CS 31: Intro to Systems
The pthread Library

Martin Gagne
Swarthmore College
April 18, 2016
Thread operations

• create
  • Starts a new thread, calling a specified function.
  • Returns the thread’s ID.

• join
  • Block until a specified thread terminates.
  • Gives access to the thread function’s return value.

• lock/acquire
  • Block until the mutex is available, then claim it.

• unlock/release
  • Release a mutex.

• barrier_wait
  • Block until a specified number of threads reach the barrier.
Some pthread library functions

pthread_create

pthread_join

pthread_mutex_lock

pthread_mutex_unlock

pthread_barrier_wait
pthread_create

Returns zero on success, nonzero on error.

```c
int pthread_create(
    pthread_t *thread,  
    const pthread_attr_t *attr, 
    void *(*start_routine) (void *),  
    void *arg);
```

First arg is a thread ID pointer.

Second arg is usually NULL.

Third arg is the thread function.

Fourth arg is a pointer to the function’s args.
void*

int pthread_create(..., void* args);

void*: a pointer to any type (a generic pointer)

• all addresses are the same number of bytes
  char *cptr; int *ptr; // store 4 byte addresses

• can pass the address of any type as a void *
  pthread_create( ..., &x); // addr of an int
  pthread_create(..., &ch); //addr of a char

• cannot de-reference a void * pointer
  x = *args; // store 6 in 1 byte? 2 bytes? 4 bytes?

• re-cast first before dereference
  x = *(int *)args; // store 6 in 4 bytes
**pthread_join**

Returns zero on success, nonzero on error.

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

First arg is a thread ID to wait for.

Second arg is a pointer to be filled with the return value.
Example 1

/* pthreads "hello, world" program */
#include <pthread.h>

void* hello(void* arg);

int main() {
    pthread_t tid[2];

    pthread_create(&tid[0], NULL, hello, NULL);
    pthread_create(&tid[1], NULL, hello, NULL);
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    exit(0);
}

void* hello(void* arg) {
    printf("Hello, world!\n");
    return NULL;
}

Thread attributes (usually NULL)
Thread arguments (void *p)
return value (void **p)
/* silly squaring pthreads program */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

void* square(void* arg);

int main() {
    pthread_t tid;
    int *res;
    int i=5;

    pthread_create(&tid, NULL, square, &i);
    pthread_join(tid, (void**) &res);
    printf("the result is: %d\n", *res);
    exit(0);
}

void* square(void* arg) {
    int *sq;
    sq = malloc(sizeof(int));
    *sq = *(((int*)arg) * *((int*)arg));
    return (void*) sq;
}
Example 2

/* silly squaring pthreads program */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdlib.h>

void* square(void* arg);

int main() {
    pthread_t tid;
    int *res;
    int i=5;

    pthread_create(&tid, NULL, square, &i);
    pthread_join(tid, (void**) &res);
    printf("the result is: %d\n", *res);
    exit(0);
}

void* square(void* arg) {
     int *sq;
    sq = malloc(sizeof(int));
    *sq = *((int*)arg) * *((int*)arg);
    return (void*) sq;
}
How can you pass multiple args to a function with pthread_create?

You’d like to call this function when you start your thread:

```c
int find_max(int* array, int size);
```

But the start routine has to have this signature:

```c
void* (*start_routine) (void*);
```

How can you rewrite find_max as a start routine?
How can you pass multiple args to a function with pthread_create?

```c
struct thread_args{
    int* arr;
    int size;
};

void* find_max(void* arg){
    int* arr = ((struct thread_args*)arg)->arr;
    ...
}
```

Use a similar trick to have multiple return values
pthread_mutex_t

pthread_mutex_t m; // should be global

// two ways to initialize (only do one)
• m = PTHREAD_MUTEX_INITIALIZER;
• pthread_mutex_init(&m, NULL);

pthread_mutex_lock(&m);
// critical section code
pthread_mutex_unlock(&m);

pthread_mutex_destroy(&m);
`pthread_barrier_t`  

```c
pthread_barrier_t b; // should be global

// initialize with number of threads
pthread_barrier_init(&b, NULL, n_threads);

// section of thread parallel code
pthread_barrier_wait(&b);

pthread_barrier_destroy(&b);
```
In-class example of hello.c

```c
main:
    pthread_create(&tids[i], 0, thread_hello, &tid_args[i]);
    // creates a thread (thread_hello is function it will run)

thread_hello:
    // each spawned thread’s “main” function
    count += i; // count: a global var, all threads can access
    // i is local: each tid gets copy on
    // its private stack
```

vim hello.c
cd ~/cs31
mkdir week12
cd week12
cp -r ~mgagne1/public/cs31/week12/* .
cd week12
make
./hello 10  # run a few times & try with diff num
More pthread library functions

• Exit a thread (can also return from thread function)
  
  `pthread_exit`

• Wait until another thread sends a signal
  
  `pthread_cond_wait`
  `pthread_cond_signal`

  • These are tricky. We’ll do an example next week.
Exercise: implement your parallel algorithm for max.

Write c code using pthreads for main and a thread function that uses pthread_create and pthread_join.

- Array size: 10,000,000
- 5 threads
- Version 1: each thread returns its local max
- Version 2: threads compare each value to global max

```c
struct thread_in {
    int *arr;
    int th_num;
};
```
Example 2

/* silly squaring pthreads program */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void* square(void* arg);

int main() {
    pthread_t tid[2];
    int *res;
    int i=5;

    pthread_create(&tid[0], NULL, square, &i);
    pthread_join(tid[0], (void**) &res);
    printf("the result is: %d\n", *res);
    free(res);
    exit(0);
}

void* square(void* arg) {
    int *sq;
    sq = malloc(sizeof(int));
    *sq = *((int*)arg) * *((int*)arg);
    return (void*) sq;
}
Exercise: update your max solution to find the K largest items.

Write c code using pthreads for main and a thread function that uses pthread_create, pthread_join, and appropriate synchronization.

• Array size M
• N threads
• Fill an array with the K largest items

Try this one on your own!
Up Next

- Synchronization for common thread patterns
- Deadlock