POSIX Thread Synchronization

- Mutex Locks
- Condition Variables
- Read-Write Locks

- Reading: R&R, Ch 13

POSIX Thread Synchronization

- Mutex Locks
- Condition Variables
- Read-Write Locks

- Reading: R&R, Ch 13

Mutex Locks

```
#include <pthread.h>

int pthread_mutex_init(      pthread_mutex_t      * restrict mutex,
                       const pthread_mutexattr_t * restrict attr);

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

EAGAIN: System lacks non-memory resources to initialize *mutex
ENOMEM: System lacks memory resources to initialize *mutex
EPERM: Caller does not have appropriate privileges

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

```
int pthread_mutex_lock    (pthread_mutex_t *mutex);
int pthread_mutex_trylock (pthread_mutex_t *mutex);
int pthread_mutex_unlock  (pthread_mutex_t *mutex);
```

EINVAL: mutex configured with priority-ceiling on, and caller's priority is higher than mutex's current priority ceiling.
EBUSY: another thread holds the lock (returned to `mutex_trylock`)

Mutex Locks: Operations

```
pthread_mutex_t mylock = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock(&mylock);
      /* critical section */
pthread_mutex_unlock(&mylock);
```

- Use mutex locks to preserve critical sections or obtain exclusive access to resources.
- **Hold mutexes for short periods of time only!**
- "Short periods"?!
  - For example, changes to shared data structures.
- Use **Condition Variables** when waiting for events!

Uses for Mutex Locks: Unsafe Library Functions

Def: **Thread-safe** function: Exhibits no race conditions in multithreaded environment.

- Many library functions are not thread-safe!
- Can be made thread-safe with mutexes.

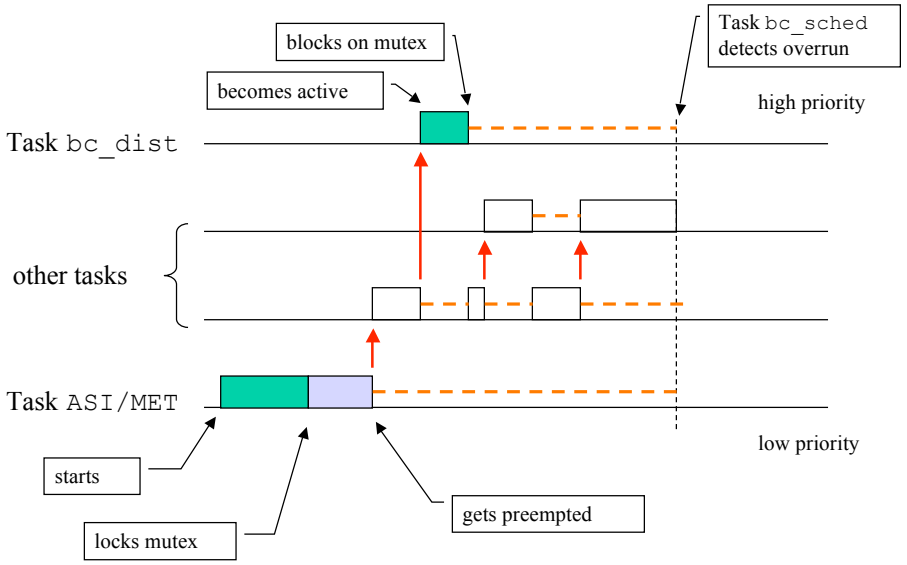
```
#include <pthread.h>
#include <stdlib.h>

int randsafe(int * result) {
    static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
    int error;

    if (error = pthread_mutex_lock(&lock)
        return error;
    *result = rand();
    return pthread_mutex_unlock(&lock);
}
```

Threads and Priorities: Interlude on Mars!

- Landing on July 4, 1997
- "experiences software glitches"
- Pathfinder experiences repeated RESETs after starting gathering of meteorological data.
- RESETs generated by watchdog process.
- Timing overruns caused by priority inversion.
- Resources:

  http://research.microsoft.com/~mbj/Mars_Pathfinder/

Priority Inversion on Mars Pathfinder



POSIX Thread Synchronization

- Mutex Locks
- Condition Variables
- Read-Write Locks

- Reading: R&R, Ch 13

POSIX Condition Variables

waiting for a particular condition (e.g. x==y)

```
while (x != y);
```

correct strategy to wait for condition

while (x != y):
1. lock a mutex
2. test the condition (x==y)
3. if TRUE, unlock mutex and exit loop
4. if FALSE, **suspend thread** and unlock mutex (?!!!)

