COS40003 Concurrent Programming

Lecture 8: Semaphore

CRICOS 00111D
TOID 3059

KNOW ING

Outline

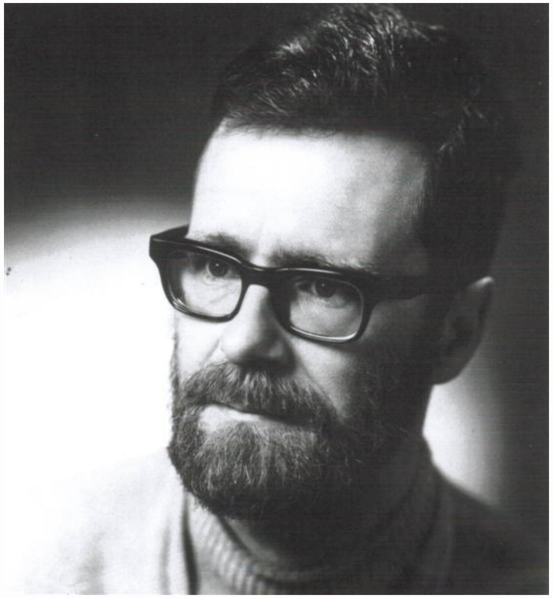
- Why semaphores?
- What is a semaphore?
- How to use semaphores?
  - Binary semaphores (similar to locks)
  - Semaphore for ordering (similar to condition variable)
  - Producer and Consumer using semaphore
  - Implementing semaphore with lock and condition variable
- Java methods

Why semaphore?

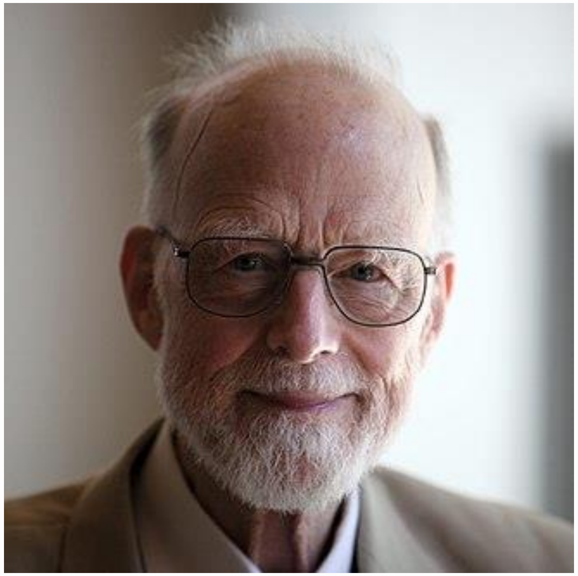
- We introduced Locks and Condition Variables, and we can write good concurrent programs using them.
- Why do we need semaphore?
- Reason
  - Two concepts share similar idea, but not the same thing. They were proposed by different people.

Why semaphore?

- Semaphore
  - 1968 by Dijkstra
  - Proposed "private semaphore" in "Cooperating sequential processes"



- Condition variable
  - 1974 by Hoare
  - In his work on "monitors" in "Monitors: An Operating System Structuring Concept"



Outline

- Why semaphores?
- What is a semaphore?
- How to use semaphores?
  - Binary semaphores (similar to locks)
  - Semaphore for ordering (similar to condition variable)
  - Producer and Consumer using semaphore
  - Implementing semaphore with lock and condition variable
- Java methods

What is a semaphore?

- A semaphore is an object with an integer value that we can manipulate with two routines;
  - P/down/wait/ acquire (decrements the count)
  - V/ up /post/ release (increments the count)
  - Eg., P(semaphore), V(semaphore)

  - P,V – (from Dutch words prolaag, verlaag)
  - wait, post – POSIX standard
  - acquire, release – Java API

Initialize a semaphore

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

- 1. declare a semaphore s
- 2. second argument of sem_init() set to 0, this indicates the semaphore is shared between threads in the same process. (A different value for sharing across different processes.)
- 3. third argument, initialize the semaphore's value

P (wait) and V (post)

- sem_wait() - atomic
- sem_post() - atomic

```
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
}
```

