

CS 5523 Lecture 13: Implementations Using Threads

- *Questions on laboratory 2*
- *Thread synchronization*
- *Using mutex locks to access a bounded buffer*
- *Using condition variables for the producer-consumer*
- *Priority inversions and the Mars pathfinder*

Posix thread management:

- *pthread_create – create a new thread*
- *pthread_join – wait for a particular thread to terminate*
- *pthread_self – find out own thread ID*
- *pthread_detach – set thread to release resources when it terminates*
- *pthread_exit – terminate self*
- *pthread_kill – terminate another thread*

Posix thread locks:

- *pthread_mutex_init – initialize a lock*
- *pthread_mutex_destroy – destroy lock*
- *pthread_mutex_lock – obtain exclusive lock, block if not available*
- *pthread_mutex_unlock – release exclusive lock*
- *pthread_mutex_trylock – obtain exclusive lock, if available*

An example:

- A circular buffer which only one thread can access at a time.
- Buffer's size is known at compile time.

The example is taken from Program 10.1 of *Practical Unix Programming...* by Robbins and Robbins

A circular buffer protected by mutex locks:

```
#include <pthread.h>
#define BUFSIZE 8
static int buffer[BUFSIZE];
static int bufin = 0;
static int bufout = 0;
static pthread_mutex_t buffer_lock = PTHREAD_MUTEX_INITIALIZER;

void get_item(int *itemp)
{
    /* Get the next item from buffer and put it in *itemp. */
    pthread_mutex_lock(&buffer_lock);
    *itemp = buffer[bufout];
    bufout = (bufout + 1) % BUFSIZE;
    pthread_mutex_unlock(&buffer_lock);
    return;
}

void put_item(int item)
{
    /* Put item into buffer at position bufin and update bufin. */
    pthread_mutex_lock(&buffer_lock);
    buffer[bufin] = item;
    bufin = (bufin + 1) % BUFSIZE;
    pthread_mutex_unlock(&buffer_lock);
    return;
}
```

When does initialization take place?

Lock initialization:

- Initialization of synchronization variables is very tricky
- Three approaches:
 - Static initializers
 - Main thread calls `pthread_mutex_init` prior to creating any threads
 - Each thread calls an initializer routine that checks to see if the mutex has already been initialized before proceeding.

What are the advantages of each approach?

How would you write code to initialize the buffer at run time?

An example of a producer-consumer:

- The producer thread places the square of each integer between 1 and SUMSIZE in a shared buffer
- The consumer thread removes each square and adds it to sum
- The main thread waits for the threads to complete

The following example is taken from Program 10.2 of *Practical Unix Programming... by Robbins and Robbins*

Producer-consumer using a synchronized buffer:

```
void *producer(void * arg1)
{
    int i;
    for (i = 1; i <= SUMSIZE; i++)
        put_item(i*i);
    return NULL;
}

void *consumer(void *arg2)
{
    int i, myitem;
    for (i = 1; i <= SUMSIZE; i++) {
        get_item(&myitem);
        sum += myitem;
    }
    return NULL;
}

/* in main program */
pthread_create(&constid, NULL, consumer, NULL);
pthread_create(&prodtid, NULL, producer, NULL);
pthread_join(prodtid, NULL);
pthread_join(constid, NULL);
```

What is wrong with this code?

Posix condition variables:

- `pthread_cond_init` – initialize a condition variable
- `pthread_cond_destroy` – destroy a condition variable
- `pthread_cond_wait` – block on a condition variable
- `pthread_cond_timedwait` – block on a condition variable, using timeout
- `pthread_cond_signal` – signal on a condition variable
- `pthread_cond_broadcast` – broadcast a signal on a condition variable

Producer-consumer using condition variables

- The producer thread places the square of each integer between 1 and SUMSIZE in a shared buffer
- The consumer thread removes each square and adds it to sum
- The threads use condition variables to represent the availability of slots and items in the buffer
- The main thread waits for the threads to complete

The example is taken from Program 10.6 of Practical Unix Programming... by Robbins and Robbins

Producer-consumer initialization:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define SUMSIZE 100
#define BUFSIZE 8

int sum = 0;
pthread_cond_t slots = PTHREAD_COND_INITIALIZER;
pthread_cond_t items = PTHREAD_COND_INITIALIZER;
pthread_mutex_t slot_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t item_lock = PTHREAD_MUTEX_INITIALIZER;
int nslots = BUFSIZE;
int producer_done = 0;
int nitems = 0;

void get_item(int *itemp);
void put_item(int item);
```

When does initialization take place?

Producer:

```
void *producer(void * arg1)
{
    int i;

    for (i = 1; i <= SUMSIZE; i++) {
        pthread_mutex_lock(&slot_lock); /* acquire right to a slot */
        while (nslots <= 0)
            pthread_cond_wait (&slots, &slot_lock);
        nslots--;
        pthread_mutex_unlock(&slot_lock);

        put_item(i*i);
        pthread_mutex_lock(&item_lock); /* release right to an item */
        nitems++;
        pthread_cond_signal(&items);
        pthread_mutex_unlock(&item_lock);
    }
    pthread_mutex_lock(&item_lock);
    producer_done = 1;
    pthread_cond_broadcast(&items);
    pthread_mutex_unlock(&item_lock);
    return NULL;
}
```

Producer shuts down after doing work

Consumer:

```
void *consumer(void *arg2)
{
    int myitem;

    for ( ; ; ) {
        pthread_mutex_lock(&item_lock); /* acquire right to an item */
        while ((nitems <= 0) && !producer_done)
            pthread_cond_wait(&items, &item_lock);
        if ((nitems <= 0) && producer_done) {
            pthread_mutex_unlock(&item_lock);
            break;
        }
        nitems--;
        pthread_mutex_unlock(&item_lock);
        get_item(&myitem);
        sum += myitem;
        pthread_mutex_lock(&slot_lock); /* release right to a slot */
        nslots++;
        pthread_cond_signal(&slots);
        pthread_mutex_unlock(&slot_lock);
    }
    return NULL;
}
```

How does the consumer detect that the producer is done?

Main:

```
void main(void)
{
    pthread_t prodtid;
    pthread_t constid;
    int i, total;

    total = 0;
    for (i = 1; i <= SUMSIZE; i++)
        total += i*i;
    printf("The checksum is %d\n", total);

    pthread_create(&prodtid, NULL, producer, NULL);
    pthread_create(&constid, NULL, consumer, NULL);
    /* wait for the threads to finish */
    pthread_join(prodtid, NULL);
    pthread_join(constid, NULL);
    printf("The threads produced the sum %d\n", sum);
    exit(0);
}
```

How could you have multiple consumers?

Posix attribute objects:

- `pthread_attr_t` – create an attributed object
- `pthread_attr_t` – destroy an attribute object
- `pthread_attr_t` – get stack size
- `pthread_attr_t` – set stack size
- `pthread_attr_t` – get stack start
- `pthread_attr_t` – set stack start
- `pthread_attr_t` – get detached state attribute
- `pthread_attr_t` – set detached state attribute

Priority inversions:

- *What they are*
- *Why they are a problem*
- *Lessons from the Mars pathfinder mission*

For next time:

- *Look over Laboratory 3 and be prepared to ask questions*
- *Prepare questions for review*
