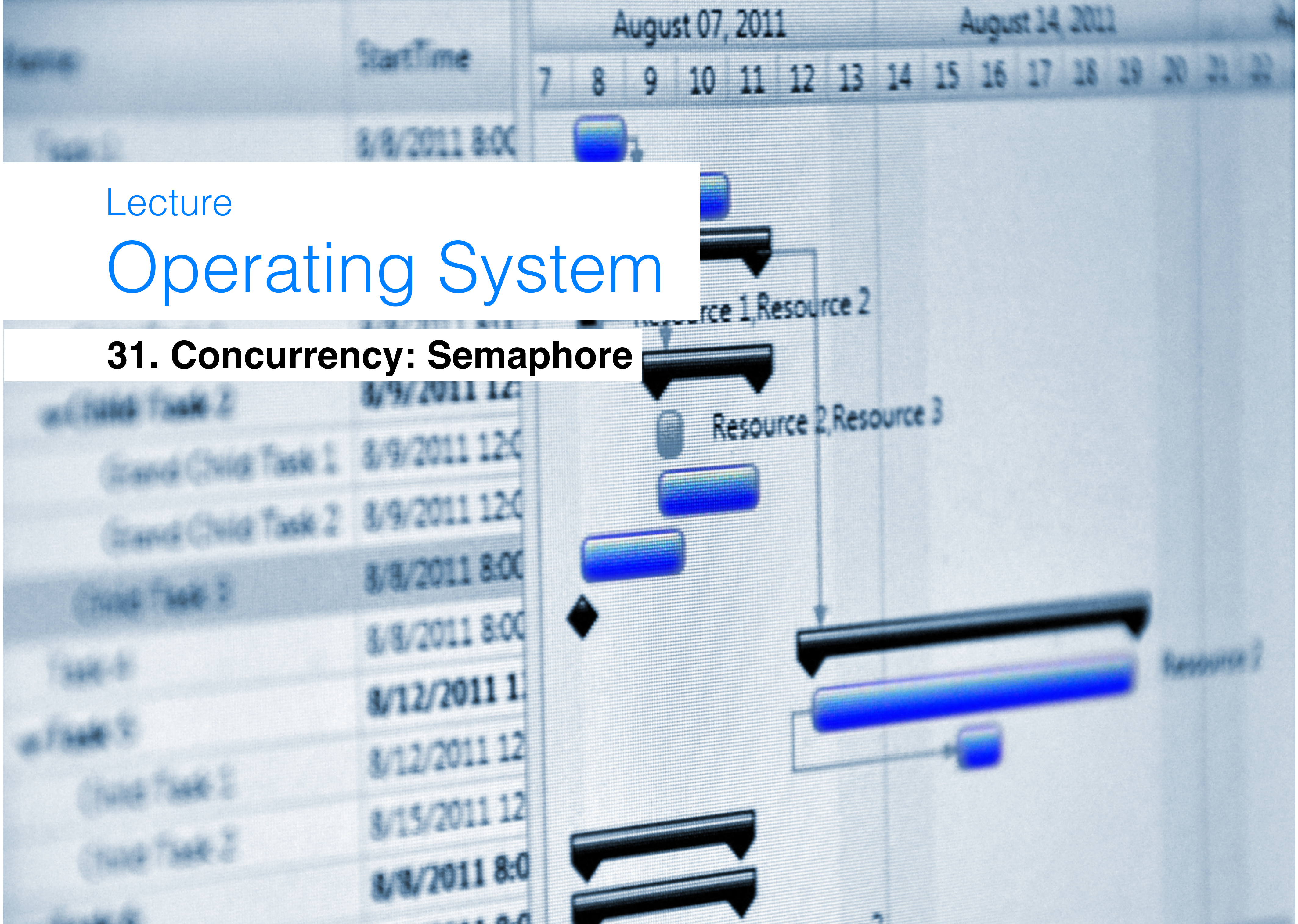


Lecture

Operating System

31. Concurrency: Semaphore

Professor Dr. Michael Mächtel

Concurrency Objectives

- **Mutual exclusion** (e.g., A and B don't run at same time)
  - solved with locks
- **Ordering** (e.g., B runs after A does something)
  - solved with condition variables and **semaphores**

31. Semaphore

1. **Protect Critical Sections**
2. **Use as Condition Variable**
3. **Solve Bounded Buffer Problem**
4. **Reader-Writer Locks**

31. Semaphore

1. **Protect Critical Sections**
2. **Use as Condition Variable**
3. **Solve Bounded Buffer Problem**
4. **Reader-Writer Locks**

Semaphore: A definition

- An object **with an integer value**
  - We can manipulate with two routines;
    - `sem_wait()` and `sem_post()`.
  - Initialization
    - Declare a semaphore `s` and initialize it to the value `1`
    - The second argument, `0`, indicates that the semaphore is shared between threads in the same process.

