Lecture 9

Condition Variables and Semaphores

Previously …

- We have seen how locks may be used to execute critical sections of code with mutual exclusion, so avoiding race conditions
- We have seen how locks may be implemented using:
  - Some hardware support (in the form of a more powerful instruction)
  - Some operating system support and a queue
- We have also seen how to check a condition is true/false before continuing its execution, in spinning wait.

CONDITION VARIABLES

Threads Synchronization

- Lock is not the only primitive needed in concurrent programs
- There are many cases where a thread wishes to check whether a condition is true before continuing its execution
- A parent thread might check whether a child thread has completed before continuing
- There are 2 questions
  - How does a child indicate that it is done?
  - How does a parent wait for a child?

Spinning until ‘true’

```
1   volatile int done = 0;
2
3   void *
4   child(void *arg) {
5       printf("child\n");
[redacted]
7       return NULL;
8   }
9
10  int
11  main(int argc, char *argv[]) {
12      printf("parent: begin\n");
13      pthread_t c;
14      Pthread_create(&c, NULL, child, NULL); // create child
[redacted]
17      printf("parent: end\n");
18      return 0;
19  }
```

Desired output

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

inefficient as the parent spins and wastes CPU time

Condition Variable

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
```

- A condition variable is an explicit queue that threads can put themselves on when some state of execution is not as desired (Waiting on the condition)
- Some other thread, when it changes the said state, can then wake the waiting threads allowing them to continue (Signaling on the condition)

```
1   int done  = 0;
2   pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3   pthread_cond_t c  = PTHREAD_COND_INITIALIZER;
4
5   void thr_exit() {
6       Pthread_mutex_lock(&m);
7       done = 1;
8       Pthread_cond_signal(&c);
9       mutex_unlock(&m);
10  }
11
12  void *child(void *arg) {
13      printf("child\n");
14      thr_exit();
15      return NULL;
16  }
17
18  void thr_join() {
19      Pthread_mutex_lock(&m);
20      while (done == 0)
21          Pthread_cond_wait(&c, &m);
22      Pthread_mutex_unlock(&m);
23  }
24
25  int main(int argc, char *argv[]) {
26      printf("parent: begin\n");
27      pthread_t p;
28      Pthread_create(&p, NULL, child, NULL);
29      thr_join();
```

CODING TIP: ALWAYS HOLD THE LOCK WHILE SIGNALING

Note

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  init = PTHREAD_COND_INITIALIZER;

Pthread_mutex_lock(&lock);
while (initialized == 0)
    Pthread_cond_wait(&init, &lock);
Pthread_mutex_unlock(&lock);
```

```
Pthread_mutex_lock(&lock);
initialized = 1;
Pthread_cond_signal(&init);
Pthread_mutex_unlock(&lock);
```

- A thread checks to see if the variable initialized has yet been set to something other than zero
- If not, the thread simply calls the wait routine in order to sleep until some other thread wakes it
- The wait call, in addition to putting the calling thread to sleep, releases the lock
- When signaling (as well as when modifying the global variable initialized), we always make sure that we hold the lock

After Wait()

- wait() takes a mutex as a parameter; it assumes that this mutex is locked when wait() is called.
- wait() will
  - Release the lock and put the calling thread to sleep
  - When the thread wakes up (after some other thread has signaled it) …
  - it must re-acquire the lock before returning to the caller

Producer/Consumer

- One or more Producer threads and one or more Consumer threads
  - Producers produce data items and wish to put them in a buffer
  - Consumers get data items out of the buffer and consume the data in some way
- Examples:
  - A multi-threaded web server – a producer puts requests into the buffer and consumer threads get and process them
  - Unix pipe: grep foo file.txt | wc –l (grep is the producer and wc the consumer)
- The bounded buffer is a shared resource and requires synchronized access to it

