Recap

Lock Implementation

- TestAndSet-based implementation for mutual exclusion, efficiency and fairness

Condition Variable

- Working based on queue
- Wait and signal operations
- Application for thread_join/thread_exit

Classic Concurrency Problems

Concurrency makes reasoning about code very difficult

There are several classic (example) problems that have been created to illustrate issues in concurrency and common solutions

It is a good idea to study the solutions to these problems and use them as patterns in your own code

The Producer/Consumer (Bounded Buffer) Problem

One or more producer threads generate data items and place them in a buffer

One or more consumers grab items from the buffer and consume them

Common example: a pipe between processes acts as a buffer with concurrent producer and consumer

```
void *producer(void *arg) {
    int i;
    int loops = (int) arg;
    for (i = 0; i < loops; i++) {
        put(i);
    }
}

void *consumer(void *arg) {
    while (1) {
        int tmp = get();
        printf("%d\n", tmp);
    }
}
```

A non-thread-safe version

Attempt 1 (Broken)

Need some way to prevent calling put of full buffer and get on empty buffer

This attempt is broken

To understand why, need to know one more detail about condition variables…

```
int loops; // must initialize somewhere...
cond_t  cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // p1
        if (count == 1)                        // p2
            Pthread_cond_wait(&cond, &mutex); // p3
        put(i);                                // p4
        Pthread_cond_signal(&cond);            // p5
        Pthread_mutex_unlock(&mutex);          // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // c1
        if (count == 0)                        // c2
            Pthread_cond_wait(&cond, &mutex); // c3
        int tmp = get();                       // c4
        Pthread_cond_signal(&cond);            // c5
        Pthread_mutex_unlock(&mutex);          // c6
        printf("%d\n", tmp);
    }
}
```

Mesa Semantics for Condition Variables

The most common implementations of condition variables (including pthreads) follow **mesa semantics**

When a sleeping (waiting) thread is woken, it is placed in a ready queue

It does not require the mutex lock to be in the queue

Only when it is its turn to run does it acquire the lock

Attempt 1 (Broken)

Example of where Mesa semantics can lead to issues

Consider 2 consumers and 1 producer

- Consumer1 waits for condition
- Producer signals which puts consumer1 in ready queue
- Before consumer1 runs consumer2 consumes buffer
- Comsumer1 is set to running and tries to consume from empty buffer (error)

```
int loops; // must initialize somewhere...
cond_t  cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);                 // p1
        if (count == 1)                             // p2
            Pthread_cond_wait(&cond, &mutex);       // p3
        put(i);                                     // p4
        Pthread_cond_signal(&cond);                 // p5
        Pthread_mutex_unlock(&mutex);               // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);                 // c1
        if (count == 0)                             // c2
            Pthread_cond_wait(&cond, &mutex);       // c3
        int tmp = get();                            // c4
        Pthread_cond_signal(&cond);                 // c5
        Pthread_mutex_unlock(&mutex);               // c6
        printf("%d\n", tmp);
    }
}
```

Attempt 2 (Better, Still Broken)

Consider 2 consumers and 1 producer

What can go wrong?

```
int loops;
cond_t  cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);                 // p1
        while (count == 1)                          // p2
            Pthread_cond_wait(&cond, &mutex);       // p3
        put(i);                                     // p4
        Pthread_cond_signal(&cond);                 // p5
        Pthread_mutex_unlock(&mutex);               // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);                 // c1
        while (count == 0)                          // c2
            Pthread_cond_wait(&cond, &mutex);       // c3
        int tmp = get();                            // c4
        Pthread_cond_signal(&cond);                 // c5
        Pthread_mutex_unlock(&mutex);               // c6
        printf("%d\n", tmp);
    }
}
```

Attempt 2 (Better, Still Broken)

Consider 2 consumers and 1 producer

Consumer1 and Consumer2 both go into wait

Producer adds to buffer and waits

Consumer1 consumes from buffer and signals

Consumer2 gets the signal, but it should be a producer not another customer to get it

```
int loops;
cond_t  cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);                  // p1
        while (count == 1)                           // p2
            Pthread_cond_wait(&cond, &mutex);        // p3
        put(i);                                      // p4
        Pthread_cond_signal(&cond);                  // p5
        Pthread_mutex_unlock(&mutex);                // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);                  // c1
        while (count == 0)                           // c2
            Pthread_cond_wait(&cond, &mutex);        // c3
        int tmp = get();                             // c4
        Pthread_cond_signal(&cond);                  // c5
        Pthread_mutex_unlock(&mutex);                // c6
        printf("%d\n", tmp);
    }
}
```