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1); // initialize s to the value 1
```

Interact with semaphore

- `sem_wait()`
  - 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.

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

Interact with semaphore (Cont.)

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

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

Possible Implementation

```
struct semaphore {
     int value;
     queueType list;
     }


 sem_wait(semaphore *S) {
 S->value--;
 if (S->value < 0) {
      add this task
          to S->list;
      block();
 }
}

sem_post(semaphore *S) {
S->value++;
if (S->value <= 0) {
    remove a task P
 from S->list;
    wakeup(P);
  }
}
```

Binary Semaphores (Locks)

- What should **X** be?
  - The initial value should be 1.

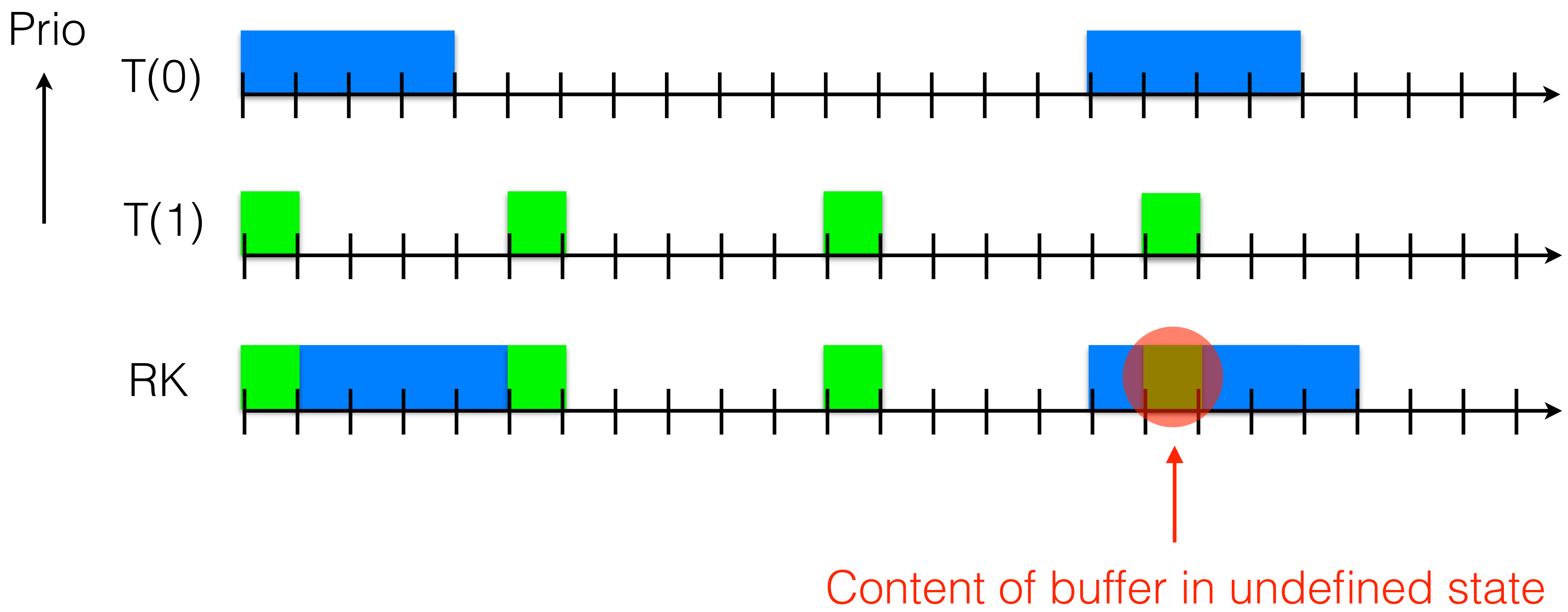
```
sem_t m;
sem_init(&m, 0, X); // initialize semaphore to
                    // X; what should X be?

