Pthreads

Jingke Li

Portland State University

Pthreads

The standard thread package on Unix/Linux systems. To use it on Linux,

- Include the header file <pthread.h> in your program.
- Compile your program with the "-pthread" flag.

Useful Routines:

subroutine (can take only single argument)

```
hread_create(&thread, attributes, func, arg);
hread_exit(status);
hread_join(thread, &status);    // sync back a thread
hread_cancel(thread);           // force termination a thread
hread_self();                   // return the self thread
hread_equal(thread1, thread2); // compare two threads
hed_yield();                    // yield to another thread
```

only parent

- The `pthread_join()` routine is for synchronizing a *terminated* thread back to its parent.
- Not all threads are joinable — a thread can be created as *detached*; when it terminates, it is destroyed and its resource released.

1. Don't call join. — leaves garbage, can become zombie
2. Run in detach — make it separate process.

Synchronizations in Pthreads

two ways to use

1 ▸ Mutex Locks:

```
pthread_mutex_t mutex;

pthread_mutex_init(&mutex, attributes); // Must be initialized to use
pthread_mutex_lock(&mutex);
pthread_mutex_trylock(&mutex); // Check - do not block
pthread_mutex_destroy();
```

2 ▸ Condition Variables:

```
pthread_cond_t cond;

pthread_cond_init(&cond, attributes); declare and init.
pthread_cond_signal(cond); - send signal to 1 thread (at least one)
pthread_cond_broadcast(cond); → wake up all threads.
pthread_cond_wait(cond, mutex);
```

It is difficult to wake up one singal wakes up more than 1 (sometimes)

Attributes - Further controls over the mutex and conditional var.

Various attributes can be set for threads, locks, and condition variables.
For normal cases, the default values work just fine.

- For threads:
  - scheduling policy, stack size, detached state → Joinable?
    - → where thread will be executed
    - → size the thread
- For mutexes:
  - *normal* — only a single thread is allowed to lock it; if a threads tries to lock it twice a deadlock occurs.
  - *recursive* — a thread can lock the mutex multiple time; each successive lock/unlock increments/decrements a counter; another thread can lock a mutex only if its counter is zero.
- For condition variables:
  - can enable cross-process sharing default is 0 (null)

Convinient to loop on the same variable during recursive call

Pthread is part of Linux so it's C based.

A Simple Example

```
#include <pthread.h>

void child(long id) {
  printf("Child %ld\n", id);
}

main() {
  long i=1, j=2;
  pthread_t thread1, thread2;
  pthread_create(&thread1, NULL,
                 (void*) child, (void*)i);
  pthread_create(&thread2, NULL,
                 (void*) child, (void*)j);
  printf("Parent: waiting for child 1\n");
  pthread_join(thread1, NULL);
  printf("Parent: waiting for child 2\n");
  pthread_join(thread2, NULL);
  printf("Parent: done\n");
}
```

Output:

```
Parent: waiting for child 1
Child 1
Child 2
Parent: waiting for child 2
Parent: done
```

C style convinience

Usualy main thread gets waisted.

More economical way: execute one copy of workload in the parent.

Execution of the threads is independent but not non deterministic.

Mutexes — Guard critical section.

Locks are implemented in Pthreads with "mutex" variables.

- To use a mutex, first it must be declared and initialized:

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
```

  NULL specifies a default attribute for the mutex.

- A critical section can then be protected with the mutex:

```
pthread_mutex_lock(&mutex);
<critical section>
pthread_mutex_unlock(&mutex);
```

  - If a thread reaches a mutex lock and finds it locked, it will wait for the lock to open.
  - If more than one thread is waiting for the lock to open when it opens, the system will select one thread to be allowed to proceed.

Mutexes (cont.)

Not always the case in different approaches.

- Only the thread that locks a mutex can unlock it.
- Pthreads offers a routine that can test whether a lock is locked without blocking the thread:

```
pthread_mutex_trylock()
```

// Check the value, do not wait.

  - It will lock an unlocked mutex and return 0 or will return with `EBUSY` if the mutex is locked — useful in overcoming deadlock.
- A mutex can be destroyed with

```
pthread_mutex_destroy()
```


Condition Variables

Slightly tricky and more complex than mutex.

Each cond variable uses mutex

- Condition variables also need to be initialized before use:

```
pthread_cond_t cond;
pthread_cond_init(&cond, NULL);
```

- Each condition variable must be associated with a mutex, since the checking and setting of the condition must be done inside a critical section. The "`wait`" routine, in particular, takes a mutex as one of its arguments:

```
pthread_cond_wait(cond, mutex);
```

// Should use dedicated lock

- Signals that are sent out by "`signal`" or "`broadcast`" routines are not remembered, which means that threads must already be waiting for a signal to receive it.

Single and broadcast are not saved, if there is no waiting threads → message lost.

Condition Variable Example

Decrement a count; if value reaches 0, send a signal.

