Introduction to Pthreads

Pthreads

- Pthreads is a POSIX standard for describing a thread model, it specifies the API and the semantics of the calls.
- Model popular – nowadays practically all major thread libraries on Unix systems are Pthreads-compatible

Preliminaries

- Include `pthread.h` in the main file
- Compile program with `-lpthread`
  - `gcc -o test test.c -lpthread`
  - may not report compilation errors otherwise but calls will fail
- Good idea to check return values on common functions

Thread creation

- Types: `pthread_t` – type of a thread
- Some calls:

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void * (*start_routine)(void *),
                   void *arg);
int pthread_join(pthread_t thread, void **status);
int pthread_detach();
void pthread_exit();
```

  - No explicit parent/child model, except main thread holds process info
  - Call `pthread_exit` in main, don't just fall through;
  - Most likely you wouldn't need `pthread_join`
    - `status` = exit value returned by joinable thread
  - Detached threads are those which cannot be joined (can also set this at creation)

Attributes

- Type: `pthread_attr_t` (see `pthread_create`)
- Attributes define the state of the new thread
- Attributes: system scope, joinable, stack size, inheritance… you can use default behaviors with `NULL` in `pthread_create()`

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
pthread_attr_{set/get}{attribute}
```

- Example:

```
pthread_attr_t attr;
pthread_attr_init(&attr); // Needed!!!
pthread_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
pthread_create(NULL, &attr, foo, NULL);
```

Contention Scope

- *Contention scope* is the POSIX term for describing bound and unbound threads
- A bound thread is said to have *system contention scope*
  - i.e., it contends with all threads in the system
- An unbound thread has *process contention scope*
  - i.e., it contends with threads in the same process
- You can experiment in your project with this, but for most practical reasons use bound threads (system scope)
  - Solaris LWP switching cheap, Linux is one-to-one anyways…

Example: Multiple Threads

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 4

void *hello (void *arg) {
      printf("Hello Thread\n");
}

main() {
  pthread_t tid[NUM_THREADS];
  for (int i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], NULL, hello, NULL);

  for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}
```

What's Wrong?

What is printed for myNum?

```
void *threadFunc(void *pArg) {
  int* p = (int*)pArg;
  int myNum = *p;
  printf( "Thread number %d\n", myNum);
}
. . .
// from main():
for (int i = 0; i < numThreads; i++) {
   pthread_create(&tid[i], NULL, threadFunc, &i);
}
```

Solution – "Local" Storage

```
void *threadFunc(void *pArg)
{
  int myNum = *((int*)pArg);
  printf( "Thread number %d\n", myNum);
}
. . .

// from main():
for (int i = 0; i < numThreads; i++) {
  tNum[i] = i;
  pthread_create(&tid[i], NULL, threadFunc, &tNum[i]);
}
```

Pthread Mutexes

- Type: `pthread_mutex_t`

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- Attributes: for shared mutexes/condition vars among processes, for priority inheritance, etc.
  - use defaults
- Important: Mutex scope must be visible to all threads!

Condition variables

- Type `pthread_cond_t`

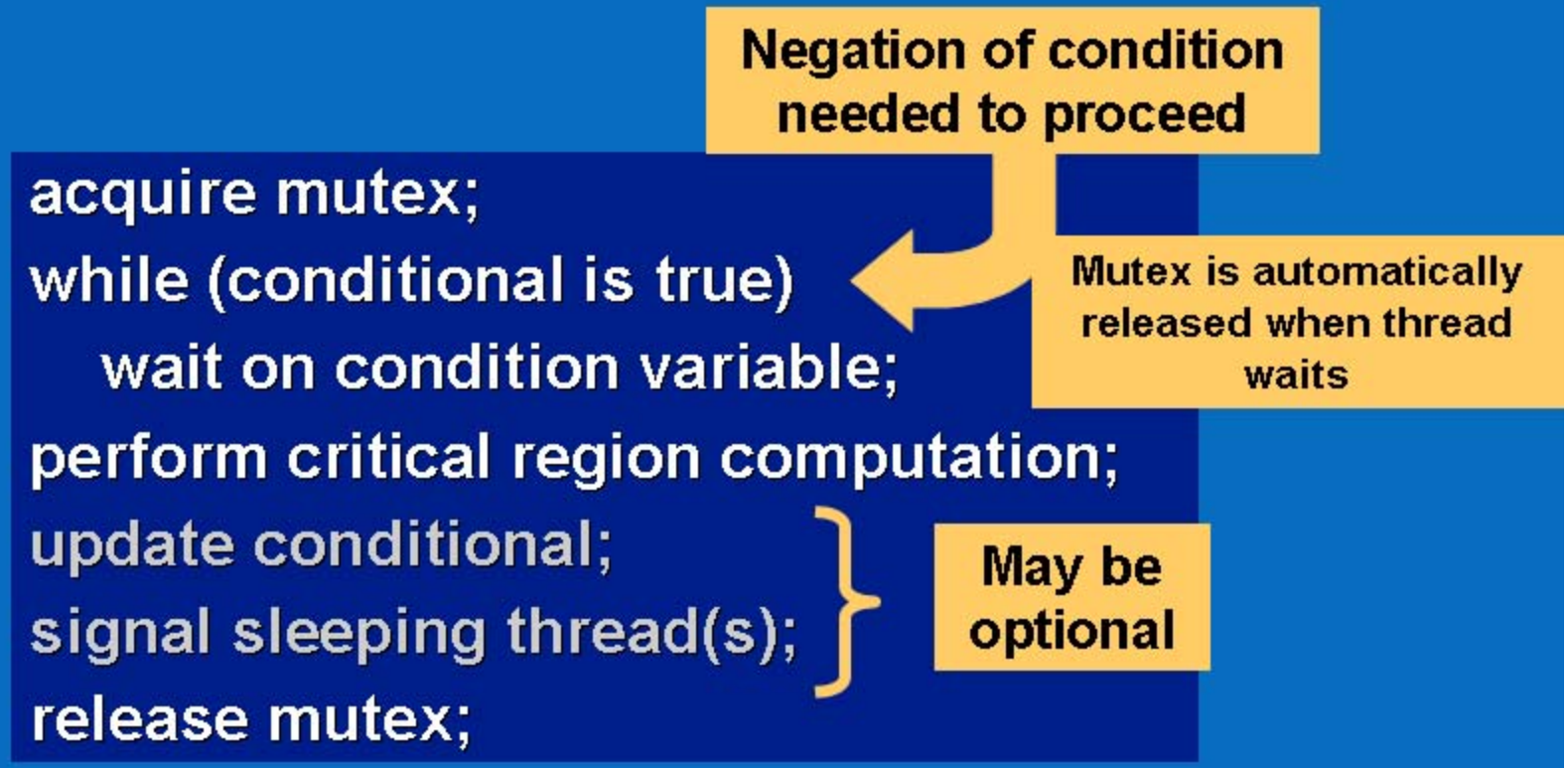
```
int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_singal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Some rules…

