Condition Variable

- Without condition variables,
  - Threads continually poll to check if the condition is met
  - Busy waiting!

- With condition variables
  - Same goal without polling
Inside a condition variable

```c
struct pthread_cond {
    int waiting;
    handle_t semaphore;
};
```

Number of threads waiting on the condition variable

A semaphore for synchronization
Inside a condition variable

```c
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex) {
    atomic_increment(&cond->waiting);
    pthread_mutex_unlock(mutex);
    if (wait(cond->semaphore, INFINITE) < 0)
        return errno;
    atomic_decrement(&cond->waiting);
    pthread_mutex_lock(mutex);
    return 0;
}
```

- `pthread_mutex_lock` is always called before `pthread_cond_wait` to acquire lock.
- Thread always has lock when returning from `pthread_cond_wait`.
More Complex Example

- Master thread
  - Spawns a number of concurrent slaves
  - Waits until all of the slaves have finished to exit
  - Tracks current number of slaves executing

- A mutex is associated with count and a condition variable with the mutex
Example

```c
#include <stdio.h>
#include <pthread.h>

#define NO_OF_PROCS  4

typedef struct __SharedType {
    int count;    /* number of active slaves */
    pthread_mutex_t lock; /* mutex for count */
    pthread_cond_t done; /* sig. by finished slave */
} SharedType, *SharedType_ptr;

SharedType_ptr shared_data;
```
Example: Main

```c
main(int argc, char **argv) {
    int res;

    /* allocate shared data */
    if ((sh_data = (SharedType *) malloc(sizeof(SharedType))) == NULL) {
        exit(1);
    }

    sh_data->count = 0;

    /* allocate mutex */
    if ((res = pthread_mutex_init(&sh_data->lock, NULL)) != 0) {
        exit(1);
    }

    /* allocate condition var */
    if ((res = pthread_cond_init(&sh_data->done, NULL)) != 0) {
        exit(1);
    }

    /* generate number of slaves to create */
    srandom(0);
    /* create up to 15 slaves */
    master((int) random() % 16);
}
```
Example: Main

```c
main(int argc, char **argv) {
    int res;
    /* allocate shared data */
    if ((sh_data = (SharedType *)
        malloc(sizeof(SharedType))) ==
        NULL) {
        exit(1);
    }
    sh_data->count = 0;

    pthread_mutex_t data_mutex =
    PTHREAD_MUTEX_INITIALIZER;

    pthread_cont_t data_cond =
    PTHREAD_COND_INITIALIZER;

    /* generate number of slaves
     * to create */
    srand(0);
    /* create up to 15 slaves */
    master((int) random() % 16);
}
```
Example: Master

```c
master(int nslaves) {
    int i;
    pthread_t id;
    for (i = 1; i <= nslaves; i += 1) {
        pthread_mutex_lock(&sh_data->lock);
        /* start slave and detach */
        shared_data->count += 1;
        pthread_create(&id, NULL,
                       (void* (*)(void *))slave,
                       (void *)sh_data);
        pthread_mutex_unlock(&sh_data->lock);
    }
    pthread_mutex_lock(&sh_data->lock);
    while (sh_data->count != 0)
        pthread_cond_wait(&sh_data->done, &sh_data->lock);
    pthread_mutex_unlock(&sh_data->lock);
    printf("All %d slaves have finished.\n", nslaves);
    pthread_exit(0);
}
```
Example: Slave

```c
void slave(void *shared) {
    int i, n;
    sh_data = shared;
    printf("Slave.\n", n);
    n = random() % 1000;

    for (i = 0; i < n; i+= 1)
        Sleep(10);

    /* mutex for shared data */
    pthread_mutex_lock(&sh_data->lock);

    /* dec number of slaves */
    sh_data->count -= 1;

    /* done running */
    printf("Slave finished %d cycles.\n", n);

    /* signal that you are done working */
    pthread_cond_signal(&sh_data->done);

    /* release mutex for shared data */
    pthread_mutex_unlock(&sh_data->lock);
}
```
Semaphores vs. CVs

Semaphore
- Integer value (>=0)
- Wait does not always block
- Signal either releases thread or inc’s counter
- If signal releases thread, both threads continue afterwards

Condition Variables
- No integer value
- Wait always blocks
- Signal either releases thread or is lost
- If signal releases thread, only one of them continue
Classical Synchronization Problems
This lecture

Goals
- Introduce classical synchronization problems

Topics
- Producer-Consumer Problem
- Reader-Writer Problem
- Dining Philosophers Problem
- Sleeping Barber’s Problem
Chefs cook items and put them on a conveyer belt.

