Threads

CS241 Discussion Section
Week 3
September 17, 2009

Outline

- MP1 Issues
- MP2 Overview
- pthreads
- Thread Safety

MP1 Issues

- How to store history, when you are not supposed to assume the size of history?

MP1 Issues

- How to store history, when you are not supposed to assume the size of history?
- Two solutions -
  - linked list
  - Array + realloc()
- Remember to free memory when you exit

MP Issues (Contd..)

- How to determine whether the command run is a valid one or not?
  (i.e. ls –a vs ls –abcde)

MP Issues (Contd..)

- How to determine whether the command run is a valid one or not?
  (i.e. ls –a vs ls –abcde)
- Check the return value of the exec system call

MP Issues (Contd..)

- Why does my PID change or get set to zero after executing the code?

MP Issues (Contd..)

- Why does my PID change or get set to zero after executing the code?
- When you enter a command, the shell checks to see if it is a built-in command, and if so it executes it. If it is not a built-in the shell forks a new process in which to execute the command.
- Therefore, if you run a built-in command in a forked process you will see the pid 0

MP2 Overview

MP2 is an introduction to threads

Goal: sort an enormous data set in parallel using threads

File I/O

I/O in C

MP2 requires you to read and write simple file in C.

Two primary means of doing I/O in C:

- Through lightly-wrapped system calls
  - open(), close(), read(), write(), etc
- Through C-language standards
  - fopen(), fclose(), fread(), fwrite(), etc

I/O in C

Opening a file (Method #1):

fopen(const char *filename, const char *mode);

- **filename**: path to file to open
- **mode**: what do you wish to do with the file?
  - "r": read only
  - "r+": read and write (file must already exist)
  - "w": write (or overwrite) a file
  - "w+": write (or overwrite) a file and allow for reading
  - "a": append to the end of the file (works for new files, too)
  - "a+": appends to end of file and allows for reading anywhere in the file; however, writing will always occur as an append

I/O in C

Opening a file (Method #2):

open(const char *filename, int flags, int mode);

filename: path to file to open

flags: what do you wish to do with the file?

- One of the following is required:
  - **O_RDONLY**, **O_WRONLY**, **O_RDWR**
- And any number of these flags:
  - **O_APPEND**: Similar to "a+" in fopen()
  - **O_CREAT**: Allows creation of a file if it doesn't exist
  - **O_SYNC**: Allows for synchronous I/O (thread-safeness)
- To "add" these flags, simply binary-OR them together:
  - (**O_WRONLY** | **O_APPEND** | **O_CREAT**)

mode: what permissions should the new file have?

- (**S_IRUSR** | **S_IWUSR**) creates a user read-write file.

Opening Files in C

Return value of opening a file:

Having called open() or fopen(), they will both create an entry in the OS's file descriptor table.

Specifics of a file descriptor table will be covered in-depth in the second-half of CS 241.

Both open() and fopen() returns information about its file descriptor:

open(): Returns an int.

fopen(): Returns a (FILE *).

Reading Files in C

Two ways to read files in C:

fread(void *ptr, size_t size, size_t count, FILE *s);

*ptr: Where should the data be read into?	C-string?
size: What is the size of each piece of data?	sizeof(char)?
count: How many pieces?	strlen(str)?
*s: What (FILE *) do we read from?	ptrFile?

read(int fd, void *buf, size_t count);

fd: What file do we read from?

*buf: Where should the data be read into?

count: How many bytes should be read?

Reading Files in C

Reading more advancely…

fscanf(FILE *stream, const char *format, …);

Allows for reading at a semantic-level (eg: ints, doubles, etc) rather than a byte-level.

The format string (*format) is of the same format as printf().

No equivalent command for open().

Writing Files in C

Writing is a lot like reading…

- fwrite(void *ptr, size_t size, size_t count, FILE *s);
  - Writing of bytes with (FILE *).
- write(int fd, void *buf, size_t count);
  - Writing of bytes with a file descriptor (int).
- fprintf(FILE *stream, const char *format, …);
  - Formatted writing to files (works like printf()).

Closing Files in C

Always close your files!

- fclose(FILE *stream);
- close(int fd);

Failure to close a file is much like a memory leak, but may corrupt files rather than memory.