```
pthread_mutex_lock(&m);
while (x != y)
    pthread_cond_wait(&v, &m);
/* now we are in the "critical section" */
pthread_mutex_unlock(&m);
```

```
pthread_mutex_lock(&m);
x++;
pthread_cond_signal(&v);
pthread_mutex_unlock(&m);
```

- When `cond_wait` returns, thread owns mutex and can test condition again.
- Call `cond_wait` only if you own mutex!

Example: Thread-Safe Barrier Locks

```
/* shared variables */
static pthread_cond_t  bcond  = PTHREAD_CONT_INITIALIZER;
static pthread_mutex_t bmutex = PTHREAD_MUTEX_INITIALIZER;
static int count = 0; /* how many threads are waiting? */
static int limit = 0;
```

```
/* initialize the barrier */
int initbarrier(int n) {
    int error;
    if (error = pthread_mutex_lock(&bmutex))
        return error;
    if (limit != 0) { /* don't initialize barrier twice! */
        pthread_mutex_unlock(&bmutex);
        return EINVAL;
    }
    limit = n;
    return pthread_mutex_unlock(&bmutex);
}
```

Example: Thread-Safe Barrier Locks

```
/* shared variables */
static pthread_co
static pthread_mu
static int count
static int limit
```

```
/* wait at barrier until all n threads arrive */
int waitbarrier(void) {
    int berror = 0;
    int error;
    if (error = pthread_mutex_lock(&bmutex))
        return error;
    if (limit <= 0) { /* barrier not initialized?! */
        pthread_mutex_unlock(&bmutex);
        return EINVAL;
    }
    count++;
    while ((count < limit) && !berror)
        berror = pthread_cont_wait(&bcond, &bmutex);
    if (!berror) /* wake up everybody */
        berror = pthread_cont_broadcast(&bcond);
    error = pthread_mutex_unlock(&bmutex);
    if (berror)
      return berror;
    return error;
}
```

Timed Wait on Condition Variables

```
#include <pthread.h>

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

Reader/Writer Locks

- R/W locks differentiate between exclusive (write) and shared (read) access.
- Reader *vs.* writer priority not specified in POSIX.

```
#include <pthread.h>

int pthread_rwlock_init(      pthread_rwlock_t      * rwlock,
                        const pthread_rwlockattr_t * attr);
```

EAGAIN: System lacks non-memory resources to initialize *rwlock
ENOMEM: Yada … Yada …

```
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

```
int pthread_rwlock_rdlock   (pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock   (pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock   (pthread_rwlock_t *rwlock);
```

R/W Lock Example: Vanilla Shared Container

```
/* shared variable */
static pthread_rwlock_t listlock;
static int lockiniterror = 0;
```

```
int init_container(void){
    return pthread_rwlock_init(&listlock, NULL)
}
```

```
/* add an item */
int add_data_r(data_t data, key_t key) {
    int error;
    if (error = pthread_rwlock_wrlock(&listlock)) {
        errno = error;
        return -1;
    }
    add_data(data, key);
    if (error = pthread_rwlock_unlock(&listlock)) {
        errno = error;
        error = -1;
    }
    return error;
}
```

R/W Lock Example: Vanilla Shared Container

```
/* shared variable */
static pthread_rwlock_t listlock;
static int lockiniterror = 0;
```

```
/* add an item */
int get_data_r(key_t key, data_t * datap) {
    int error;
    if (error = pthread_rwlock_rdlock(&listlock)) {
        errno = error;
        return -1;
    }
    get_data(key, datap);
    if (error = pthread_rwlock_unlock(&listlock)) {
        errno = error;
        error = -1;
    }
    return error;
}
```