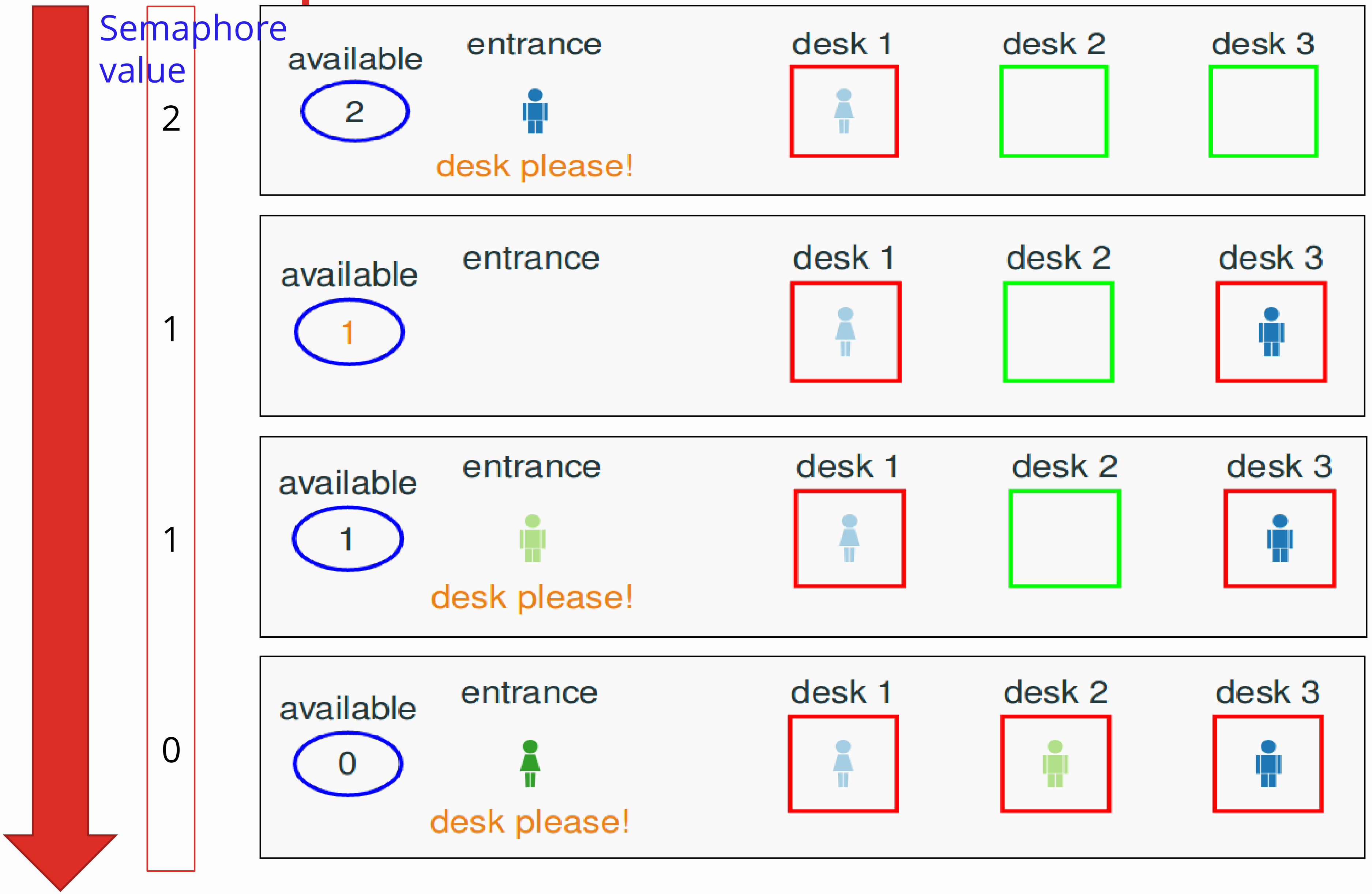
Discussion of wait and post

- 1. sem_wait()
  - either return right way,
  - Or cause the caller to suspend, waiting for a subsequent post.
- 2. sem_post()
  - simply increments the value of the semaphore and then, if there is a thread waiting to be woken, wakes one of them up.
- 3. The value of the semaphore, when negative, its additive inverse is equal to the number of waiting threads