```
counter() {
  pthread_mutex_lock(&mutex);
  c--;
  if (c == 0)
    pthread_cond_signal(cond);
  pthread_mutex_unlock(&mutex);
}
```

Template
Standard way to do it

vs

Can create problems

Compare with two other versions:

```
counter_v2() {
  pthread_mutex_lock(&mutex);
  c--;
  if (c == 0)
    pthread_cond_broadcast(cond);
  pthread_mutex_unlock(&mutex);
}
```

```
counter_v3() {
  pthread_mutex_lock(&mutex);
  c--;
  pthread_mutex_unlock(&mutex);
  if (c == 0)
    pthread_cond_signal(cond);
}
```

Something can come in between and change c

Condition Variable Example (cont.)

```
action() {
  pthread_mutex_lock(&mutex);
  while (c <> 0)
    pthread_cond_wait(cond, mutex);
  pthread_mutex_unlock(&mutex);
  take_action();
}
```

Standard template

Compare with the following version:

```
action_v2() {
  pthread_mutex_lock(&mutex);
  if (c <> 0)
    pthread_cond_wait(cond, mutex);
  pthread_mutex_unlock(&mutex);
  take_action();
}
```

Won't work if more than one thread is woken up.

1. What if broadcast is used?
2. Signal sometimes wakes up more than 1.
3. Third independent thread might come in.

So it is safer to use while loop and guard the execution.

Array-Sum Example

```c
#include <pthread.h>

int arraySize = 1000;           // default array size
int numThreads = 10;            // default number of threads
int *array;                     // shared array
int sum = 0, idx = 0;           // global sum and idx
pthread_mutex_t sumLock;

int main(int argc, char **argv) {
  pthread_t thread[numThreads];
  array = init_array(arraySize);        // initialize array
  pthread_mutex_init(&sumLock, NULL);   // initialize mutex

  for (long k = 0; k < numThreads; k++) {       // create threads
    pthread_create(&thread[k], NULL, (void*)slave, (void*)k);
  }
  for (long k = 0; k < numThreads; k++) {       // join threads
    pthread_join(thread[k], NULL);
  }
  printf("The sum of 1 to %i is %d\n", arraySize, sum);
}
```


Array-Sum Example (cont.)

```c
int *init_array(int size) {
  int *array = (int *) malloc(sizeof(int) * size);
  for (int i = 0; i<size; i++)
    array[i] = i + 1;
  return array;
}

void slave(long tid) {
  printf("Thread %ld started\n", tid);

  int i, psum = 0;
  do {
    pthread_mutex_lock(&sumLock);     // read and increment idx
    i = idx++;
    pthread_mutex_unlock(&sumLock);
    if (i < arraySize)                // add one array element
      psum += array[i];
  } while (i < arraySize);
  pthread_mutex_lock(&sumLock);       // add local psum to global sum
  sum += psum;
  pthread_mutex_unlock(&sumLock);
}
```

Array-Sum Version 2

Show where threads are executed.

```
#define _GNU_SOURCE
#include <pthread.h>
#include <sched.h>                  // for getting cpu id

void slave(long tid) {
  printf("Thread %ld started on %d\n", tid, sched_getcpu());

  ...
}
```

Question: Can we have more control over where threads execute?

Array-Sum Version 3

Control where threads are executed.

```
#define _GNU_SOURCE
#include <pthread.h>
#include <sched.h>                  // for getting cpu id
#include <unistd.h>                 // for getting nprocs

int main(int argc, char **argv) {
  pthread_t thread[numThreads];
  array = init_array(arraySize);                    // initialize array
  pthread_mutex_init(&sumLock, NULL);               // initialize mutex
  int nprocs = sysconf(_SC_NPROCESSORS_ONLN);
  cpu_set_t cpuset;
  int cid = 0;
  for (long k = 0; k < numThreads; k++) {           // create threads
    pthread_create(&thread[k], NULL, (void*)slave, (void*)k);
    CPU_ZERO(&cpuset);
    CPU_SET(cid++ % nprocs, &cpuset);
    pthread_setaffinity_np(thread[k], sizeof(cpu_set_t), &cpuset);
  }
  ...
}
```

Array-Sum Version 4

Add command-line arguments for parameter configurations.

