Concurrency: Semaphores

Sridhar Alagar

Synchronization Objectives

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

Condition Variables

- Pthread_cond_wait(cond_t *cv, mutex_t *lock)
  - assumes the lock is held when wait() is called
  - puts caller to sleep + releases the lock (atomically)
  - when awoken, reacquires lock before returning

- Pthread_cond_signal(cond_t *cv)
  - wake a single waiting thread (if >= 1 thread is waiting)
  - if there is no waiting thread, just return, doing nothing

Join Implementation

Parent:

```
void thread_join() {
        Mutex_lock(&m);              // w
        if (done == 0)               // x
                Cond_wait(&c, &m);   // y
        Mutex_unlock(&m);            // z
}
```

Child:

```
void thread_exit() {
        Mutex_lock(&m);              // a
        done = 1                     // b
        Cond_signal(&c);             // c
        Mutex_unlock(&m);            // d
}
```

Producer Consumer Problem

- Class of problems where producer generates data/jobs and consumer consumes/services

- Synchronization is required among producers and consumers

Producer Consumer Solution

Simple case:

- One producer thread
- One consumer thread
- Shared buffer of size 1

Producer Consumer: Two CVs and while

```
void *producer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&empty, &m)
        do_fill();
        Cond_signal(&fill);
        Mutex_unlock(&m);
    }
}
```

```
void *consumer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        while (numfull == 0)
            Cond_wait(&fill, &m)
        int tmp = do_get();
        Cond_signal(&empty);
        Mutex_unlock(&m);
    }
}
```

Is this correct?

Correct

- no concurrent access to shared state
- every time lock is acquired, assumptions are reevaluated
- a consumer will get to run after every do_fill()
- a producer will get to run after every do_get()

Semaphores - State variable + Queue

- CVs have no state variable other than a queue
  - Programmer has to maintain it

- Semaphores have state – an integer variable
  - User cannot access state directly other than initializing it
  - Behaves based on its state

Semaphores - operations

- Initialization

```
sem_t sem;
sem_init(sem_t *s, int initval) {
  s->value = initval;
}
```

  - user cannot read or write after initialization

Semaphores - operations

- Wait or P()

```
sem_wait(sem_t *s) {
  Decrements the value of semaphore s by 1;
  wait if the value of semaphore s is -ve
}
```

  - When value of semaphore is -ve, the absolute value denotes the number of waiting thread

Semaphores - operations

- Signal or V()

```
sem_post(sem_t *s) {
  increments the value of semaphore s by 1;
  wakeup one thread if the value of
  semaphore s is <= 0
}
```

Locks using Semaphores (binary)

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

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

What should be the value of X?

Locks using Semaphores (binary)

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

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

Trace: two threads - binary semaphore

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	`call sem_wait()`	Running		Ready
0	`sem_wait() returns`	Running		Ready
0	`(crit sect: 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

Join with CVs vs Semaphores

Parent:

```
void thread_join() {
        Mutex_lock(&m);             // w
        if (done == 0)              // x
                Cond_wait(&c, &m);  // y
        Mutex_unlock(&m);           // z
}
```

Child:

```
void thread_exit() {
        Mutex_lock(&m);             // a
        done = 1                    // b
        Cond_signal(&c);            // c
        Mutex_unlock(&m);           // d
}
```

Semaphores:

```
sem_t s;
sem_init(&s, ??)
void thread_join() { sem_wait(&s); }
Void thread_exit() { sem_post(&s); }
```

Trace 1: Parent waiting for Child

Parent calls sem_wait() before child calls sem_post()

Value	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

Trace 2: Parent waiting for Child

Child calls sem_post() before parent calls sem_wait()

Value	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

Producer/Consumer: Semaphores #1

Simplest case:

- Single producer thread, single consumer thread
- Single shared buffer between producer and consumer

Requirements

- Consumer must wait for producer to fill buffer
- Producer must wait for consumer to empty buffer (if filled)

Requires 2 semaphores

- emptyBuffer: Initialize to ??? 1 → 1 empty buffer; producer can run 1 time first
- fillBuffer: Initialize to ??? 0 → 0 full buffer; consumer can run 0 times first

