POSIX Threads

HUJI
Spring 2011
The primary motivation for using threads is to realize potential program performance gains and structuring.

– Overlapping CPU work with I/O.
– Priority/real-time scheduling.
– Asynchronous event handling.
– Keeping track of concurrent activities

Inter-thread communication is more efficient and easier to use than inter-process communication (IPC).
P-thread Attributes

A thread can possess an independent flow of control and be schedulable because it maintains its own:

– Stack.

– Registers. (CPU STATE!)

– Scheduling properties (such as policy or priority).

– Set of pending and blocked signals.

– Thread specific data.

![Diagram of PCB and thread descriptors]
POSIX Threads

In the UNIX environment a thread:

• Exists within a process and uses the process resources.
• Has its own independent flow of control
• Duplicates only the essential resources
• May share the process resources with other threads.
• Dies if the parent process dies.
• Is “lightweight” because most of the overhead accomplished through the creation of the parent process.
Implications

Threads within the same process share resources:

• Changes made by one thread to shared system resources will be seen by all other threads

• Two pointers having the same value point to the same data

• Read/Write to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.
Shared Memory
Thread-safeness:

- Thread-safeness: in a nutshell, refers an application’s ability to execute multiple threads simultaneously without corrupting shared data.

- The implication to users of external library routines is that if you aren’t 100% certain the routines is thread-safe, then you take your chances with problems that could arise.
Parallel Programming

• Whatever applies to parallel programming in general, applies to parallel pthreads programs.

• There are many considerations for designing parallel programs, such as:
  – Problem partitioning
  – Load balancing
  – Communications
  – Data dependencies
  – Synchronization
The subroutines which comprise the Pthreads API can be informally grouped into 3 major classes:

- **Thread management**: Working directly on threads.

- **Mutex**: Dealing with locks used for mutual exclusion type synchronization. They allow a thread to wait for the condition that no other thread is doing something.

- **Condition variables**: A generalization that allows a thread to wait for an arbitrary condition.

**How to Compile?**

```bash
#include <pthread.h>
gcc ex3.c -o ex3 -lpthread
```
Thread management
Creating Threads

Initially, your main() program comprises a single, default thread. All other threads must be explicitly created by the programmer.

```c
int pthread_create ( 
    pthread_t *thread, 
    const pthread_attr_t *attr=NULL, 
    void *(*start_routine) (void *), 
    void *arg) ;
```
Terminating Thread Execution

• The thread returns from its starting routine (the main routine for the initial thread).

• The thread makes a call to the `pthread_exit(status)` subroutine.

• The entire process is terminated due to a call to either the `exec` or `exit` subroutines.
```c
#define NUM_THREADS 5

void *PrintHello(void *index) {
    printf("%d: Hello World!\n", index);
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int res, t;
    for(t=0; t<NUM_THREADS; t++) {
        printf("Creating thread %d\n", t);
        res = pthread_create(&threads[t], NULL,
                               PrintHello, (void *)t);
        if (res<0) {
            printf("ERROR\n");
            exit(-1);
        }
    }
    return 0;
}
```
Joining Threads

```c
int pthread_join(pthread_t thread, void **value_ptr);
```

- The `pthread_join()` subroutine blocks the calling thread until the specified `thread` thread terminates.
- The programmer is able to obtain the target thread's termination return status if specified through `pthread_exit(void *status)`.
int pthread_join(pthread_t thread, void **value_ptr);
Example Cont.

// main thread waits for the other threads
for(t=0; t<NUM_THREADS; t++) {
    res = pthread_join(threads[t], (void **) &status);
    if (res<0) {
        printf("ERROR \n");
        exit(-1);
    }
    printf("Completed join with thread %d status= %d \n", t, status);
}
Mutex
## Critical Section

<table>
<thead>
<tr>
<th>Balance</th>
<th>1\textsuperscript{st} Thread</th>
<th>2\textsuperscript{nd} Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000$</td>
<td>Read balance: 1000$</td>
<td></td>
</tr>
<tr>
<td>1000$</td>
<td></td>
<td>Read balance: 1000$</td>
</tr>
<tr>
<td>1000$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000$</td>
<td></td>
<td>Deposit: 200$</td>
</tr>
<tr>
<td>1000$</td>
<td>Deposit: 200$</td>
<td></td>
</tr>
<tr>
<td>1200$</td>
<td>Update balance</td>
<td></td>
</tr>
<tr>
<td>1200$</td>
<td></td>
<td>Update balance</td>
</tr>
</tbody>
</table>
Mutex

• Mutex (MUTual EXclusion) variables are one of the primary means of implementing *thread synchronization* and for protecting shared data when multiple writes occur.

• A mutex variable acts like a lock protecting access to a shared data resource.

• The basic concept of a mutex as used in P-threads is that *only one thread can lock* (or own) a mutex variable at any given time.

• The *programmer* is responsible to make sure every thread that needs to use a mutex does so.
Mutex Work Flow

A typical sequence in the use of a mutex is as follows:

– Create and initialize a mutex variable.
– Several threads attempt to lock the mutex.
– Only one succeeds and that thread owns the mutex.
– The owner thread performs some set of actions.
– The owner unlocks the mutex.
– Another thread acquires the mutex and repeats the process.
– Finally the mutex is destroyed.
Beware of Deadlocks!