Producer and Consumer Code

```
1   void *producer(void *arg) {
2       int i;
3       int loops = (int) arg;
4       for (i = 0; i < loops; i++) {
5           put(i);
6       }
7   }
8
9   void *consumer(void *arg) {
10      int i;
11      while (1) {
12          int tmp = get();
13          printf("%d\n", tmp);
14      }
15  }
```

puts an integer into the shared buffer 'loops' number of times

gets the data out of that shared buffer (forever)

This Buffer (Just a Single *int*) with basic actions: *Put* and *Get*

```
1   int buffer;
2   int count = 0; // initially, empty
3
4   void put(int value) {
5       assert(count == 0);
6       count = 1;
7       buffer = value;
8   }
9
10  int get() {
11      assert(count == 1);
12      count = 0;
13      return buffer;
14  }
```

Assume buffer is empty

Mark it as ‘full’

Take the data and make the buffer empty again

```
1   int buffer;
2   int count = 0; // initially, empty
3
4   void put(int value) {
5       assert(count == 0);
6       count = 1;
7       buffer = value;
8   }
9
10  int get() {
11      assert(count == 1);
12      count = 0;
13      return buffer;
14  }
```

```
1   void *producer(void *arg) {
2       int i;
3       int loops = (int) arg;
4       for (i = 0; i < loops; i++) {
5           put(i);
6       }
7   }
8
9   void *consumer(void *arg) {
10      int i;
11      while (1) {
12          int tmp = get();
13          printf("%d\n", tmp);
14      }
15  }
```

Need Condition Variables

- The put() and get() functions have critical sections within them
  - put() updates the buffer and get() reads from it
  - Eg. two producers might try to update the buffer at the same time and an item could be lost
  - Race conditions must be prevented
- But putting a lock around the critical section alone does not work
- We also need condition variables
- Let's see how …

```
1   void *producer(void *arg) {
2       int i;
3       int loops = (int) arg;
4       for (i = 0; i < loops; i++) {
5           put(i);
6       }
7   }
8
9   void *consumer(void *arg) {
10      int i;
11      while (1) {
12          int tmp = get();
13          printf("%d\n", tmp);
14      }
15  }
```

```
1   cond_t  cond;
2   mutex_t mutex;
3
4   void *producer(void *arg) {
5       int i;
6       for (i = 0; i < loops; i++) {
7           mutex_lock(&mutex);
8           while (count == 1)
9               cond_wait(&cond, &mutex);
10          put(i);
11          cond_signal(&cond);
12          mutex_unlock(&mutex);
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          mutex_lock(&mutex);
20          while (count == 0)
21              cond_wait(&cond, &mutex);
22          int tmp = get();
23          cond_signal(&cond);
24          mutex_unlock(&mutex);
25          printf("%d\n", tmp);
26      }
27  }
```

```
1   cond_t  cond;
2   mutex_t mutex;
3
4   void *producer(void *arg) {
5       int i;
6       for (i = 0; i < loops; i++) {
7           mutex_lock(&mutex);
8           while (count == 1)
9               cond_wait(&cond, &mutex);
10          put(i);
11          cond_signal(&cond);
12          mutex_unlock(&mutex);
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          mutex_lock(&mutex);
20          while (count == 0)
21              cond_wait(&cond, &mutex);
22          int tmp = get();
23          cond_signal(&cond);
24          mutex_unlock(&mutex);
25          printf("%d\n", tmp);
26      }
27  }
```

Scenario with
two consumers (Tc1 and Tc2)
and a producer (Tp)

If Tc1 and Tc2 run first, what might be a possible sequence of events?

Problem

- What is the problem with this code?
  - Suppose the two consumers run first (Tc1 and Tc2), and both go to sleep (line 21) as the buffer is empty
  - Then, the producer runs, puts a value in the buffer, wakes one of the consumers (say Tc1), and goes to sleep
  - Tc1 then runs (returning from wait() at line 21), re-checks the condition (line 20), finds the buffer is full, and gets the value (line 22)
  - This consumer then signals on the condition, waking one of the threads that is sleeping
  - However, which thread should be woken?