Waiters pick items off the belt.
Now imagine many chefs!

And many waiters!
A potential mess!
Producers insert items

Consumers remove items

Shared resource: bounded buffer

- Efficient implementation: circular buffer with an insert and a removal pointer
Producer-Consumer

Chef = Producer
Waiter = Consumer
Producer-Consumer

Chef = Producer
Waiter = Consumer

What does the chef do with a new pizza?

Where does the waiter take a pizza from?

Copyright ©: University of Illinois CS 241 Staff
Producer-Consumer

Chef = Producer
Waiter = Consumer

Insert pizza

insertPtr

removePtr
Producer-Consumer

Chef = Producer
Waiter = Consumer

insertPtr

removePtr

Insert pizza
Producer-Consumer

Chef = Producer
Waiter = Consumer

Insert pizza
Producer-Consumer

Chef = Producer
Waiter = Consumer

removePtr
insertPtr

Remove pizza
Producer-Consumer

Chef = Producer
Waiter = Consumer

Insert pizza

insertPtr
removePtr
Producer-Consumer

Chef
Waiter

= Producer
= Consumer

Insert pizza

insertPtr
removePtr

Copyright ©: University of Illinois CS 241 Staff

Producer-Consumer

Chef = Producer
Waiter = Consumer

BUFFER FULL: Producer must be blocked!

Insert pizza

insertPtr
removePtr
Producer-Consumer

Chef = Producer
Waiter = Consumer

removePtr
insertPtr

Remove pizza
Producer-Consumer

Chef = Producer
Waiter = Consumer

removePtr

insertPtr

Remove pizza
Producer-Consumer

Chef
Waiter

= Producer
= Consumer

removePtr
insertPtr

Remove pizza
Producer-Consumer

Chef = Producer
Waiter = Consumer

insertPtr

removePtr

Remove pizza
Producer-Consumer

Chef = Producer
Waiter = Consumer

Remove pizza

insertPtr
removePtr
Producer-Consumer

Chef = Producer
Waiter = Consumer

removePtr
insertPtr

Remove pizza

Copyright ©: University of Illinois CS 241 Staff
Producer-Consumer

Chef = Producer
Waiter = Consumer

insertPtr
removePtr

Remove pizza
Producer-Consumer

Chef = Producer
Waiter = Consumer

BUFFER EMPTY: Consumer must be blocked!

removePtr
insertPtr

Remove pizza

Copyright ©: University of Illinois CS 241 Staff
Producer-Consumer Summary

- **Producer**
  - Insert items
  - Update insertion pointer

- **Consumer**
  - Execute destructive read on the buffer
  - Update removal pointer

- **Both**
  - Update information about how full/empty the buffer is

- **Solution**
  - Must allow multiple producers and consumers
Challenges

- Prevent buffer overflow
- Prevent buffer underflow
- Mutual exclusion when modifying the buffer data structure
Solutions

- Prevent buffer overflow
  - Block producer when full
  - Counting semaphore to count #free slots
  - 0 ➔ block producer

- Prevent buffer underflow

- Mutual exclusion when modifying the buffer data structure
Solutions

- Prevent buffer overflow
  - Block producer when full
  - Counting semaphore to count #free slots
  - 0 → block producer

- Prevent buffer underflow
  - Block consumer when empty
  - Counting semaphore to count #items in buffer
  - 0 → block consumer

- Mutual exclusion when modifying the buffer data structure
Solutions

- Prevent buffer overflow
  - Block producer when full
  - **Counting semaphore** to count #free slots
  - 0 ➔ block producer

- Prevent buffer underflow
  - Block consumer when empty
  - **Counting semaphore** to count #items in buffer
  - 0 ➔ block consumer

- Mutual exclusion when modifying the buffer data structure
  - **Mutex** protects shared buffer & pointers
Assembling the solution

- **Producer**
  - `sem_wait(slots), sem_signal(slots)`
  - Initialize `slots` to `N`