Thread 1:
Mutex (A)
Mutex (B)

a=b+a;
b=b*a;

Thread 2:
Mutex (B)
Mutex (A)

b=a+b
a=b*a;
Creating and Destroying Mutex

Mutex variables must be declared with type `pthread_mutex_t`, and must be initialized before they can be used:

- Statically, when it is declared. For example:
  ```c
  pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
  ```

- Dynamically,
  ```c
  pthread_mutex_init(mutex, attr)
  ```

This allows setting mutex attributes (default setting use NULL).

`pthread_mutex_destroy(mutex)` should be used to free a mutex object when is no longer needed.
Locking a Mutex

• The `pthread_mutex_lock(mutex)` routine is used by a thread to acquire a lock on the specified `mutex` variable.

• If the `mutex` is already locked by another thread, this call will block the calling thread until the `mutex` is unlocked.
Unlock a Mutex

• `pthread_mutex_unlock(mutex)` will unlock a mutex if called by the owning thread. Calling this routine is required after a thread has completed its use of protected data.

• An error will be returned if:
  – If the mutex was already unlocked.
  – If the mutex is owned by another thread.
Example

Thread 1:

a = counter;

a++;

counter = a;

Thread 2:

b = counter;

b--;

counter = b;
Thread 1:

```c
pthread_mutex_lock (&mut);
a = counter;
a++;
counter = a;
pthread_mutex_unlock (&mut);
```

Thread 2:

```c
pthread_mutex_lock (&mut);
b = counter;
b--;
counter = b;
pthread_mutex_unlock (&mut);
```

//Checking of return values omitted for brevity
Conditional Variables
Conditional Variables

• While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.

• Without condition variables, the programmer would need to have threads continually polling (usually in a critical section), to check if the condition is met.

• A condition variable is a way to achieve the same goal without polling (a.k.a. “busy wait”)

Condition Variables

- Useful when a thread needs to wait for a certain condition to be true.

- In pthreads, there are four relevant procedures involving condition variables:
  - `pthread_cond_init(pthread_cond_t *cv, NULL);`
  - `pthread_cond_destroy(pthread_cond_t *cv);`
  - `pthread_cond_wait(pthread_cond_t *cv, pthread_mutex_t *lock);`
  - `pthread_cond_signal(pthread_cond_t *cv);`
Creating and Destroying Conditional Variables

• Condition variables must be declared with type `pthread_cond_t`, and must be initialized before they can be used.
  
  – Statically, when it is declared. For example:
    ```c
    pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;
    ```
  
  – Dynamically
    ```c
    pthread_cond_init(cond, attr);
    ```
  
  Upon successful initialization, the state of the condition variable becomes initialized.

• `pthread_cond_destroy(cond)` should be used to free a condition variable that is no longer needed.
**Pthread_cond_wait**

- `pthread_cond_wait(cv, lock)` is called by a thread when it wants to block and wait for a condition to be true.

- It is assumed that the thread has locked the mutex indicated by the second parameter.

- The thread releases the mutex, and blocks until awakened by a `pthread_cond_signal()` call from another thread.

- When it is awakened, it waits until it can acquire the mutex, and once acquired, it returns from the `pthread_cond_wait()` call.
• `pthread_cond_signal()` checks to see if there are any threads waiting on the specified condition variable. If not, then it simply returns.

• If there are threads waiting, then one is awakened.

• There can be no assumption about the order in which threads are awakened by `pthread_cond_signal()` calls.

• It is natural to assume that they will be awakened in the order in which they waited, but that may not be the case...

• *Use loop or `pthread_cond_broadcast()` to awake all waiting threads.*
typedef struct {
    pthread_mutex_t *lock;
    pthread_cond_t *cv;
    int *ndone;
    int id;
} TStruct;

#define NTHREADS 5

void *barrier(void *arg) { //Checking of return values omitted for brevity
    TStruct *ts;
    int i;
    ts = (TStruct *) arg;
    printf("Thread %d -- waiting for barrier\n", ts->id);
    pthread_mutex_lock(ts->lock);
    *ts->ndone = *ts->ndone + 1;
    if (*ts->ndone < NTHREADS) {
        pthread_cond_wait(ts->cv, ts->lock);
    }
    else {
        for (i = 1; i < NTHREADS; i++)
            pthread_cond_signal(ts->cv);
    }
    pthread_mutex_unlock(ts->lock);
    printf("Thread %d -- after barrier\n", ts->id);
}
main() { //Checking of return values omitted for brevity
    TStruct ts[NTHREADS];
    pthread_t tids[NTHREADS];
    int i, ndone;
    pthread_mutex_t lock;
    pthread_cond_t cv;
    void *retval;
    pthread_mutex_init(&lock, NULL);
    pthread_cond_init(&cv, NULL);
    ndone = 0;
    for (i = 0; i < NTHREADS; i++) {
        ts[i].lock = &lock;
        ts[i].cv = &cv;
        ts[i].ndone = &ndone;
        ts[i].id = i;
    }
    for (i = 0; i < NTHREADS; i++)
        pthread_create(tids+i, NULL, barrier, ts+i);
    for (i = 0; i < NTHREADS; i++)
        pthread_join(tids[i], &retval);
    printf("done
");
}
Thread 0 -- waiting for barrier
Thread 1 -- waiting for barrier
Thread 2 -- waiting for barrier
Thread 3 -- waiting for barrier
Thread 4 -- waiting for barrier
Thread 4 -- after barrier
Thread 0 -- after barrier
Thread 1 -- after barrier
Thread 2 -- after barrier
Thread 3 -- after barrier
done