

COMP421

Unix Environment for Programmers

Lecture 15: Threads

Jeff Wiegley, Ph.D.

Computer Science

jeffw@csun.edu

10/17/2005

‘‘Chains do not hold a marriage together. It is threads, hundreds of tiny threads, which sew people together through the years..’’

–Robert Oxton Bolt

Multiprocessing

There are many problems that are solved best by parallel execution of tasks.

- Web servers can process a greater number of requests if they are handled simultaneously.
- GUI interfaces can display and update multiple pieces of information with less code complexity if the tasks are treated independently.

Historically, Unix handled these through its standard process model of forked behavior.

Each task handled by an independent (yet related) process.

fork() disadvantages.

Although the `fork()` solution works, there are a number of disadvantages associated with it:

- Creation and Destruction of whole processes is computationally expensive.
- Processes do not share anything in common. (Everything is a copy; a child cannot “see” any of its parent’s resources.)
- Interprocess communication is complicated and slow.
- The context switch to another process is time consuming.
- Concurrency is complicated, difficult to manage, difficult to debug and error prone.

But time and problems and march on.

The need for multiprocessing and asynchronous computation have increased to the point where Unix must evolve to maintain efficiency.

Threads:

The instructor believes that Unix owes a significant portion of its success and longevity to its excellent original design.

Without recreating a new operating system from scratch Unix addressed the need for “light-weight” processes by introducing the concept of *threads*.

Threads are like a new process but the creation of thread doesn't create a whole new process.

Instead the thread is created within the original process.

This requires some change in the kernel's process table, scheduling and system calls but aside from this the rest of Unix remains relatively unchanged.^a

^aUnix is also going to be able to adapt with relatively minor changes to support future security models based on roles, resources and policies rather than objects and permissions. (SELinux).

Thread advantages:

Threads provide the following advantages over forked processes:

- Light-weight. thread creation and destruction requires a fraction of the time it would take to fork an entirely new process.
- threads within a process share code and heap space. Thus all threads share the same code and same global variables.
 - But this means you have to be careful about concurrency issues when you manipulate global variables.
- Each thread has an independent register set and stack. So they execute independently from each other.^a
- Thread libraries have built-in routines for implementing concurrency control.
- The ‘P’ in pthreads stands for POSIX. So coding a threaded program in compliance with POSIX standards produces platform (Unix) independent code.

^aStacks can “see” each others stacks as well; all memory is shared.

Thread creation:

Threaded programs start as any other process. The process just happens to consist of a single thread.

Creating more threads is really easy:

```
int err;
pthread_t ntid;
err = pthread_create(&ntid, NULL, thread_function, NULL);
```

- The new thread starts executing in the function `thread_function()`. (This is different from `fork()` where both processes execute from the same line of code.)
- The thread terminates:
 - explicitly, when it calls `pthread_exit()`, or
 - implicitly, if it returns from the `thread_function()` function.
- The thread ID^a of the new thread is returned in the structure `ntid`.

^aJust as every process has a “process ID”; every thread has a “thread ID” assigned to it.

Simple Examples:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <pthread.h>

pthread_t ntid;

void printids(const char *s)
{
    pid_t    pid;
    pthread_t tid;

    pid = getpid();
    tid = pthread_self();
    printf("%s pid %u tid %u (0x%x)\n", s, (unsigned int)pid,
          (unsigned int)tid, (unsigned int)tid);
}

void *thr_fn(void *arg)
{
    printids("new thread: ");
    return((void *)0);
}
```

```
int main(int argc, char *argv[])
{
    int err;
    err = pthread_create(&ntid, NULL, thr_fn, NULL);
    err = pthread_create(&ntid, NULL, thr_fn, NULL);
    printids("main thread: ");
    sleep(1);
    exit(0);
}
```

output:

```
$ gcc threadtest.c -lpthread
$ ./a.out
main thread:  pid 29739 tid 3084707520 (0xb7dce6c0)
new thread:   pid 29739 tid 3084704688 (0xb7dcdbb0)
new thread:   pid 29739 tid 3076311984 (0xb75ccb0)
$
```

Notice that the pid of each thread is the same. The threads are distinguished within the process by their thread id instead.^a

^aThe data in the textbook is no longer up-to-date. Linux does not use `clone()` to implement threads in kernel version 2.6 and thus process ids are identical.

Concurrency issues still arise:

Although thread synchronization is simpler in threads concurrency issues still arise. The previous sample took care of two problems:

1. When the main thread exits, all threads exit. So a call to `sleep(1)` has been inserted into the main thread to give the other threads time to wake up and complete their task.^a
2. Although the thread id is returned in `ntid` and `ntid` is global it would seem possible for the child thread to obtain its thread id by reading from `ntid` directly. However, if the thread begins executing before the main thread has had time to assign the result into `ntid` then the child would read an initialized value. So a call to `pthread_self()` is used instead.

^aThe behavior of killing all threads when the main thread exists is platform and implementation dependent.

Thread termination:

Any call to `exit()`, `_Exit()` or `_exit()` terminates the entire program (and all threads).

A thread can terminate itself without terminating the program in any of three ways:

1. Simply return from the start function that was passed to `pthread_create`.
2. A thread can be cancelled by another thread in the same process.
3. A thread can call `pthread_exit(coid *rval_ptr)`.

`pthread_join(pthread_t thread, void **rval_ptr)` is a handy function.

The calling thread blocks until the thread with thread id *thread* exits. `rval_ptr` returns the reason for the threads exit.

Instead of an arbitrary length call to `sleep()` it is better for the main thread to `join` with every thread that was created before exiting.

rval_ptr warning:

`rval_ptr` is just a mechanism to pass any memory pointer from the exiting thread to the joining thread.

WARNING: Using a structure allocated on the exiting thread's stack is dangerous. The stack can be deallocated before the joining thread is done with the data passed to it.

You can solve this problem in a number of ways:

- Use a globally allocated structures.
- Use a structure allocated by `malloc`.^a
- Arrange to communicate and coordinate about other persistent memory locations in the joining threads stack before the exit occurs.^b

^athe overall heap is fully shared.

^bAs mentioned early, the stacks aren't private. You would just have to be told of a valid position in the stack. But then, actually, you wouldn't need `rval_ptr` would you?

Cancelling:

Any thread can request that another thread be cancelled by calling:

```
void pthread_cancel(pthread_t tid);
```

Any thread can elect to ignore or otherwise control how it is cancelled.

`pthread_cancel()` does not block until the thread cancels. instead it is merely a request.

Cleaning Up:

Threads can arrange to have a list of functions that are called when it exits.

This is useful for performing clean-up tasks prior to execution.

Two functions are provided for manipulating the stack of functions to be called.

```
void pthread_cleanup_push(void (*rtn)(void *), void *arg);  
void pthread_cleanup_pop(int execute);
```

`push` pushes the function *rtn* onto the stack of functions to be called.

`pop` takes the last pushed function off the stack. If *execute* is non-zero then the function is executed in conjunction with the pop.^a

The functions are executed in the reverse order that they were pushed on. (As one would expect from a stack → “LIFO”).

^apush and `pop` can be implemented as macros. This means every `push` must have a `pop` associated with and within the same nesting level. Otherwise curly braces in the macros don't match up and the program doesn't compile.