```
int arraySize;                  // array size, given by user
int numThreads = 1;             // default number of threads

int main(int argc, char **argv) {
  if (argc < 2) {
    printf ("Usage: ./arraysum4 <arraySize> [<numThreads>]\n");
    exit(0);
  } else if (argc > 2) {
    if ((numThreads=atoi(argv[2])) < 1) {
      printf ("<numThreads> must be greater than 0\n");
      exit(0);
    }
  }
  if ((arraySize=atoi(argv[1])) < 1) {
    printf ("<arraySize> must be greater than 0\n");
    exit(0);
  }
  ...
}
```


Producer-Consumer with Bounded Buffer

Problem Description:

- One producer, multiple consumers, and a bounded task queue.
- The producer creates tasks and adds them one by one to the end of the task queue. If the queue is full, it waits for new space to open up.
- Each consumer removes tasks one by one from the head of the task queue. If the queue is empty, it waits for new task to appear.

Programming Issues:

- Task and queue representations
- Threads creation and management
- Synchronization
- Termination

Producer-Consumer: Task and Queue Representations

```
typedef struct task_ task_t;
struct task_ {
  int val;          // each task holds an integer value
  task_t *next;     // and a pointer to next task
};

typedef struct queue_ queue_t;
struct queue_ {
  task_t *head;
  task_t *tail;
  int limit;        // queue size limit
  int length;       // current number of tasks
};
```

Supporting Routines:

```
task_t *create_task(int val) { ... }
queue_t *init_queue(int limit) { ... }
void add_task(queue_t *queue, task_t *task) { ... }
task_t *remove_task(queue_t *queue) { ... }
```


Producer-Consumer: Threads Creation and Management

```
int main(int argc, char **argv) {

  // create consumer threads
  pthread_t *threads =
    (pthread_t *) malloc(sizeof(pthread_t) * numConsumers);
  for (long k = 0; k < numConsumers; k++)
    pthread_create(&threads[k], NULL, (void*)consumer, (void*)k);

  // execute the producer code
  producer();

  // wait for consumer threads to terminate
  for (long k = 0; k < numConsumers; k++)
    pthread_join(threads[k], NULL);
}
```

Producer-Consumer: Synchronization

```
void producer() {
  while (<there is still task>) {
    <try to add a task to the queue>
    <wait if no space available>
  }
}

void consumer(long tid) {
  while (1) {
    <try to get a task from the queue>
    <wait if none available>
  }
}
```

Questions:

1. Other than the two `wait`s, is there a need for additional synchronizations?
2. Who is/are responsible for sending signals to the waiting threads?

Producer-Consumer: Termination

```
void producer() {
  while (<there is still task>) {
    <try to add a task to the queue>
    <wait if no space available>
  }
}

void consumer(long tid) {
  while (1) {
    <try to get a task from the queue>
    <wait if none available>
  }
}
```

Questions:

1. Can the producer thread terminate on its own?
2. Can the consumer threads ternimate on their own?
3. If not, what additional mechanism is needed?

Quicksort Program Framework

```
// A global array of size N contains the integers to be sorted.
// A global task queue is initialized with the sort range [0,N-1].
//
int main(int argc, char **argv) {
  // 1. read in command-line arguments, N and numThreads;
  // 2. initialize array, queue, and other shared variables

  // 3. create numThreads-1 worker threads, each executes a copy
  //    of the worker() routine
  for (long k = 0; k < numThreads-1; k++)
    pthread_create(&thread[k], NULL, (void*)worker, (void*)k);

  // 4. the main thread also runs a copy of the worker() routine;
  //    its copy has the last id, numThreads-1
  worker(numThreads-1);

  // 5. the main thread waits for worker threads to join back
  for (long k = 0; k < numThreads-1; k++)
    pthread_join(thread[k], NULL);

  // 6. verify the result
}
```


Quicksort Program Framework (cont.)

```
void worker(long wid) {
  while (<termination condition is not met>) {
    task = remove_task();
    quicksort(array, task->low, task->high);
  }
}

void quicksort(int *array, int low, int high, long wid) {
  // 1. find a pivot and partition the array into two segments
  int middle = partition(array, low, high);
  // 2. add the first segment to the task queue
  if (low < middle)
    <add task [low, middle-1] to queue>
  // 3. recursively sort on the second segment
  if (middle < high)
    quicksort(array, middle+1, high, wid);
}
```

Questions:

1. What synchronizations are needed?
2. What should the termination condition be?