- If a file is written to, the program ends, and the file is never closed: the write never may never be written!
  - Reason: write(), and especially fwrite()/fprintf(), may be buffered before being written out to disk.
  - On close()/fclose(), all buffers are cleared and the file is properly closed.

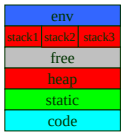
Threads vs. Processes

What are the main differences?

- Memory / address space
- Scheduling (concurrent execution)
- Overhead

When should we use a process?

A thread?



Threads versus Processes

Property	Processes created with fork	Threads of a process	Ordinary function calls
variables	get copies of all variables	share global variables	share global variables
IDs	get new process IDs	share the same process ID but have unique thread ID	share the same process ID (and thread ID)
Communication	Must explicitly communicate, e.g.pipes or use small integer return value	May communicate with return value or shared variables if done carefully	May communicate with return value or shared variables (don't have to be careful)
Parallelism (one CPU)	Concurrent	Concurrent	Sequential
Parallelism (multiple CPUs)	May be executed simultaneously	Kernel threads may be executed simultaneously	Sequential

Threads vs. Processes

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Each thread execute separately

Threads in the same process share resources

No protection among threads!!

POSIX Threads (Pthreads)

Standardized, portable thread API

To use POSIX thread functions

```
#include <pthread.h>
gcc –o main main.c -lpthread
```

Creating a Thread

When a new thread is created it runs concurrently with the creating process.

When creating a thread you indicate which function the thread should execute.

Creating a thread with pthread

A thread is created with

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

The creating process (or thread) must provide a location for storage of the thread id.

The third parameter is just the name of the function for the thread to run.

The last parameter is a pointer to the arguments.

Problem 1

Hello World! (thread edition)

We'll create two threads and one will print out "Hello", and the other "World".

Problem 1

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

void *hello_thread(void *arg)
{
    fprintf(stderr, "Hello ");
    return NULL;
}

void *world_thread(void *arg)
{
        fprintf(stderr, "World!\n");
        return NULL;
}

int main(int argc, char **argv)
{
        pthread_t hello, world;
        pthread_create(&hello, NULL, hello_thread, NULL);
        pthread_create(&world, NULL, world_thread, NULL);
        return 0;
}
```

Problem 1

Hello World! (thread edition)

(Code provided as "p1-hello.c")

What's wrong?

Hint: how many threads are there?

Solution: Joining Threads

Remember the orphan concept?

The parent thread exited and the child threads were orphaned (and killed by the OS).

We can wait on threads, too, but we call it "joining":

```
int pthread_join(pthread_t thread,
    void **value_ptr);
```

Solution: Hello World

Let us see p1-hello-soln.c

Problem 1

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

void *hello_thread(void *arg)
{
    fprintf(stderr, "Hello ");
    return NULL;
}

void *world_thread(void *arg)
{
        fprintf(stderr, "World!\n");
        return NULL;
}

int main(int argc, char **argv)
{
        pthread_t hello, world;
        pthread_create(&hello, NULL, hello_thread, NULL);
        pthread_create(&world, NULL, world_thread, NULL);
        pthread_join(hello, NULL);
        pthread_join(world, NULL);
        return 0;
}
```

Detaching Threads

We have another option:

`int pthread_detach(…);`

Lets the system reclaim the thread's resources after it terminates

Good practice – call `pthread_detach` or `pthread_join` for every thread you create

Exiting the process

We know what happens when main returns.

But what if one of the other threads calls `exit()`? Try it out.

Hello World!

Everything seems to work; “Hello World!” is printed out correctly.

But does it *always* work?

- More on this later…

Lets take a small detour to

Function Pointers

Aside: Function Pointers

Let’s again take a look at pthread_create():

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

What is start_routine?

- A Function?

Passing Functions in C

Yes, it is a function.

- When calling pthread_create(), you must pass a function:
  - pthread_create(…, …, void *(*start_routine)(void*), …);
- What is `void *(*start_routine)(void*)`?
  - A parameter that requires a `void *__(void *__)` function!
  - Ex:

```
void *myfunc(void *vptr)
{
    printf("I'm a thread!");
}
```

Passing Functions in C

In this MP, you must use qsort()…

The qsort() function definition looks like:

```
void qsort (void *base, size_t num, size_t size,
            int (*comparator)(const void *, const void *));
```

What is `int (*comparator)(const void *, const void *)`?