This is a Bug

- All three threads are left sleeping
  - Because the consumer has emptied the buffer, it clearly should wake the producer
  - However, if it wakes the consumer Tc2 (definitely possible depending on how the wait queue is managed), we have a problem
  - Specifically, the consumer Tc2 will wake up and find the buffer empty (line 20), and go back to sleep (line 21)
  - The producer Tp, which has a value to put into the buffer, is left sleeping
  - The other consumer thread, Tc1, also goes back to sleep

```
1   cond_t  empty, fill;
2   mutex_t mutex;
3
4   void *producer(void *arg) {
5       int i;
6       for (i = 0; i < loops; i++) {
7           mutex_lock(&mutex);
8           while (count == 1)
9               cond_wait(&empty, &mutex);
10          put(i);
11          cond_signal(&fill);
12          mutex_unlock(&mutex);
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          mutex_lock(&mutex);
20          while (count == 0)
21              cond_wait(&fill, &mutex);
22          int tmp = get();
23          cond_signal(&empty);
24          mutex_unlock(&mutex);
25          printf("%d\n", tmp);
26      }
27  }
```

Solution

Use two condition variables in order to properly signal which type of thread should wake up when the state of the system changes

Use While (not If)

- Why do we use while and not if to check the condition?
  - Suppose we have two consumers (Tc1 and Tc2) as before
  - Tc1 runs and goes to sleep (line 21) as the buffer is empty
  - Tc2 is ready, but does not yet run
  - Then, the producer runs, puts a value in the buffer, wakes Tc1, and goes to sleep
  - Before Tc1 returns from the wait() and reacquires the lock, Tc2 runs, finds an item in the buffer and consumes it
  - Then Tc1 returns from the wait() at line 21 – using if would allow it to get an item from an empty buffer

Using while is always correct – using if does not always work

Final Put and Get

```
1   int buffer[MAX];
2   int fill  = 0;
3   int use   = 0;
4   int count = 0;
5
6   void put(int value) {
7       buffer[fill] = value;
8       fill = (fill + 1) % MAX;
9       count++;
10  }
11
12  int get() {
13      int tmp = buffer[use];
14      use = (use + 1) % MAX;
15      count--;
16      return tmp;
17  }
```

Circular Buffer

The single int has been extended into a *circular* buffer of size MAX

- fill is the input pointer
- use is the output pointer

Final Producer/ Consumer

```
1   cond_t empty, fill;
2   mutex_t mutex;
3
4   void *producer(void *arg) {
5       int i;
6       for (i = 0; i < loops; i++) {
7           mutex_lock(&mutex);
8           while (count == MAX)
9               cond_wait(&empty, &mutex);
10          put(i);
11          cond_signal(&fill);
12          mutex_unlock(&mutex);
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          mutex_lock(&mutex);
20          while (count == 0)
21              cond_wait(&fill, &mutex);
22          int tmp = get();
23          cond_signal(&empty);
24          mutex_unlock(&mutex);
25          printf("%d\n", tmp);
26      }
27  }
```

SEMAPHORES

Introduction

- We have seen that we need both locks and condition variables to solve a broad range of relevant and interesting concurrency problems
  - Locks can be used to ensure mutual exclusion
  - Condition variables can be used for synchronize
- Now … we can actually use **semaphores** as both locks and condition variables

Definition

- A semaphore is as an object with an integer value that we can manipulate with two routines
- In the POSIX standard, these are called
  - sem_wait()
  - sem_post()
- The initial value of the semaphore determines its behavior
  - sem_init() is also needed

How it Works

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

0 indicates semaphore shared between threads in same process

s is initialized to a value of 1

Returns immediately or causes the caller to wait for a subsequent post

```
int sem_wait(sem_t *s) {
    wait until value of semaphore s is greater than 0
    decrement the value of semaphore s by 1
}
```

Does not ever suspend the caller, just wakes a waiting thread

```
int sem_post(sem_t *s) {
    increment the value of semaphore s by 1
    if there are 1 or more threads waiting, wake 1
}
```

Binary Semaphores

- Binary Semaphore can be used 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);
```

- Two threads are trying to enter their critical sections
  - Suppose one thread holds the lock, i.e. it has called sem wait(), but not yet called sem post()
  - If another thread tries to enter the critical section it will find that the value of the semaphore is 0, and thus wait

Semaphore as Condition Variable

- Semaphores are also useful when a thread wants to halt its own progress waiting for something to change

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

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

What should the initial value of this semaphore be?

