CSCI 210 Systems Programming

Week 13

Threads

The Linux Programming Interface (Ch. 29)

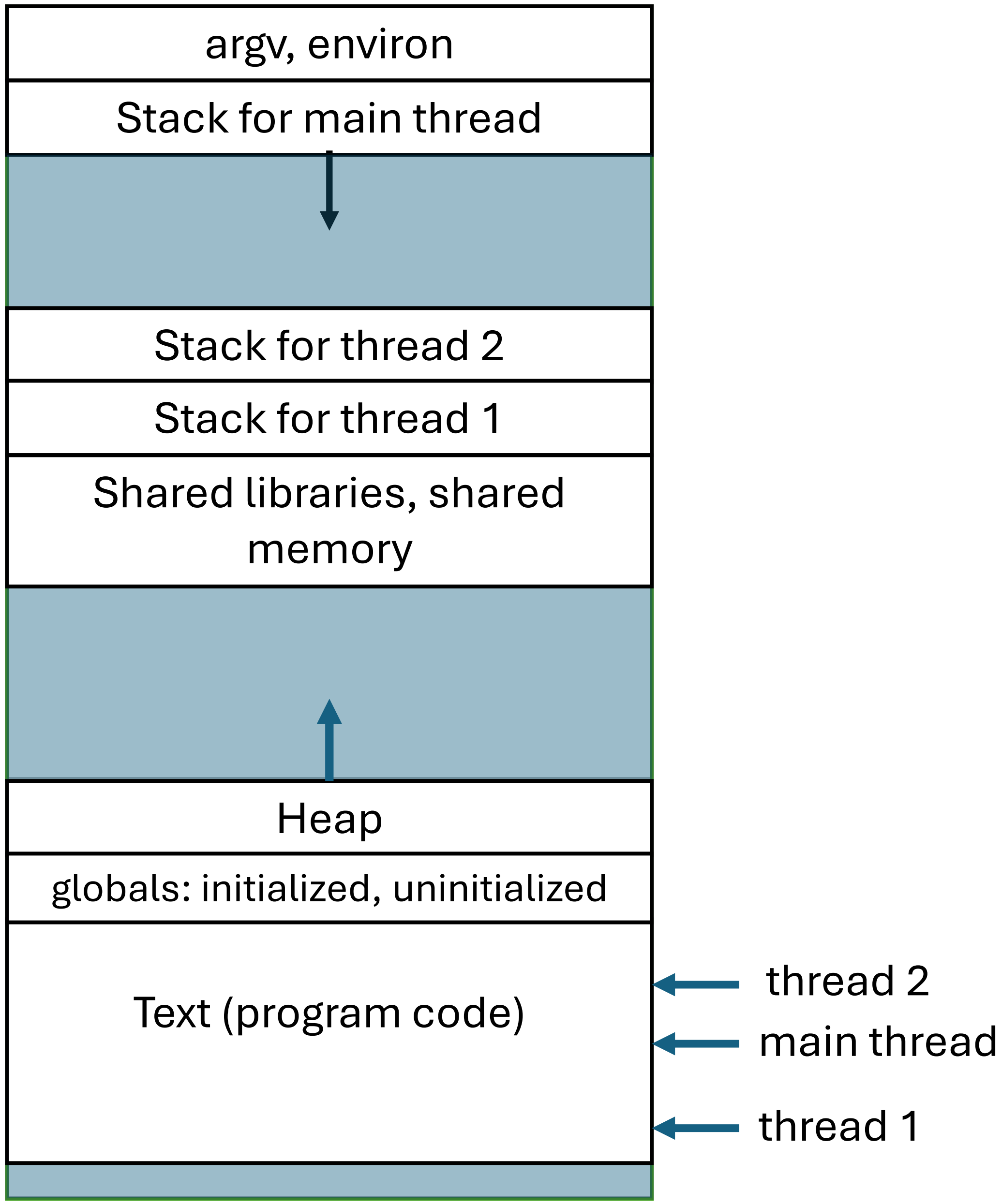
Systems Programming in Unix/Linux (Ch. 4)

Overview

- Introduction to threads
- Using the Pthreads API for multi-threaded programming
    - Pthreads data types
    - Thread creation
    - Thread termination
    - “Joining” with a terminated thread
    - Detached threads

Threads overview

- A thread is similar to a process in that it has its own execution path which is run "concurrently"
  - Parallelism vs. concurrency discussion
- Threads execute in a program
  - They share the same global memory
    - Global variables, heap segments
  - They have their own stack space (similar to function calls)



Advantages of threads

- Advantages of a multi-threaded program over a multi-process program are:
    - Sharing information between threads is easy and fast
        - Just a matter of copying data into shared (global or heap) variable
    - Thread creation is much faster (maybe 10 times or better) than process creation

Disadvantages of threads

- You have to synchronize your threads (or decouple them to work on different parts of shared data) manually. There is a whole chapter on Thread Synchronization (Ch. 30) on the LPI book.
- If there is a bug in one thread, this can damage all the other threads. Unlike processes they are not isolated from one another.
- They share and use the same virtual address space of the process. They compete to allocate resources.
  - Separate processes can each employ full range of available virtual memory

Attributes of a process shared by its threads

- Process ID, parent process ID
- Open file descriptors
- Process user and group IDs
- Signal dispositions (but threads can have their own signal masks)
- CPU time consumed
- … and others (see page 619 in the LPI book).