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

Thread Trace: Single Thread Using A Semaphore

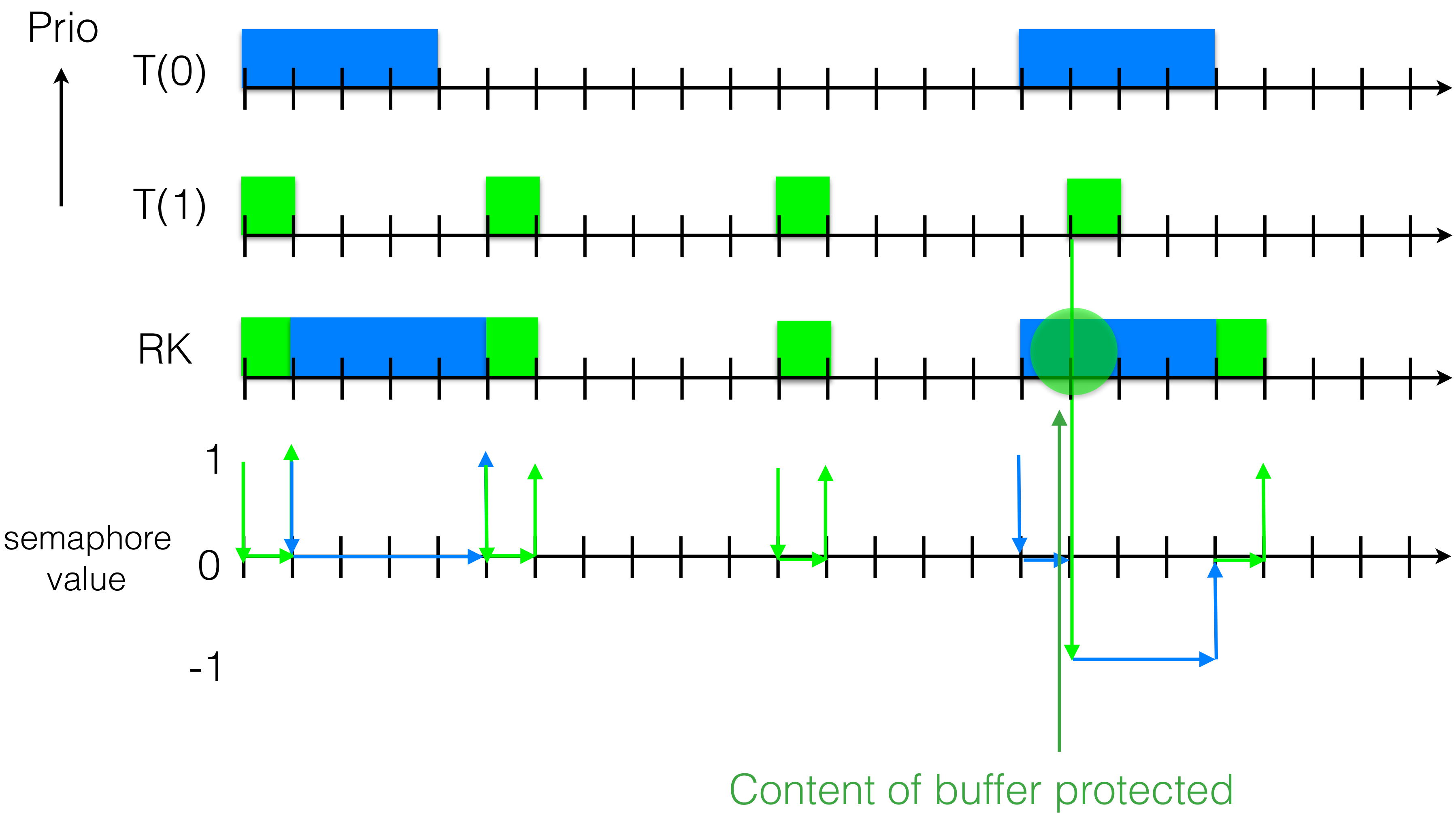
Value of Semaphore	Thread 0
1	
1	call sema_wait()
0	sem_wait() returns
0	(crit sect)
0	call sem_post()
1	sem_post() returns

Critical Section (buffer) unprotected

- Two periodic threads: T(0) and T(1)
  - Priority scheduling: T(1) has higher Priority than T(0)
- T(0) and T(1) share a buffer during runtime each instance



Critical Section (buffer) protected
by semaphore



Two Threads Using A Semaphore

Thread Trace

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call sem_wait()	Running		Ready
0	sem_wait() retruns	Running		Ready
0	(crit set: begin)	Running		Ready
0	*Interrupt; Switch → T1*	Ready		Running
0		Ready	call sem_wait()	Running
-1		Ready	decrement sem	Running
-1		Ready	(sem < 0)→sleep	sleeping
-1		Running	*Switch → T0*	sleeping
-1	(crit sect: end)	Running		sleeping
-1	call sem_post()	Running		sleeping
0	increment sem	Running		sleeping
0	wake(T1)	Running		Ready
0	sem_post() returns	Running		Ready
0	*Interrupt; Switch → T1*	Ready		Running
0		Ready	sem_wait() retruns	Running
0		Ready	(crit sect)	Running
0		Ready	call sem_post()	Running
1		Ready	sem_post() returns	Running