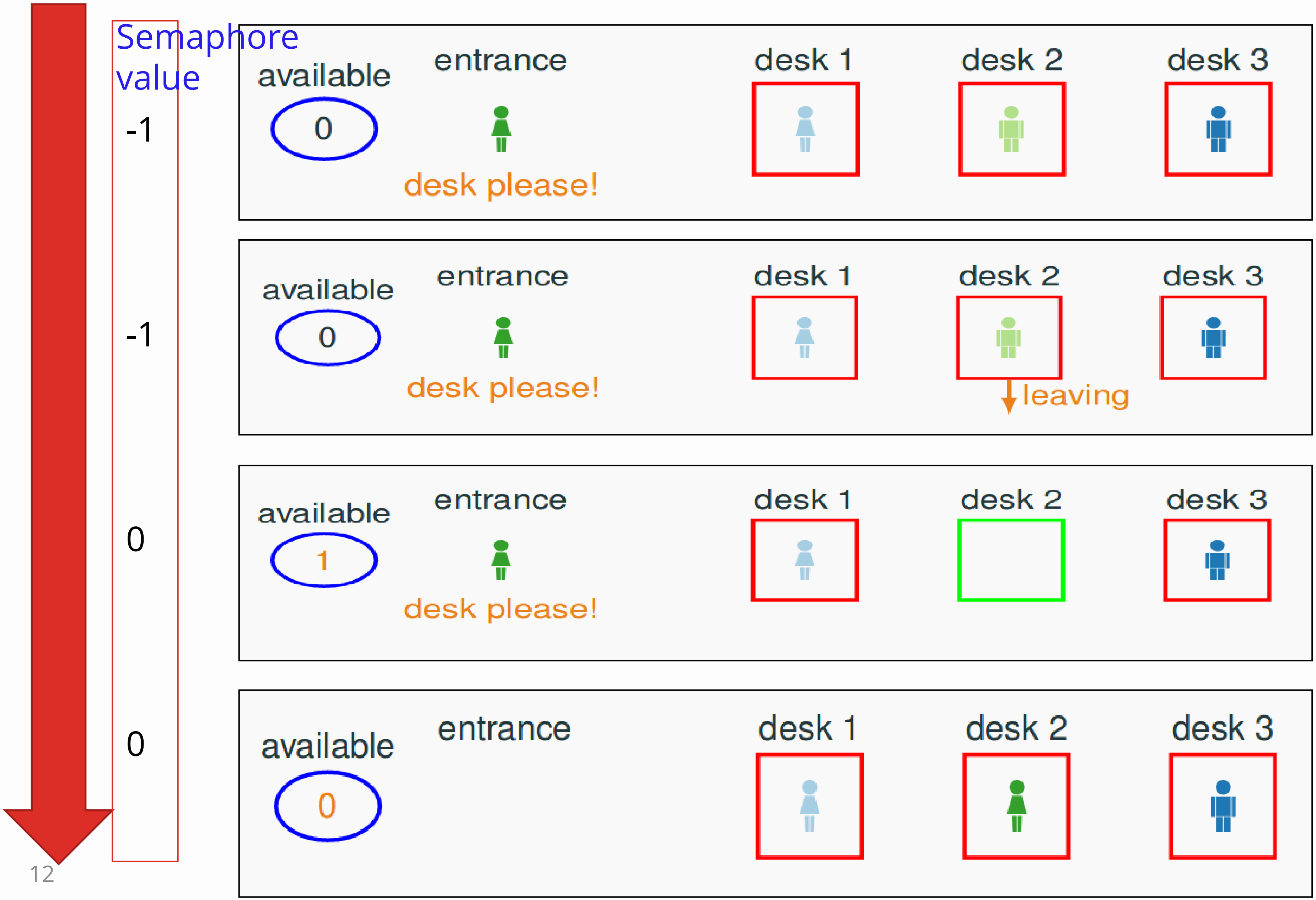
Break

- Ignore how sem_wait() and sem_post() are implemented atomically, which will be discussed later
- First see how semaphore can be used?