Attributes that are distinct for each thread

- Thread ID
- Signal mask
- The “errno” global variable
- Stack (local variables and function call linkage information)
- … full list on page 620 of the LPI book

The Pthreads API

- In 1980s and early 1990s, there were several different threading APIs
- In 1995, POSIX.1c standardized the POSIX threads API – Pthreads
- The Pthreads API defines a number of data types:

Data type	Description
`pthread_t`	Thread identifier
`pthread_mutex_t`	Mutex (for synchronization)
`pthread_attr_t`	Thread attributes object
`pthread_cond_t`	Condition variable

- .. and more but we will use only a couple of them in this course.

Compiling Pthreads programs

- You should use the –pthread option. Using this option has the following effects:
  - The `_REENTRANT` preprocessor macro is defined, which causes the compiler to use thread safe (i.e., re-entrant) versions of several functions in the C library.
  - The program is linked with the libpthread library (same as compiling it with the `–lpthread` option).

Creating threads

- When a program starts running, the process contains a single thread also known as the *main* thread. Additional threads can be created with the following function call:

```
#include <phread.h>

int pthread_create(pthread_t *threadid, const pthread_attr_t *attr,
                   void *(*start)(void *), void *arg);
```

- Returns 0 on success, or a positive error number on error

Example

```
#include <pthread.h>

void *mythread(void *arg) {
       int *a;
       a = (int *)arg;
       a[100] = 42;
       pthread_exit((void *)0); // normal thread exit - same as return
}

int main() {
       pthread_t tid;
       int *arr;
       arr = (int *)malloc(sizeof(int)*500);
       pthread_create(&tid, NULL, mythread, (void *)arr);
       printf("%d\n",arr[100]); // will it print 42? Maybe not :)
       return 0;
}
```

Arguments and return values

- Threads get their arguments with the `void *arg` argument
  - This is just a pointer to the data that is in the process address space – using heap space for this purpose is a good idea.
  - You can also send addresses of global variables and local variables in the main function (provided that exiting the main function terminates all the threads too – using return or exit() will do that).
- Threads return a pointer to some data if they want to return something
  - But this pointer should not point to threads local variables, since they are on thread's local stack, which is immediately freed and available for allocation by other threads

Terminating threads

- The execution of a thread terminates in one of the following ways:
  - The thread's start function executes the `return` statement
  - The thread calls `pthread_exit()`
  - The thread is canceled using `pthread_cancel()` – will not cover in this course
  - Any of the threads calls `exit()` or the main thread performs a return in the `main()` function, which will cause all the threads in the process to terminate immediately.

```
#include <phread.h>

void pthread_exit(void *retval);
```

Joining with a Terminated Thread

- Join has the same semantics as waitpid(), with some differences:
  - Threads are peers with other threads, which means they can wait for each other – there is no parent-child hierarchy when waiting for other threads to finish
  - A thread cannot wait for "any" thread to complete, specification of the id of the thread to join with (i.e. to wait for) is mandatory
  - The whole return value (the `void *`) is retrieved by the thread that joins with (i.e., reaps) the terminated thread, unlike the single exist status in processes.

```
#include <phread.h>

int pthread_join(pthread_t tid, void **retval);
```

Same example with something added

```
#include <pthread.h>

void *mythread(void *arg) {
        int *a;
        a = (int *)arg;
        a[100] = 42;
        pthread_exit((void *)0); // normal thread exit - same as return
}

int main() {
        pthread_t tid;
        int *arr;
        arr = (int *)malloc(sizeof(int)*500);
        pthread_create(&tid, NULL, mythread, (void *)arr);
        pthread_join(tid, NULL);
        printf("%d\n",arr[100]); // will it print 42? Definitely yes!
        return 0;
}
```

Zombie threads

- If a thread terminates and no other thread joins with it, it will become a zombie thread
  - In addition to wasting system resources, if enough zombie threads accumulate, we won't be able to create new threads
- But it is possible to "detach" a thread, if we don't care about the return value of that thread.
  - Detached threads are not "joinable" and they don't become zombies when they terminate.

Detaching a thread after creation

- There is a function to detach a thread after it is created with default attributes:

```
#include <phread.h>

int pthread_detach(pthread_t tid);
```

- Example:

```
pthread_detach(pthread_self());
```

Creating a "detached" thread

- We can use the `pthread_attr_t` type attributes, to specify a "detached" thread when creating it. Example:

```
pthread_t thr;
pthread_attr_t attr;

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_create(&thr, &attr, threadFunc, (void *)NULL);
pthread_attr_destroy(&attr);
```

Exercise 1

- What will happen if a thread executes the following code:

```
pthread_join(pthread_self(), NULL);
```

- Try it by writing a program that executes this line.

Exercise 2

- What are the potential problems with the following program?

```
struct vec2 { int x,y; };

void * threadFunc(void *arg) {
        struct vec2 *pbuf = (struct vec2 *) arg;
        pbuf->x = 5;
        pbuf->y = 8;
}
int main(int argc, char*argv[]) {
        pthread_t tid;
        struct vect2 buf;
        pthread_create(&tid, NULL, threadFunc, (void *) &buf);
        pthread_exit(NULL);
}
```

Examples

- From the textbook:
  - Summing up all elements of an NxN matrix
  - Multi-threaded quicksort