Working Version

Need condition variables to signal both empty and full

```
cond_t  empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 1)
            Pthread_cond_wait(&empty, &mutex);
        put(i);
        Pthread_cond_signal(&fill);
        Pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 0)
            Pthread_cond_wait(&fill, &mutex);
        int tmp = get();
        Pthread_cond_signal(&empty);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

Semaphores

(based on Ch. 31)

Slides revised from
Matthew Tancreti
Iowa State University

Semaphore

We have seen examples of how locks and condition variables are used together

While working on a new OS in the 1960s, Edsger Dijkstra saw these useage patterns and created the **semaphore** – a variable that provides both locking and signaling

How to provide locking and signaling in a single abstraction?

Semaphore

Semaphores restrict how many threads have access to a resource at any time, by keeping a count of the resources available

```
#include <semaphore.h>
sem_t s;

sem_init(&s, pshared, 5);
```

Must always initialize a semaphore with the number of resources available

Indicates 5 resources available

```
/* decrease the value of s by 1
 * if s < 0, wait (sleep) until woken up
 */
sem_wait(&s);
```

Try to get a resource, if none available, wait until one is

```
/* increment the value of s by 1
 * if there is one or more threads waiting, wake one
 */
sem_post(&s);
```

Give up resource, signal one resource is available

Using Semaphores as Locks

To use a semaphore like a mutex lock

- Initialize s to 1
- Acquire lock with sem_wait(&s)
- Release lock with sem_post(&s)

Only one thread will be allowed into critical section

Because semaphore can be interpreted as being in one of two states, locked or unlocked, this usage is called a **binary semaphore**

Using a Semaphore to Signal

Sometimes it is useful to initialize a semaphore to 0

For example, one thread waits for another to send a signal

```
sem_t s;

void *
child(void *arg) {
  printf(”child\n”);
  sem_post(&s);
  return NULL;
}

int
Main() {
  sem_init(%s, 0, 0);
  printf("parent: being");
  pthread_t c;
  pthread_create(&c, NULL, child, NULL);
  sem_wait(&s); // wait here for child
  printf("parent: end\n”);
  return 0;
}
```

Producer/Consumer (Bounded Buffer) Problem with Semaphores

Empty represents the number of empty buffer “spaces”, when 0 the producer must wait for a space to empty out

Full represents the number of filled buffer “spaces”, when 0 the consumer must wait for a space to fill up

Filled one space

Emptied one space

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);           // Line P1
        sem_wait(&mutex);           // Line P1.5 (MUTEX HERE)
        put(i);                     // Line P2
        sem_post(&mutex);           // Line P2.5 (AND 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);           // Line C1.5 (MUTEX HERE)
        int tmp = get();            // Line C2
        sem_post(&mutex);           // Line C2.5 (AND HERE)
        sem_post(&empty);           // Line C3
        printf("%d\n", tmp);
    }
}
```

Classic Problem: Readers-Writers Problem

A data set is shared among multiple threads

- **Readers** – only read the data set
- **Writers** – can both read and write

Solution must satisfy the following:

1. Only one writer can perform writing at any time
2. Reading is not allowed while a writer is writing
3. Many readers can perform reading concurrently

Solution Setup

Shared variables:

- Semaphore `rw_mutex` initialized to 1
  - Used by both readers and writers to ensure that the writers have exclusive access to the shared data set
- Integer `read_count` counts the number of readers that are currently reading, initialized to 0
- Semaphore `mutex` initialized to 1
  - Used by readers to ensure mutual exclusion when `read_count` is updated

Readers-Writers Problem Solution

The structure of a writer process

```
while (true) {
  sem_wait(&rw_mutex);
      ...
  /* writing is performed */
      ...
  sem_post(&rw_mutex);
}
```

The structure of a reader process

```
while (true) {
  sem_wait(&mutex);
  read_count++;
  if (read_count == 1)
     sem_wait(&rw_mutex);
  sem_post(&mutex);
     ...
  /* reading is performed */
     ...
  sem_wait(&mutex);
  read_count--;
  if (read_count == 0)
     sem_post(&rw_mutex);
  sem_post(&mutex);
}
```

First reader in grabs rw_mutex

Last reader out releases rw_mutex

Reader-Writers Variations

In previous solution, no reader is kept waiting unless a writer is writing

- When readers have the `rw_mutex` they block any writers

An alternative solution is if a writer is waiting for the `rw_mutex`, no new readers may start reading

- When writer wants to run it gets priority over new readers

Both solutions have issue of starvation

- Version 1: readers can starve a writer
- Version 2: writers can starve a reader