Producer

```
While (1) {

      Fill(&buffer);


}
```

Consumer

```
While (1) {

      Use(&buffer);


}
```

Producer/Consumer: Semaphores #1

Simplest case:

- Single producer thread, single consumer thread
- Single shared buffer between producer and consumer

Requirements

- Consumer must wait for producer to fill buffer
- Producer must wait for consumer to empty buffer (if filled)

Requires 2 semaphores

- emptyBuffer: Initialize to ??? 1 → 1 empty buffer; producer can run 1 time first
- fillBuffer: Initialize to ??? 0 → 0 full buffer; consumer can run 0 times first

Producer

```
While (1) {
        sem_wait(&emptyBuffer);
        Fill(&buffer);
        sem_post(&fillBuffer);
}
```

Consumer

```
While (1) {
        sem_wait(&fillBuffer);
        Use(&buffer);
        sem_post(&emptyBuffer);
}
```

Producer/Consumer: Semaphores #2

Next case: Circular buffer

- Single producer thread, single consumer thread
- Shared buffer of size N between producer and consumer

Requires 2 semaphores

- emptyBuffer: Initialize to ??? N → N empty buffer; producer can run N times first
- fillBuffer: Initialize to ??? 0 → 0 full buffer; consumer can run 0 time first

Producer

```
i = 0;
while (1) {
        sem_wait(&emptyBuffer);
        Fill(&buffer);
        i = (i+1)%N;
        sem_post(&fillBuffer);
}
```

Consumer

```
j = 0;
while (1) {
        sem_wait(&fillBuffer);
        Use(&buffer);
        j = (j+1)%N;
        sem_post(&emptyBuffer);
}
```

Producer/Consumer: Semaphores #3

Final case:

- Multiple producer thread, Multiple consumer thread
- Shared buffer of size N between producer and consumer

Requires 2 semaphores

- emptyBuffer: Initialize to ??? N → N empty buffer; producer can run N times first
- fillBuffer: Initialize to ??? 0 → 0 full buffer; consumer can run 0 time first

Why will the previous code (shown below) not work? **Need Mutex**

Producer

```
i = 0;
while (1) {
        sem_wait(&emptyBuffer);
        Fill(&buffer);
        i = (i+1)%N;
        sem_post(&fillBuffer);
}
```

Consumer

```
j = 0;
while (1) {
        sem_wait(&fillBuffer);
        Use(&buffer);
        j = (j+1)%N;
        sem_post(&emptyBuffer);
}
```

Producer/Consumer: Semaphores #3

Final case:

- Multiple producer thread, Multiple consumer thread
- Shared buffer of size N between producer and consumer

Mutex added. Will this work? Deadlock

Producer

```
i = 0;

while (1) {

        sem_wait(&mutex);
        sem_wait(&emptyBuffer);
        Fill(&buffer);
        i = (i+1)%N;
        sem_post(&fillBuffer);
        sem_post(&mutex);

}
```

Consumer

```
j = 0;

while (1) {

        sem_wait(&mutex);
        sem_wait(&fillBuffer);
        Use(&buffer);
        j = (j+1)%N;
        sem_post(&emptyBuffer);
        sem_post(&mutex);

}
```

Producer/Consumer: Semaphores #3

Final case:

- Multiple producer thread, Multiple consumer thread
- Shared buffer of size N between producer and consumer

Another version with mutex. Will this work? Works

Producer

```
i = 0;

while (1) {

        sem_wait(&emptyBuffer);
        sem_wait(&mutex);
        Fill(&buffer);
        i = (i+1)%N;
        sem_post(&mutex);
        sem_post(& fillBuffer);

}
```

Consumer

```
j = 0;

while (1) {

        sem_wait(&fillBuffer);
        sem_wait(&mutex);
        Use(&buffer);
        j = (j+1)%N;
        sem_post(&mutex);
        sem_post(&emptyBuffer);

}
```

Readers Writers Problem (RWP)

- A data set is shared among a number of concurrent processes/threads
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Problem
  - allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time

RW Problem: Solution structure

The structure of a writer process

```
do {

            ...
 // writing is performed
            ...

} while (true);
```

The structure of a reader process

```
do {

            ...
 // reading is performed
            ...

} while (true);
```

RW Problem: Ensure ME

Writer process

```
do {

  wait(rw_mutex);
          ...
  // writing is performed
          ...
  signal(rw_mutex);

} while (true);
```