```
sem_t s;

void *
child(void *arg) {
    printf("child\n");
    // signal here: child is done
    sem_post(&s);
    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);
    // wait here for child
    sem_wait(&s);
    printf("parent: end\n");
    return 0;
}
```

Parent waits for its child to complete

Semaphore on Producer/Consumer

```
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 buffers are empty to begin with.
    sem_init(&full, 0, 0);    // ... and 0 are full
    // ...
}
```

Introduces Two Semaphores, *empty* and *full*

- Suppose first we have a single int buffer (i.e. MAX = 1)
- Assume the consumer gets to run first – thus, the consumer will call sem_wait(&full)
- Because full was initialized to the value 0, the call will block the consumer and wait for another thread to call sem_post() on the semaphore
- Assume the producer then runs – it will call sem wait(&empty), but will continue, because empty was initialized to the value MAX (in this case, 1)
- Thus, empty will be decremented to 0 and the producer will put a data value into the buffer
- The producer will then call sem post(&full), changing the value of the full semaphore from 0 to 1 and waking the consumer

Producer/Consumer Problem

- We now have a race condition

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

void *consumer(void *arg) {
    int i, tmp = 0;
    while (tmp != -1) {
        sem_wait(&full);
        tmp = get();
        sem_post(&empty);
        printf("%d\n", tmp);
    }
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with.
    sem_init(&full, 0, 0);    // ... and 0 are full
    // ...
}
```

Producer waits for a buffer slot to become empty in order to put data into it

put(i) and get() are done without locks
Race condition may occur

Consumer similarly waits for a buffer slot to become filled before using it

```
// line c1
// line c2
// line c3
```

How can locks be added ?

Add Mutex Locks

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

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&full);
        int tmp = get();
        sem_post(&empty);
        sem_post(&mutex);
        printf("%d\n", tmp);
    }
}
```



Think of a scenario that may cause deadlock from this code

How do we fix it ?

Deadlock Occur

- Two threads are one producer (Tp1) and one consumer (Tc1)
- Tc1 gets to run first – it acquires the mutex , and then calls sem wait() on the full semaphore
- Because there is no data yet, this call causes Tc1 to sleep, but the consumer still holds the lock
- Tp1 then runs – unfortunately, the first thing it does is call sem wait() on the binary mutex semaphore and waits because the lock is already held
- There is a simple cycle here – the consumer holds the mutex and is waiting for the someone to signal full
- The producer could signal full but is waiting for the mutex – thus, the producer and consumer are each stuck waiting for each other: a classic deadlock

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        sem_wait(&mutex);
        put(i);
        sem_post(&mutex);
        sem_post(&full);
    }
}
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);
        sem_wait(&mutex);
        int tmp = get();
        sem_post(&mutex);
        sem_post(&empty);
        printf("%d\n", tmp);
    }
}
```

Solution – Reduce the Scope of the Lock

How to Implement a Semaphore

```
typedef struct __Sem_t {
    int value;
    pthread_cond_t cond;
    pthread_mutex_t lock;
} Sem_t;

// only one thread can call this
void Sem_init(Sem_t *s, int value) {
    s->value = value;
    Cond_init(&s->cond);
    Mutex_init(&s->lock);
}
```

```
void Sem_wait(Sem_t *s) {
    Mutex_lock(&s->lock);
    while (s->value <= 0)
        Cond_wait(&s->cond, &s->lock);
    s->value--;
    Mutex_unlock(&s->lock);
}

void Sem_post(Sem_t *s) {
    Mutex_lock(&s->lock);
    s->value++;
    Cond_signal(&s->cond);
    Mutex_unlock(&s->lock);
}
```