A parameter that requires a function of the following format:

```
int ___(const void *___, const void *___);
```

That function should return:

(negative) if the (first param) < (second param)
0 if the (first param) == (second param)
(positive) if the (first param) > (second param)

Passing Functions in C

Comparing Strings:

A function that could be accepted by qsort() to compare strings is:

```
int mystrcmp(const void *s1, const void *s2)
{
   return strcmp((char *)s1, (char *)s2);
}
```

In MP #2, you will write a function for integers.

Detour ends

Back to threads..

Back to threads…

It is possible, in theory, for our program to print “World!” before “Hello”.

How can we fix this?

Passing Arguments to Threads

`void *arg`

- Pointer to any data type
- Have to cast it to a specific pointer type before dereferencing

Let's make `world_thread` wait on `hello_thread` (using pthread_join).

Thread Return Values

Threads return a `void*`, too. Return value can be retrieved by pthread_join

- Be careful about not returning pointers to local variables!

Have each thread return a pointer to the string they print out.

- Print these out again in main().

Can threads have more than one argument?

Yes! Sort of.

We can pass a pointer to a struct, e.g.:

```
typedef struct {
    int arg1;
    char *arg2;
} myargs;
myargs a;
pthread_create(…, myfunc, &a);
```

Aside: Review of structs in C

Keyword `struct` used to define complex data types.

Structs can contain variables, arrays, pointers, other structs…

Can structs contain pointers to functions?

- Does that remind you of anything?

Thread Entry Points

A function that is used to start a thread must have a special format.

```
void* threadfunction(void* arg);
```

`void*` - can be a pointer to anything

- Use a pointer to a `struct` to effectually pass any number of parameters.
- Return any data structure too.

Problem 2: Remember linked lists?

Functions to create and print a linked list are provided (p2-list.c)

- Recreate the list_node struct

Now, create two threads to do this

- Thread 1 calls create_list
- Thread 2 calls print_list
- Don't use any global variables!

Problem 2 Code

```
node *create_list()
{
  node *head, *cur;
  int i;

  for(head = NULL, i = 10; i >= 1; i--) {
    cur = (node *)malloc(sizeof (node));
    cur->value = i;
    cur->next  = head;
    head = cur;
  }
  return head;
}
```

```
void print_list(node *head)
{
        while (head) {
                printf("%d\n", head->value);
                head = head->next ;
        }
}
```

Problem 2 Code

```
struct list_node {
   /* TASK 1: What goes in a linked list node? */
};

typedef struct list_node node;

int main(int argc, char **argv)
{
   node *list;

   /* TASK 1: How do we create two threads and get return
   values? */
   list = create_list();
   print_list(list);

   return 0;
}
```

Solution

Let us see the solution p2-list-soln.c

```
/* list = create_list(); */
pthread_create(&thread_id, NULL, create_list, NULL);
pthread_join(thread_id, &list);

/* print_list(list); */
pthread_create(&thread_id, NULL, print_list, list);
pthread_join(thread_id, NULL);
```

Concurrency

Threads execute concurrently

- True concurrency on multiple processors
- Interleaving on a uniprocessor machine

All memory, except the stack, is shared between the threads in a process

- What happens if multiple threads access a shared variable concurrently?

Problem 3: Modifying a shared variable

Write a program with global variable `x = 0`

- One thread increments it N times (`x++`)
- One thread decrements it N times (`x--`)
- main() joins the threads and prints out x

Is the value of x always 0?

- Run it a few times each with N = 100, 1000, 1e5, 1e6, …

What is going on?

Thread 1	Thread2
`x++;`	`x--;`

What is really going on

Thread 1		Thread2	
`read x`	(100)		
`Increment`	(101)		
Context switch!	→	`read x`	(100)
		`Decrement`	(99)
		`write x`	(99)
`write x`	(101)	← Context switch!	

How to solve this problem?

•We will see more about this in the next discussion section and MP

•But, if you are curious you can read about locks and semaphores

A few useful Pthreads functions

POSIX function	Description
pthread_create	create a thread
pthread_detach	set thread to release resources
pthread_equal	test two thread IDs for equality
pthread_exit	exit a thread without exiting process
pthread_join	wait for a thread
pthread_self	find out own thread ID