Unix System Programming - Chapter 12

Neal Nelson

The Evergreen State College

Apr 2010

USP Chapter 12 - POSIX pthreads

- Section 12.1 - Concurrency example: monitoring file descriptors
- Section 12.2 - pthread example implementation
- Section 12.3 - Thread management
- Section 12.4 - Thread safety
- Section 12.5 - User Threads vs Kernel Threads
- Section 12.6 - Thread Attributes
- Section 12.7 - Parallel File Copy Exercise

Concurrency example: monitoring file descriptors

- A program is monitoring input from two sources
- In Unix IO model this means monitoring two file descriptors
- File reads usually block, but we can only block at one or the other fd
- We can't know which one to block on
- Six general approaches in USP

Approaches to the multiple file descriptor problem

- A separate process monitors each file descriptor - processes don't share address space, so this technique works well when IO stream processing is independent. If communication is required then you need to use interprocess communication (IPC) facilies like message passing (Chapter 15).
- `select` function (Chapter 4) - code can be complicated.
- `poll` function (Chaper 4) - code can be complicated.
- Nonblocking IO with polling (Chapter 4) - hard to get the timing right on polling intervals.
- POSIX asynchronous IO (Chapter 8) - using signal handlers is difficult to implement and prone to errors.
- Separate threads - as simple as separate processes, but inter-thread communication options via shared variables is available because threads share address space. You need to use thread synchronization facilities like semaphores, mutex locks, and condition variables (Chapter 13).

pthread example - Multiple File Descriptor Monitor

- Fig 12.1 - processfd with ordinary function call, single thread
- Fig 12.2 - processfd with pthreads, two threads
- Program 12.1 - processfd code for monitoring a single file descriptor
- Example 12.1 - single thread calling processfd function.
- Example 12.2 - two threads, each calling processfd function.
- processfd code is the same for single or double threads.
- pthread_create is analogous to fork - pass function and arguments and get back tid.
- Program 12.2 monitorfd.c - monitor an array of file descriptors using a separate thread for each file descriptor.

pthreads - management functions

- pthreads functions don't use errno, so can't use perror
- You can use strerror, but you have to obey thread safety issues (Section 12.4)
- No POSIX thread function returns EINTR, so they don't have to be restarted if interrupted by a signal.
- `pthread_t` is the type of a thread ID (tid).
- Use `pthread_self` to get your own thread id
- `pthread_t` may be a structure, so you can't just treat it like int.
- Use `pthread_equal` to compare thread IDs

pthread_detach and pthread_join

- When a thread exits, it does not release its resourses unless it is a detached thread.
- `pthread_detach` sets a target thread's internal options to specify that storage for the thread can be reclaimed when the thread exits.
- A thread can call pthread_detach on itself or another thread.
- `pthread_join` joins a target thread to the caller, suspending the calling thread until the target thread terminates.
- You cannot call a pthread_join on yourself; expect an error or a deadlock. POSIX does not require error detection in this case.
- pthread_join releases resources of the target thread when it exits.
- pthread_join is analogous to a waitpid for processes.
- All threads should be either detached or joined to prevent space leaks.
- Try writing a program to use the cute example 12.6 p418 - detachfun.

pthread_exit

- Any thread that calls exit exits all threads and the process
- Threads can individually call `thread_exit` or just return from its top level code as an implicit thread_exit.
- When a thread exits, it does not release its resourses unless it is a detached thread or it is a joined thread that later will have its resources released.
- Threads can pass back an address of an exit code, but this is a bit tricky - the address must point to memory that the caller can see, not local storage that has disappeared.
- Threads should be designed to properly coordinate their mutual exit and cleanup of resources using pthread_detach and pthread_join.

pthread_cancel

- `thread_cancel` can request a target thread be cancelled.
- Cancellation can cause difficulties if a thread holds resources such as a lock or a file descriptor that must be released before exiting.
- Threads can enable or disable cancellation when they are the target of cancellation requests.
- Any function that changes its cancellation state should restore the value before returning.
- There is also a cancellation type that can be set so threads can defer cancel requests to safe times when they are the target of cancellation.
- Safe cancellation times can be set by a target thread at particular code points by calling `pthread_testcancel`.

Passing parameters to threads

- A thread creator may pass a single parameter to a thread at creation time, using a pointer to void (USP p422).
- To communicate multiple values, the creator must use a pointer to an array or structure (USP p422).
- See the `fds` array declaration and argument to pthread_create in program 12.4, callcopymalloc.c
- On invocation of the copy thread in program 12.5, copyfilemalloc.c, see the `infd` and `outfd` extraction of the parameters from the passed array in the (void *). You need a cast, of course.
- The argument extraction of Program 12.7, copyfilepass.c, is much better (clearer) in my view.

Returning results from threads

- Threads can pass back an address of an exit code, but this is a bit tricky - the address must point to memory that the caller can see, not local storage that has disappeared.
- Threads share address space, so thread exit does not release malloc memory of the thread.
- A thread can malloc a block and return the pointer to the caller on the stack using a return statement.
- See Program 12.5, copyfilemalloc.c return value `int *bytesp`.
- Where does the return value show up in the caller? See the callcopymalloc.c `bytesptr`
- This works for threads and not processes because threads share address space and processes do not.
- Exercise 12.9 - can you pass a value back via pointer to a variable with static storage class?
- Sure, but then what if you create two copies of this thread code?