31. Semaphore

1. Protect Critical Sections
2. **Use as Condition Variable**
3. Solve Bounded Buffer Problem
4. Reader-Writer Locks

Semaphores As Condition Variables

```
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 **X** be?
  - The initial value should be 0.

What we would like to see here is:

```
parent: begin
child
parent: end
```

Parent Waiting For Child (Case 1)

Thread Trace

The parent call `sem_wait()` before the child has called `sem_post()`.

Val	Parent	State	Child	State
0	`Create(Child)`	Running	*`(Child exists; is runnable)`*	Ready
0	`call sem_wait()`	Running		Ready
-1	`decrement sem`	Running		Ready
-1	`(sem < 0)→sleep`	sleeping		Ready
-1	*`Switch→Child`*	sleeping	`child runs`	Running
-1		sleeping	`call sem_post()`	Running
0		sleeping	`increment sem`	Running
0		Ready	`wake(Parent)`	Running
0		Ready	`sem_post() returns`	Running
0		Ready	*`Interrupt; Switch→Parent`*	Ready
0	`sem_wait() returns`	Running		Ready

Parent Waiting For Child (Case 2)

Thread Trace

The child runs to completion before the parent call `sem_wait()`.

Val	Parent	State	Child	State
0	`Create(Child)`	Running	(Child exists; is runnable)	Ready
0	*`Interrupt; switch→Child`*	Ready	`child runs`	Running
0		Ready	`call sem_post()`	Running
1		Ready	`increment sem`	Running
1		Ready	`wake(nobody)`	Running
1		Ready	`sem_post() returns`	Running
1	`parent runs`	Running	*`Interrupt; Switch→Parent`*	Ready
1	`call sem_wait()`	Running		Ready
0	`decrement sem`	Running		Ready
0	`(sem<0)→awake`	Running		Ready
0	`sem_wait() returns`	Running		Ready

31. Semaphore

1. Protect Critical Sections
2. Use as Condition Variable
3. **Solve Bounded Buffer Problem**
4. Reader-Writer Locks

The Producer/Consumer
(Bounded-Buffer) Problem

- **Producer**: `put( )` interface
  - Wait for a buffer to become **empty** in order to put data into it.
- **Consumer**: `get( )` interface
  - Wait for a buffer to become **filled** before using it.

```c
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;
}
```

A Solution: Adding `empty/full`

(Bounded-Buffer) Problem

```c
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);
    }
}
```

Result of First Attempt?

- Imagine that `MAX` is greater than 1 .
  - If there are **multiple** producers, **race condition** can happen at line **f1**.
  - It means that the old data there is overwritten.
- We've forgotten here is **mutual exclusion**.
  - The filling of a buffer and incrementing of the index into the buffer is a **critical section**.

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

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

A Solution: Adding Mutual Exclusion

(Bounded-Buffer) Problem

```
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);
    }
}
```

~~Correct?~~

Imagine two thread: one producer and one consumer....

No Solution: It's a deadlock

- Imagine two thread: one producer and one consumer.
  - The consumer **acquire** the `mutex` (line c0).
  - The consumer **calls** `sem_wait()` on the full semaphore (line 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 (line p0).
  - The producer is now **stuck** waiting too.
    - **a classic deadlock!**

A Solution: Adding Mutual Exclusion

(Bounded-Buffer) Problem

Correctly!

```
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 p1
        sem_wait(&mutex);   (MOVED MUTEX HERE...)
        put(i);             // line p2
        sem_post(&mutex);   (MOVED MUTEX HERE...)
        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);   (MOVED MUTEX HERE...)
        int tmp = get();    // line c2
        sem_post(&mutex);   (MOVED MUTEX HERE...)
        sem_post(&empty);   // line c3
    }
}
```

31. Semaphore

1. Protect Critical Sections
2. Use as Condition Variable
3. Solve Bounded Buffer Problem
4. **Reader-Writer Locks**

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-writer 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);
}
```

Acquire/Release Reader-Writer-Lock

```c
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);
}
```

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

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

Summary Reader-Writer-Lock

- 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?

Thanks

Questions?