Example: hot desks



Example: hot desks



Outline

- Why semaphores?
- What is a semaphore?
- How to use semaphores?
  - Binary semaphores (similar to locks)
  - Semaphore for ordering (similar to condition variable)
  - Producer and Consumer using semaphore
  - Implementing semaphore with lock and condition variable
- Java methods

Binary Semaphores (Locks)

- Using a semaphore as a lock

```
sem_t m;
sem_init(&m, 0, X);

// initialize semaphore to X; what should X be?

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

- X=1

Binary Semaphores (Locks): running example 1

- Case 1:
  - Thread 0 calls sem_wait();
  - Semaphore value X=1 decreases to X=0;
  - X>=0, so return from sem_wait();
  - Thread 0 enters critical section and finishes;
  - Thread 0 calls sem_post();
  - Semaphore value X=0 increases to X=1;
  - Wake any waiting thread, none in this case, we are done.

Binary Semaphores (Locks): running example 2

- Case 2:
  - Thread 0 calls sem_wait();
  - Semaphore value X=1 decreases to X=0;
  - X>=0, so return from sem_wait();
  - Thread 0 enters critical section;

  --context switch, recall only wait(),post() are atomic--

  - Thread 1 calls sem_wait();
  - Semaphore value X=0 decreases to X= -1;
  - X<0, so Thread 1 puts itself to sleep, waiting;

  - (To be continued)

Binary Semaphores (Locks): running example 2

- Case 2:
  - (continue with the previous page)

  --context switch to Thread 0, because Thread 1 is sleeping--

  - Thread 0 finishes and calls sem_post();
  - Semaphore value X= -1 increases to X=0;
  - Wake any waiting thread, in this case, Thread 1 waken up.

  - Thread1 enters critical section and finishes
  - Thread 1 calls sem_post();
  - Semaphore value X=0 increases to X=1;
  - Wake any waiting thread, in this case, none and we are done

Binary Semaphores (Locks): conclusion

- Binary semaphores can simulate locks
- Question: any difference you find between binary semaphores and locks?
- Answer: semaphore support transferring of permissions
  - using a lock, only the thread decrements the counter can increment it back to 1;
  - using a semaphore, one thread decrements the counter to 0 and may let another thread increment it to 1.

Outline

- Why semaphores?
- What is a semaphore?
- How to use semaphores?
  - Binary semaphores (similar to locks)
  - Semaphore for ordering (similar to condition variable)
  - Producer and Consumer using semaphore
  - Implementing semaphore with lock and condition variable
- Java methods

Recall the example: parent waiting for child

```
void *child(void *arg) {
    printf("child\n");
   // XXX how to indicate we are done?
return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL);
    // create child
    // XXX how to wait for child?
   printf("parent: end\n");
   return 0;
}
```

Semaphores for Ordering (similar to condition variable)

```
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;
}
```

Answer:

X=0

Semaphores For Ordering (similar to condition variable)

```
sem_t s;
void * child(void *arg) {
        printf("child\n");
        sem_post(&s); // signal here: child is don
        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;
}
```

Case 1: parent runs before child

1. Parent calls wait() before child calls post();
2. Semaphore X=0 decreased to X= -1
3. X<0, so parent puts itself to sleep
4. Child calls post;
5. Semaphore X= -1 increased to X=0;
6. Parent waken up and finishes

Semaphores For Ordering (similar to condition variable)

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

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

Case 2: parent runs after child

1. child calls post() before parent calls wait();
2. Semaphore X=0 increased to X=1
3. Wake up any one waiting, in this case, none
4. Later, parent calls wait;
5. Semaphore X=1 decreased to X=0;
6. X>=0, parent will not be put to sleep. Parent returns from wait() and go on

Outline

- Why semaphores?
- What is a semaphore?
- How to use semaphores?
  - Binary semaphores (similar to locks)
  - Semaphore for ordering (similar to condition variable)
  - Producer and Consumer using semaphore
  - Implementing semaphore with lock and condition variable
- Java methods

Semaphore in the Producer/Consumer Problem

- Recall: In Lecture 7
- A consumer should only wake producers, and a producer should only wake consumers.
  - Solution: two condition variable
  - Otherwise: wrong wakeups

Semaphore in the PC Problem - First attempt

- Use two semaphores, empty and full
- The producer first waits for a buffer to become empty in order to put data into it, and the consumer similarly waits for a buffer to become filled before using it.

Semaphore in the PC Problem - First attempt

