Clarifications and Corrections

- Response Time
  - Time from job submission until it starts running for the first time
- Waiting Time
  - Total time that the job is not running but queued
- Turnaround Time
  - Time between job submission and completion

Semaphores, Mutexes and Condition Variables

Synchronization Primatives

- Counting Semaphores
  - Permit a limited number of threads to execute a section of the code
- Mutexes
  - Permit only one thread to execute a section of the code
- Condition Variables
  - Communicate information about the state of shared data

Counting Semaphores

- Before entering critical section
  - **`semWait(s)`**
  - Wait until value is > 0, then decrement
- After finishing critical section
  - **`semSignal(s)`**
  - Increment value
- Implementation requirements
  - **`semSignal`** and **`semWait`** must be atomic

```
semaphore s = 1;
Pi {
  while(1) {
    semWait(s);
    <Critical Section>
    semSignal(s);
    <Other work>
  }
}
```

POSIX Semaphores

- Data type
  - Semaphore is a variable of type **`sem_t`**
  - Include **`<semaphore.h>`**
- Atomic Operations

use **`pshared==0`**

```
int sem_init(sem_t *sem, int pshared,
   unsigned value);
int sem_destroy(sem_t *sem);
int sem_post(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_wait(sem_t *sem);
```

Example: bank balance

- Shared variable: **`balance`**
- Protected by semaphore when used in:
  - **`decshared`**
    - Decrements **`balance`**
  - **`incshared`**
    - Increments **`balance`**

Example: bank balance

```
#include <errno.h>
#include <semaphore.h>

static int balance = 0;
static sem_t bal_sem;

int initshared(int val) {
   if (sem_init(&bal_sem, 0, 1) == -1)
      return -1;
   balance = val;
   return 0;
}
```

pshared

value

Example: bank balance

```
int decshared() {
  while (sem_wait(&bal_sem)
             == -1)
    if (errno != EINTR)
      return -1;
  balance--;
  return sem_post(&bal_sem);
}
```

```
int incshared() {
  while (sem_wait(&bal_sem)
             == -1)
    if (errno != EINTR)
      return -1;
  balance++;
  return sem_post(&bal_sem);
}
```

Which one is going first?

Pthread Synchronization

- Mutex
  - Semaphore with maximum value 1
  - Simple and efficient
  - Locked: some thread holds the mutex
  - Unlocked: no thread holds the mutex
  - When several threads compete
    - One wins
    - The rest block
      - Queue of blocked threads

Mutex Variables

- A typical sequence in the use of a mutex
  - Create and initialize a mutex variable
  - Several threads attempt to lock the mutex
  - Only one succeeds and now owns the mutex
  - The owner performs some set of actions
  - The owner unlocks the mutex
  - Another thread acquires the mutex and repeats the process
  - Finally the mutex is destroyed

Creating a mutex

```
#include <pthread.h>
int int pthread_mutex_init(pthread_mutex_t
    *mutex, const pthread_mutexattr_t *attr);
```

- Initialize a pthread mutex: the mutex is initially unlocked
- Returns
  - 0 on success
  - Error number on failure
    - **EAGAIN:** The system lacked the necessary resources; **ENOMEM:** Insufficient memory ; **EPERM:** Caller does not have privileges; **EBUSY:** An attempt to re-initialise a mutex; **EINVAL:** The value specified by attr is invalid
- Parameters
  - **mutex**: Target mutex
  - **attr**:
    - NULL: the default mutex attributes are used
    - Non-NULL: initializes with specified attributes

Creating a mutex

- Default attributes
  - Use **`PTHREAD_MUTEX_INITIALIZER`**
    - Statically allocated
    - Equivalent to dynamic initialization by a call to **`pthread_mutex_init()`** with parameter **`attr`** specified as NULL
    - No error checks are performed

Destroying a mutex

```
#include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t
    *mutex);
```

- Destroy a pthread mutex
- Returns
  - 0 on success
  - Error number on failure
    - **EBUSY:** An attempt to re-initialise a mutex; **EINVAL:** The value specified by attr is invalid
- Parameters
  - **mutex**: Target mutex

Locking/unlocking a mutex

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t
    *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Returns
  - 0 on success
  - Error number on failure
    - **`EBUSY:`** already locked; **`EINVAL:`** Not an initialised mutex; **`EDEADLK:`** The current thread already owns the mutex; **`EPERM:`** The current thread does not own the mutex

Example

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

static pthread_mutex_t my_lock =
     PTHREAD_MUTEX_INITIALIZER;

void *mythread(void *ptr) {
   long int i,j;
   while (1) {

     pthread_mutex_lock (&my_lock);

     for (i=0; i<10; i++) {
       printf ("Thread %d\n", int) ptr);
       for (j=0; j<50000000; j++);
     }

     pthread_mutex_unlock (&my_lock);
     for (j=0; j<50000000; j++);
   }
}
```

```
int main (int argc, char *argv[]) {
   pthread_t thread[2];

   pthread_create(&thread[0], NULL,
    mythread, (void *)0);

   pthread_create(&thread[1], NULL,
    mythread, (void *)1);

   getchar();
}
```

Condition Variables

- Used to communicate information about the state of shared data
  - Execution of code depends on the state of
    - A data structure or
    - Another running thread
- Allows threads to synchronize based upon the actual value of data
- Without condition variables
  - Threads continually poll to check if the condition is met

Condition Variables

- Signaling, not mutual exclusion
  - A mutex is needed to synchronize access to the shared data
- Each condition variable is associated with a single mutex
  - Wait atomically unlocks the mutex and blocks the thread
  - Signal awakens a blocked thread

Creating a Condition Variable

- Similar to pthread mutexes