- **Consumer**
  - `sem_wait(items), sem_signal(items)`
  - Initialize semaphore `items` to `0`

- **Synchronization**
  - `mutex_lock(m), mutex_unlock(m)`

- **Buffer management**
  - `insertptr = (insertptr+1) % N`
  - `removalptr = (removalptr+1) % N`
Readers-Writers Problem

Shared Resource
Readers-Writers Problem
Readers-Writers Problem

Shared Resource
II. Reader-Writer Problem

- Readers read data
- Writers write data
- Rules
  - Multiple readers may read the data simultaneously
  - Only one writer can write the data at any time
  - A reader and a writer cannot access data simultaneously
- Locking table
  - Whether any two can be in the critical section simultaneously

<table>
<thead>
<tr>
<th></th>
<th>Reader</th>
<th>Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reader</td>
<td>OK</td>
<td>No</td>
</tr>
<tr>
<td>Writer</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Reader-Writer: First Solution

```
reader() {
    while(TRUE) {
        <other stuff>;
        sem_wait(mutex);
        readCount++;
        if(readCount == 1)
            sem_wait(writeBlock);
        sem_signal(mutex);
        /* Critical section */
        access(resource);
        sem_wait(mutex);
        readCount--;
        if(readCount == 0)
            sem_signal(writeBlock);
        sem_post(mutex);
    }
}

writer() {
    while(TRUE) {
        <other computing>;
        sem_wait(writeBlock);
        /* Critical section */
        access(resource);
        sem_signal(writeBlock);
    }
}
```

```
int readCount = 0;
semaphore mutex = 1;
semaphore writeBlock = 1;
```
int readCount=0, writeCount=0;
semaphore mutex1=1, mutex2=1;
Semaphore readBlock=1,writeBlock=1

reader() {
    while(TRUE) {
        <other computing>;
        sem_wait(readBlock);
        sem_wait(mutex1);
        readCount++;
        if(readCount == 1)
            sem_wait(writeBlock);
        sem_signal(mutex1);
        sem_signal(readBlock);
        access(resource);
        sem_wait(mutex1);
        readCount--;
        if(readCount == 0)
            sem_signal(writeBlock);
        sem_signal(mutex1);
    }
}

writer() {
    while(TRUE) {
        <other computing>;
        sem_wait(mutex2);
        writeCount++;
        if(writeCount == 1)
            sem_wait(readBlock);
        sem_signal(mutex2);
        sem_wait(writeBlock);
        access(resource);
        sem_wait(mutex2);
        writeCount--;
        if(writeCount == 0)
            sem_signal(readBlock);
        sem_signal(mutex2);
    }
}
Better R-W solution idea

- Idea: serve requests in order
  - Once a writer requests access, any entering readers have to block until the writer is done

- Advantage?
- Disadvantage?
Reader-Writer: Fairer Solution?

```c
int readCount = 0, writeCount = 0;
semaphore mutex1 = 1, mutex2 = 1;
semaphore readBlock = 1, writeBlock = 1, writePending = 1;

reader() {
    while(TRUE) {
        <other computing>;
        sem_wait(writePending);
        sem_wait(readBlock);
        sem_wait(mutex1);
        readCount++;
        if(readCount == 1)
            sem_wait(writeBlock);
        sem_signal(mutex1);
        sem_signal(readBlock);
        sem_signal(writePending);
        access(resource);
        sem_wait(mutex1);
        readCount--;
        if(readCount == 0)
            sem_signal(writeBlock);
        sem_signal(mutex1);
    }
}

writer() {
    while(TRUE) {
        <other computing>;
        sem_wait(writePending);
        sem_wait(mutex2);
        writeCount++;
        if(writeCount == 1)
            sem_wait(readBlock);
        sem_signal(mutex2);
        sem_wait(writeBlock);
        access(resource);
        sem_wait(mutex1);
        writeCount--;
        if(writeCount == 0)
            sem_signal(readBlock);
        sem_signal(mutex2);
    }
}
```
Summary

- Classic synchronization problems
  - Producer-Consumer Problem
  - Reader-Writer Problem

- Saved for next time:
  - Sleeping Barber’s Problem
  - Dining Philosophers Problem
Dining Philosophers

- N philosophers and N forks
  - Philosophers eat/think
  - Eating needs 2 forks
  - Pick one fork at a time