Memory management among threads

- In the return parameter example above a created thread did malloc and a thread creator had to do the corresponding free.
- Whenever possible, a thread should clean up its own mess rather than requiring another thread to do it (USP p424).
- Have the creator malloc and pass space for the return parameter into a created thread.
- With non-aggregate values the caller can do this with a stack-local variable space! So no malloc in the caller is needed.
- See Program 12.6, callcopypass.c, the `targs` parameter to pthread_create now has an extra slot in the argument array for the return value.
- See the Program 12.7 copyfilepass.c return statement passing back the byte count of the copy as the address of slot 3 of the array.
- Also examine the much nicer alternative method of accessing thread arguments in the copyfilepass.c program via argint[2].

Parallel File Copy, Program 12.8 copymultiple.c

- When creating multiple threads, do not reuse the variable holding a thread's parameters until you are sure that the thread has finished accessing the parameter.
- Rule - use a separate variable for each thread.
- Program 12.8, copymultiple.c uses an array of copy_t struct to pass a separate args array space to each of its subordinate treads.
- See the `copy_t` struct and the `copy_t copies` array variable with its `(copy_t *) calloc`
- Notice the pthread_self() used for failure notification in the thread creation loop where it is set and then in the join loop where it is tested to skip the join.
- Program 12.9 badparameters.c and Exercise 12.12, 12.13, 12.14 look like informative reading and study. Have fun trying those exercises and testing your understanding.

Thread Safety

- A function is thread-safe if multiple threads can execute simultaneous active invocations of the function without interference (USP p431).
- POSIX specifies that all required functions, including the functions from the standard C library, be implemented in a thread-safe manner **except** for the specific functions listed in Table 12.2 (USP p431-432).
- You can't use perror because POSIX pthreads don't set errno
- You can't assume strerror is thread-safe because it is in Table 12.2
- Propogate errors. Use strerror in the main thread when it is safe.
- Example - use strerror after pthread_create and pthread_join.

Thread Safety

- Thread safety when using shared variables requires synchronization facilities, such as mutex locks, condition variables and semaphores discussed in USP Ch 13 and Ch 14
- Ideas on error propogation from USP p432 quoted in the following bullets.
- In most thread implementations errno is a macro that returns thread-specific information, so each thread has a private copy of errno
- The main thread does not have direct access to errno for a joined thread, so if needed, this information must be returned through the last parameter of pthread_join.

User Threads vs Kernel Threads

- User level threads all run in one process, so any thread blocks all threads.
- Jacketing thread library functions allows blocking threads to be intercepted and handled by scheduling outside the process.
- The jacketing functions actions are invisible to both the user and the operating system.
- The jacketing functions are essentially a run-time thread management shim between a user level multi-threaded process and the operating system.
- Read details on USP p433 bottom.
- Another problem with user level threads is that a CPU bound thread can keep the jacketing functions from being called and muck up the thread scheduling that is being done by them outside the process.
- User-level threads don't actually get any parallelism, only concurrency.

User Threads vs Kernel Threads

- Kernel threads are implemented by the operating system.
- Kernel threads give real parallelism as well as concurrency.
- Kernel threads are almost as expensive as processes because they're done by the OS.
- Kernel threads are like processes, but now share address space.
- Kernel threads support more efficient shared memory synchronization
- Kernel threads require synchronization primitives like semaphores, locks, and so forth to be available to the programmer who must handle the thread-safety.
- Hybrid thread models like Solaris allow a user-defined mapping between user-level threads and kernel threads
- The POSIX pthread standard has a thread attribute option that supports the mapping of user threads to kernel threads.
- See PTHREAD_SCOPE attributes and functions, USP p435-436.

Thread Attributes

- POSIX pthread attributes are encapsulated in a `pthread_attr_t` object.
- The attribute object only affects a thread at the time of creation.
- The thread attribute object is passed as the second parameter of pthread_create.
- Table 12.3 p436 lists settable POSIX thread attributes. Sample: stack size, scheduling.
- Note that pthread_attr_init sets errno, contrary (sort of) to the USP p415 claim that they [most pthread functions] do not set errno.
- The thread_attr_t methods are discussed in USP pp437-443 if you need them.

Assignment Exercise 12.7: Parallel File Copy

- This exercise starts with Program 12.8, copymultiple.c. As usual, make a separate directory for this program, copy over and pare down the makefile, and play with the program until you fully understand it.
- When your extracted copy of copymultiple.c works, create a development directory for the first version of your copydirectory.c program in step 1.
- Develop the copydirectory program described in steps 1 and 2.
- Be sure to use only thread-safe calls in this implementation (see table 12.2 for what NOT to use)

Assignment Exercise 12.7: Parallel File Copy

- Test your programs thoroughly and describe the tests you performed in your lab writeup.
- When you submit this lab you should have a makefile that allows you to make each version of your program with a separate target name. You should also have a make all at the beginning. Use the USP makefile as an example to model.
- Replace the lint stuff with splint. You'll need to select different options for splint than those listed in the USP makefile for lint.
- You will need to figure out how to walk through the files in a directory. This is where USP Ch 5 research will come in handy.
- Study the USP Program 5.3 shownames.c p 153 to see if that will work for you.

Assignment Exercise 12.7: Parallel File Copy

- Your first version of the main test program for copydirectory will be a loop with one copydirectory thread created and joined each time through the loop (steps 1,2).
- Your second version will create the target directory if it does not exist (step 3). This is really a minor variation of the first version, so you do not need to report on this if you get it to work.
- Your third version will create a separate thread for each file to be copied (step 4).
- This third version is your first version of a parallel file copy, so test it thoroughly and describe your tests in your lab writeup.
- Do not go beyond this step in the exercise unless you are so excited you just can't keep yourself from forging ahead. But remember, the parallel virtual machine of Chapter 17 is waiting for you!

Done

- Done.