```
int pthread_cond_init(pthread_cond_t *cond, const
    pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Using a Condition Variable

- Waiting
  - Block on a condition variable.
  - Called with mutex locked by the calling thread
  - Atomically release the mutex and cause the calling thread to block on the condition variable
  - On return, mutex is locked again

```
int pthread_cond_wait(pthread_cond_t *cond,
   pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
   pthread_mutex_t *mutex, const struct timespec
   *abstime);
```

Using a Condition Variable

- Signaling

```
int pthread_cond_signal(pthread_cond_t *cond);
```

  - unblocks at least one of the blocked threads

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

  - unblocks all of the blocked threads

Using a Condition Variable: Challenges

- Call `pthread_cond_signal()` before calling `pthread_cond_wait()`
  - Logical error
- Fail to lock the mutex before calling `pthread_cond_wait()`
  - May cause it NOT to block
- Fail to unlock the mutex after calling `pthread_cond_signal()`
  - May not allow a matching `pthread_cond_wait()` routine to complete (it will remain blocked).

Condition Variable: Why do we need the mutex?

```
pthread_mutex_lock(&mutex);                  /* lock mutex */
while (!predicate) {                         /* check predicate */
  pthread_cond_wait(&condvar,&mutex);
                                             /* go to sleep - recheck
                                                pred on awakening */
}
pthread_mutex_unlock(&mutex);                /* unlock mutex */
```

```
pthread_mutex_lock(&mutex);                  /* lock the mutex       */
predicate=1;                                 /* set the predicate    */
pthread_cond_broadcast(&condvar);            /* wake everyone up     */
pthread_mutex_unlock(&mutex);                /* unlock the mutex     */
```

Condition Variable: Why do we need the mutex?

The problem is here

```
pthread_mutex_lock(&mutex);          /* lock mutex */
while (!predicate) {                 /* check predicate      */
  pthread_mutex_unlock(&mutex);      /* unlock mutex         */
  pthread_cond_wait(&condvar);       /* go to sleep - recheck
                                        pred on awakening    */
  pthread_mutex_lock(&mutex);        /* lock mutex           */
}
pthread_mutex_unlock(&mutex);        /* unlock mutex         */
```

What can happen here?

```
pthread_mutex_lock(&mutex);          /* lock the mutex       */
                                     /* set the predicate    */
                                     /* wake everyone up     */
                                     /* unlock the mutex     */
```

Another thread might acquire the mutex, set the predicate, and issue the broadcast before **`pthread_cond_wait()`** gets called

Condition Variable: Why do we need the mutex?

- Separating the condition variable from the mutex
  - Thread goes to sleep when it shouldn't
  - Problem
    - **`pthread_mutex_unlock()`** and **`pthread_cond_wait()`** are not guaranteed to be atomic
- Joining condition variable and mutex
  - Call to **`pthread_cond_wait()`** unlocks the mutex
  - UNIX kernel can guarantee that the calling thread will not miss the broadcast

Example without Condition Variables

```
int data_avail = 0;
pthread_mutex_t data_mutex =
    PTHREAD_MUTEX_INITIALIZER;

void *producer(void *) {
    pthread_mutex_lock(&data_mutex);

    <Produce data>
    <Insert data into queue;>
    data_avail=1;

    pthread_mutex_unlock(&data_mutex);
}
```

Example without Condition Variables

```
void *consumer(void *) {
   while( !data_avail );  /* do nothing */

   pthread_mutex_lock(&data_mutex);

   <Extract data from queue;>
   if (queue is empty)
      data_avail = 0;

   pthread_mutex_unlock(&data_mutex);
   <Consume Data>
}
```

Example with Condition Variables

```
int data_avail = 0;
pthread_mutex_t data_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cont_t data_cond = PTHREAD_COND_INITIALIZER;

void *producer(void *) {
   pthread_mutex_lock(&data_mutex);
   <Produce data>
   <Insert data into queue;>
   data_avail = 1;

   pthread_cond_signal(&data_cond);
   pthread_mutex_unlock(&data_mutex);
}
```

Example with Condition Variables

```
void *consumer(void *) {
   pthread_mutex_lock(&data_mutex);
   while( !data_avail ) {
      /* sleep on condition variable*/
      pthread_cond_wait(&data_cond, &data_mutex);
   }
   /* woken up */
   <Extract data from queue;>
   if (queue is empty)
      data_avail = 0;
   pthread_mutex_unlock(&data_mutex);
   <Consume Data>
}
```

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

```
#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

```
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: Slave

```
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);
}
```

Example: Master

```
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);
}
```

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

Dining Philosophers

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