- Shared data should always be accessed through a single mutex (write a comment)
- Think of a boolean condition (expressed in terms of program variables) for each condition variable. Every time the value of the boolean condition may have changed, call Broadcast for the condition variable
  - Only call Signal when you are absolutely certain that *any and only one* waiting thread can enter the critical section
  - if noone is waiting, signal is lost
- Signaling/Broadcasting thread need not have the mutex
  - may be more efficient to release it first…
- Globally order locks, acquire in order in all threads

Condition Variable Algorithm

Avoids problems with lost and spurious signals



- Examples follow…

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

/* Example program creating
   thread to compute square of
   value */

int value;/* thread stores result here */
void *my_thread(void *param);
          /* the thread */

main (int argc, char *argv[])
{
   pthread_t tid;                /*
   thread identifier */
   int retcode;

   /* check input parameters */
   if (argc != 2) {
        fprintf (stderr, "usage:
   a.out <integer value>\n");
        exit(0);
   }
   /* create the thread */
   retcode =
   pthread_create(&tid,NULL,my_thr
   ead,argv[1]);
   if (retcode != 0) {
        fprintf (stderr, "Unable
   to create thread\n");
        exit (1);
   }

   /* wait for created thread to exit */
   pthread_join(tid,NULL);
   printf ("I am the parent:
   Square = %d\n", value);
} //main

/* The thread will begin control in this
   function */
void *my_thread(void *param)
{
   int i = atoi (param);

   printf ("I am the child, passed
   value %d\n", i);
   value = i * i;

   /* next line is not really
   necessary */
   pthread_exit(0);
}
```

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

/* Producer/consumer program illustrating conditional variables */

/* Size of shared buffer */
#define BUF_SIZE 3

int buffer[BUF_SIZE];  /*shared buffer */
int add=0;             /* place to add next element */
int rem=0;             /* place to remove next element */
int num=0;             /* number elements in buffer */
pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER; /* mutex lock for buffer */
pthread_cond_t c_cons=PTHREAD_COND_INITIALIZER; /* consumer waits on this
   cond var */
pthread_cond_t c_prod=PTHREAD_COND_INITIALIZER; /* producer waits on this
   cond var */

void *producer(void *param);
void *consumer(void *param);
```

```
main (int argc, char *argv[])
{
   pthread_t tid1, tid2;              /* thread identifiers */
   int i;

   /* create the threads; may be any number, in general */
   if (pthread_create(&tid1,NULL,producer,NULL) != 0) {
        fprintf (stderr, "Unable to create producer thread\n");
        exit (1);
   }
   if (pthread_create(&tid2,NULL,consumer,NULL) != 0) {
        fprintf (stderr, "Unable to create consumer thread\n");
        exit (1);
   }
   /* wait for created thread to exit */
   pthread_join(tid1,NULL);
   pthread_join(tid2,NULL);
   printf ("Parent quiting\n");
}
```

```
/* Produce value(s) */
void *producer(void *param)
{
   int i;
   for (i=1; i<=20; i++) {
        /* Insert into buffer */
        pthread_mutex_lock (&m);
        if (num > BUF_SIZE) exit(1);  /* overflow */
        while (num == BUF_SIZE)       /* block if buffer is full */
                pthread_cond_wait (&c_prod, &m);
        /* if executing here, buffer not full so add element */
        buffer[add] = i;
        add = (add+1) % BUF_SIZE;
        num++;
        pthread_mutex_unlock (&m);
        pthread_cond_signal (&c_cons);
        printf ("producer: inserted %d\n", i);  fflush (stdout);
   }
   printf ("producer quiting\n");  fflush (stdout);
}
```

```
/* Consume value(s); Note the consumer never terminates */
void *consumer(void *param)
{
   int i;
   while (1) {
        pthread_mutex_lock (&m);
        if (num < 0) exit(1);   /* underflow */
        while (num == 0)        /* block if buffer empty */
                pthread_cond_wait (&c_cons, &m);
        /* if executing here, buffer not empty so remove element */
        i = buffer[rem];
        rem = (rem+1) % BUF_SIZE;
        num--;
        pthread_mutex_unlock (&m);
        pthread_cond_signal (&c_prod);
        printf ("Consume value %d\n", i);  fflush(stdout);
   }
}
```