```
int buffer[MAX];
int fill = 0;   // put pointer
int use = 0;   // get pointer

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

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

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

void *consumer(void *arg) {
        while (tmp != -1) {
                sem_wait(&full);
                tmp = get();
                sem_post(&empty);
        }
}
int main(int argc, char *argv[]) {
sem_init(&empty, 0, MAX); // MAX buffers are empty to begin
sem_init(&full, 0, 0); // ... and 0 are full
...
}
```

Is it correct?

Suppose MAX=1, let us examine

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

void *consumer(void *arg) {
        while (tmp != -1) {
                sem_wait(&full);
                tmp = get();
                sem_post(&empty);
        }
}
int main(int argc, char *argv[]) {
sem_init(&empty, 0, MAX); // MAX buffers are
sem_init(&full, 0, 0); // ... and 0 are full
...
}
```

Example: MAX = 1

1. A consumer runs first, calls wait(&full), full is set to -1, consumer is blocked waiting for a post(&full);
2. A producer comes, calls wait(&empty), empty is set to 0 from MAX=1.
3. The producer puts data into the buffer and calls post(&full), full will be set to 0 from -1, wakes up the consumer.
4. (1) The producer keeps running, calls wait() again, empty is set to -1, the producer is blocked;
5. (2) The consumer waken up, calls get(), then post(&empty)

Semaphore in the PC Problem - First attempt

- Discussion:

- Question, suppose MAX =1, will it work with “multiple producers, and multiple consumers”?

- Answer: Yes

Semaphore in the PC Problem - First attempt

- Discussion:

- Question 2, suppose MAX=10, will it work with “multiple producers, and multiple consumers”?

- Answer: No

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

void *consumer(void *arg) {
        while (tmp != -1) {
                sem_wait(&full);
                tmp = get();
                sem_post(&empty);
        }
}
int main(int argc, char *argv[]) {
sem_init(&empty, 0, MAX); // MAX buffers are empty to begin
sem_init(&full, 0, 0); // ... and 0 are full
...
}
```

Recall:

only wait() and post(), the functions themselves are guaranteed to be atomic!

Semaphore in the PC Problem - First attempt

```
int buffer[MAX];
int fill = 0;    // put pointer
int use = 0;     // get pointer

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

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

Thread 1 calls put();

Thread 2 call put();

Sequence is:

Thread1:

buffer[fill] = value;

Thread 2:

buffer[fill] = value;

fill = (fill + 1) % MAX

Thread1:

fill = (fill + 1) % MAX

A solution: adding mutual exclusion

- What we’ve forgotten here is mutual exclusion
  - The filling of a buffer and incrementing of the index into the buffer is a critical section, and thus must be guarded carefully.
  - Any idea?

```
sem_t empty;    sem_t full;        sem_t mutex;
void *producer(void *arg) {
        for (int i = 0; i < loops; i++) {
                sem_wait(&mutex);
                sem_wait(&empty);
                put(i);
                sem_post(&full);
                sem_post(&mutex);
        }
}
void *consumer(void *arg) {
        while (tmp != -1) {
                sem_wait(&mutex);
                sem_wait(&full);
                tmp = get();
                sem_post(&empty);
                sem_post(&mutex);
        }
}
int main(int argc, char *argv[]) {
sem_init(&empty, 0, MAX); // MAX buffers are empty to begin
sem_init(&full, 0, 0); // ... and 0 are full
sem_init(&mutex, 0, 1);
...
}
```

Question:

Any problem of this implementation?

```
sem_t empty;     sem_t full;        sem_t mutex;
void *producer(void *arg) {
        for (int i = 0; i < loops; i++) {
                sem_wait(&mutex);
                sem_wait(&empty);
                put(i);
                sem_post(&full);
                sem_post(&mutex);
        }
}
void *consumer(void *arg) {
        while (tmp != -1) {
                sem_wait(&mutex);
                sem_wait(&full);
                tmp = get();
                sem_post(&empty);
                sem_post(&mutex);
        }
}
int main(int argc, char *argv[]) {
sem_init(&empty, 0, MAX); // MAX buffers are
sem_init(&full, 0, 0); // ... and 0 are full
sem_init(&mutex, 0, 1);
...
}
```

Dead lock:

1. A consumer comes first, acquires the mutex, call wait(&full), since the buffer is empty, the consumer will be blocked waiting for a producer to post(&full);
2. A producer comes later, try to acquire the mutex, but not successful, will be blocked and wait for a consumer to post(&mutex);
3. Ending up with waiting for each other

```c
sem_t empty;     sem_t full;        sem_t mutex;
void *producer(void *arg) {
        for (int i = 0; i < loops; i++) {
                sem_wait(&mutex);
                sem_wait(&empty);
                put(i);
                sem_post(&full);
                sem_post(&mutex);
        }
}
void *consumer(void *arg) {
        while (tmp != -1) {
                sem_wait(&mutex);
                sem_wait(&full);
                tmp = get();
                sem_post(&empty);
                sem_post(&mutex);
        }
}
int main(int argc, char *argv[]) {
sem_init(&empty, 0, MAX); // MAX buffers are empty to begin
sem_init(&full, 0, 0); // ... and 0 are full
sem_init(&mutex, 0, 1);
...
}
```

How to fix this?

Hint:

Change the scope of the lock

```
sem_t empty;     sem_t full;          sem_t mutex;
void *producer(void *arg) {
        for (int i = 0; i < loops; i++) {
                sem_wait(&empty);
                sem_wait(&mutex);
                put(i);
                sem_post(&mutex);
                sem_post(&full);
                        }
}
void *consumer(void *arg) {
        while (tmp != -1) {
                sem_wait(&full);
                sem_wait(&mutex);
                tmp = get();
                sem_post(&mutex);
                sem_post(&empty);
        }
}
int main(int argc, char *argv[]) {
sem_init(&empty, 0, MAX); // MAX buffers are empty to begin
sem_init(&full, 0, 0); // ... and 0 are full
sem_init(&mutex, 0, 1);
...
}
```

How to fix this?

Answer:

Change the order of the lock

Protect only the critical section.

Outline

- Why semaphores?
- What is a semaphore?
- How to use semaphores?
  - Binary semaphores (similar to locks)
  - Semaphore for ordering (similar to condition variable)
  - Producer and Consumer using semaphore
  - Implementing semaphore with lock and condition variable
- Java methods

Simulate Semaphores with locks and condition variables

- Example of using locks and condition variable to simulate semaphore, called “Zemaphore”

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

```c
// 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);
}

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);
}
```

Outline

- Why semaphores?
- What is a semaphore?
- How to use semaphores?
  - Binary semaphores (similar to locks)
  - Semaphore for ordering (similar to condition variable)
  - Producer and Consumer using semaphore
  - Implementing semaphore with lock and condition variable
- Java methods

Java methods

```
Semaphore sem = new Semaphore(count);
//or
sem = new Semaphore(count, true/false);
// true: strong - false: weak

void acquire(); // P

void release(); // V

int availablePermits(); //returns count

/* more useful functions in:
https://docs.oracle.com/javase/10/docs/api/java/util/concurrent/Semaphore.html
*/
```

Strong/Weak Semaphore

- A **queue** is used to hold threads **waiting** on the semaphore
  - In what order are threads removed from the queue?

- ***Strong Semaphores*** use **FIFO**
- ***Weak Semaphores*** not deterministic

Acknowledgement

- Chapter 31
  - Operating Systems: Three Easy Pieces
- Java documentation
  - https://docs.oracle.com/javase/10/docs/api/java/util/concurrent/Semaphore.html

Further reading (recommended, not required)

- One great (and free reference) is Allen Downey's book on concurrency and programming with semaphores:

- "**The Little Book of Semaphores**"

- This book has lots of puzzles you can work on to improve your understanding of both semaphores in specific and concurrency in general.

Questions?