Reader process

```
do {

  wait(rw_mutex);
          ...
  // reading is performed
          ...
  signal(rw_mutex);

} while (true);
```

RW Problem: Allow multiple readers

Writer process

```
do {

  wait(rw_mutex);
                ...
  // writing is performed
                ...
  signal(rw_mutex);

} while (true);
```

Reader process

```
do {

   read_count++;
    if (read_count == 1)
          wait(rw_mutex);

               ...
  // reading is performed

   read_count--;
    if (read_count == 0)
          signal(rw_mutex);

 } while (true);
```

RW Problem: Allow multiple readers

Writer process

Writer starves

```
do {

  wait(rw_mutex);
            ...
  // writing is performed
            ...
  signal(rw_mutex);

} while (true);
```

Reader process

```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
        wait(rw_mutex);
   signal(mutex);
            ...
   // reading is performed
   wait(mutex);
   read_count--;
   if (read_count == 0)
        signal(rw_mutex);
   signal(mutex);
} while (true);
```

RW Problem: Starvation free

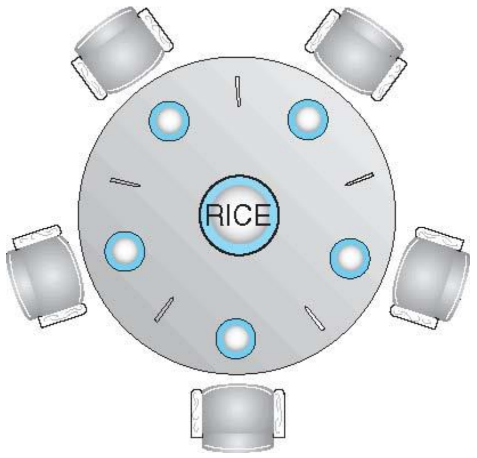
Writer process

```
do {

  wait(rw_line);
  wait(rw_mutex);
  signal(rw_line);
          ...
 // writing is performed
          ...
 signal(rw_mutex);


} while (true);
```

Reader process

```
do {
  wait(rw_line);
  wait(mutex);
  read_count++;
   if (read_count == 1)
        wait(rw_mutex);
  signal(rw_line);
  signal(mutex);
          ...
 // reading is performed
  wait(mutex);
  read_count--;
   if (read_count == 0)
        signal(rw_mutex);
  signal(mutex);
} while (true);
```

Dining Philosophers Problem



- Philosophers spend their lives alternating between thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat
  - Need both to eat, then release both when done

Dining Philosophers - Solution 1

```
sem_t chopstick[5];
```

Philosopher *i*:

```
do {
    wait (chopstick[i]);
    wait (chopstick[(i + 1) % 5]);

            //  eat

    signal (chopstick[i]);
    signal (chopstick[(i + 1) % 5]);

            //  think

} while (TRUE);
```

- What is the problem with this algorithm? **deadlock**

Dining Philosophers Problem

- Deadlock free solution:
  - Allow a philosopher to pick up the chopsticks only if both are available (picking must be done in a critical section).
  - Use an asymmetric solution
    - Odd-numbered philosopher picks up the left chopstick first and then the right chopstick.
    - Even-numbered philosopher picks up the right chopstick first and then the left chopstick.
  - Another asymmetric solution
    - The last philosopher picks up the left chopstick first and then the right chopstick.
    - Other philosophers picks up the right chopstick first and then the left chopstick.

Semaphore Implementation

```
typedef struct{
   int value;
   struct process *q;
   } semaphore;
```

Semaphore Implementation

```
wait(semaphore *S) {
   disable interrupts;
   S->value--;
   if (S->value < 0) {
    add this proc to S->q;
    block();
   enable interrupts;
   }
}
```

```
signal(semaphore *S) {
   disable interrupts;
   S->value++;
   if (S->value < 0) {
    remove proc P from S->q;
    wakeup(p);
   }
   enable interrupts;
}
```

Semaphore Implementation

- Semaphore can be implemented using locks and CVs
  - Homework: Read the book

Disclaimer

- Some of the materials in this lecture slides are from the lecture slides by Prof. Arpaci, Prof. Youjip, and other educators. Thanks to all of them.