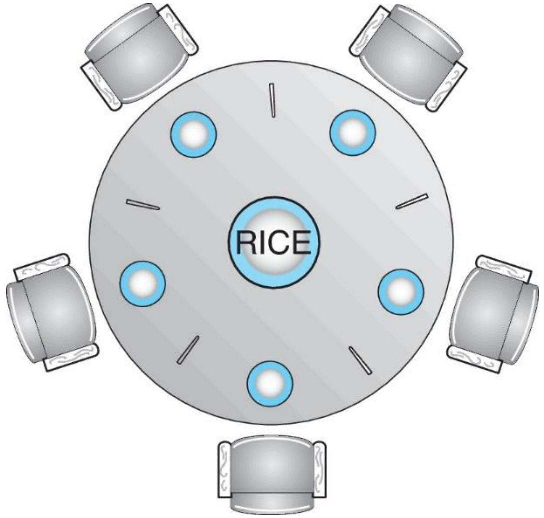
- Possible solution
  - Let's assume that philosopher 4 acquire the chopstick in a different order.

```
if (p == 4) {
    sem_wait(chopstick[(i+1) % 5]); // pick up right chopstick
    sem_wait(chopstick[i]);         // pick up left chopstick
} else {
    sem_wait(chopstick[i]);         // pick up left chopstick
    sem_wait(chopstick[(i+1) % 5]); // pick up right chopstick
}
```

  - There is no situation where each philosopher grabs one chopstick and is stuck waiting for another. The cycle of waiting is broken.

A Solution: Breaking The Dependency

- Another possible solution
  - Allow at most four philosophers to be sitting simultaneously at the table
  - Allow a philosopher to pick up her chopsticks only if both chopsticks are available
  - An odd numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even-numbered philosophers picks up her right chopstick and then her left chopstick



Above solutions don't necessarily eliminate the possibility of starvation.

References

- Ch31, Operating Systems: Three Easy Pieces
- https://oslab.kaist.ac.kr/ostepslides/