

Maximum Multithreading - Thread Programming in UnixWare 2.0

With the advent of UnixWare 2.0, threads have finally made their way to the desktop Unix world. A superset of the thread specification in POSIX Portable Operating Systems Standard (draft standard P1003.1c), threads have the potential to liberate UnixWare developers from the limitations of the age-old **fork()** model. Along with complementing this ancient spawning mechanism, threads also give developers the opportunity to exploit the capabilities of multi-processing hardware. Freedom and opportunity have their price, though, in this case, a set of issues and possibilities that UnixWare developers have never had to consider before.

The fork/exec Model:

Prior to 2.0 (POSIX 1003.1c and SVR4.2MP), UnixWare had two methods of creating new processes, fork and fork-exec. The fork system call creates an exact copy of the calling process and sets it running at the return from the fork call. The new process gets a copy of the parents data space and valid file descriptors for all files opened by the parent. The new process is also a child of the old process. If you want to start a different process, the child process calls exec right after the return from fork.

Creating a new process consists of a few lines of code like those in the following code snippet. What a process has to do to start another process, is clone itself, then ask the operating system which of the two copies it is.

```
if(( child_pid = fork() ) != 0 )
    // do child process stuff
    exec( "new_program" ); // overlay this clone with a new executable
else
    // continue doing parent process stuff
```

While simple enough, to my mind, this put the ug in ugly. The set of cases you're trying to cover with fork() is those cases where you want a new, concurrent process. The cases where you might want a clone of the original process is a trivial subset of that set. Until recently, though, fork/exec has been the only avenue for concurrent programming.

Light-Weight Processes

In order to understand some of the finer points of threads, you need to know something about the underlying kernel mechanism upon which threads are based. Pre 2.0 UnixWare kernels had only one type of process, what I now dub a heavy-weight process and the object of such calls as ps, kill() and getpid(). Heavy-weight processes still exist, but only as collections of light-weight processes. Light weight processes, or LWPs, are the only schedulable entity in the UW2.0

universe. A heavy-weight process consists of from 1 to MAXULWP (see below) light-weight processes. If you run a non-threaded application in UW2.0 you will get, in memory, a heavy-weight process that consists of a single light-weight process. In effect, instead of being a pointer to a piece of executable code, a heavy-weight process is now a pointer to a list of pieces of executable code.

In a multi-processor system, separate LWPs from a single HWP can run on different processors. For the user, this means that separate functions within a single program can achieve true concurrency. The best example of the need for this is the print function. The user wants to hit the print button, then move on, not sit there watching a stupid dialog box say "Now formatting page XX ... please wait".

Since the heavy-weight process concept is still supported, the old process-specific calls like `getpid`, `kill` and `nice` still work much as they did before. That being the case, the threads programmer needs analogues to those calls in order to control threads and their LWPs the way he's always controlled HWPs. Table 1 shows a list of some process control calls and their threads lib analogues.

There is an LWP API, (I use one `_lwp` call in the sample listings), but you can't use it. Actually you can, but shouldn't. The threads lib is the portable, standards following interface. Use `_lwp` calls at your own risk.

Threads

Threads are not LWPs. The kernel itself knows nothing about threads, it only schedules LWPs. Each running LWP makes calls to the dynamic threads library which schedules threads to run on LWPs. So you now have two levels of scheduling, kernel scheduling of LWPs on processors, and threads lib scheduling of threads on LWPs. A single instance of a thread can run, at different points in its life, on different processors, and on different LWPs. In order to really get this, you have to view the scheduled process as something completely independent from the lines of code that will run when that process gets scheduled. Think of the processor as a field, each LWP is someone who has signed up to use the field, and each thread is a particular activity such as baseball, soccer or football. Now when the kernel schedules someone to use the field, that person can play football for his whole time (a bound thread), or he can play football for five minutes and baseball for ten minutes. The kernel doesn't care. The person using the field (the threads library through any of its LWPs) has to keep track of the games being played (thread instances) within the time that he's using the field. Thus, a thread is simply a series of logical statements, independent of the process, or processor upon which it might be executed.

Bound or Multiplexed?

There are two basic kinds of threads, bound and multiplexed. A bound thread is one that gets its own dedicated LWP. A multiplexed thread (muxthread) can run on any LWP in the HWP's LWP pool. Each HWP has a number of LWPs in its pool, and it can run a particular muxthread on any LWP in the pool at any given point.

The major consideration in choosing between bound and multiplexed threads is the tradeoff between performance and concurrency. On a uniprocessor, bound threads can have up to five times the context-switching overhead of muxthreads. Bound threads, though, enjoy the most concurrency. Five bound threads on a five processor system could be running physically concurrently, one thread to a processor, while five muxthreads on the same system might end up running on a single processor.

Concurrency

The idea of concurrency is easiest to understand in the multi-processor model. LWPs are the only scheduling entity in UW2.0. In a multi-processing machine an LWP can be farmed out to another processor. Two LWPs, or two threads bound to different LWPs, running on different processors at the same time are running concurrently - true concurrency. As they run on the same LWP, two multiplexed threads in the same LWP can never run on separate physical processors, and can thus never be truly concurrent.

Thus you can see the two extremes of concurrency: the maximum being 1 LWP per thread and the minimum being 1 LWP for all the threads. In reality, the threads library will not let you pile a large number of threads onto a single LWP. UW2.0 allows you to set the concurrency level through the `thr_setconcurrency` call. Listing 1 is a program that creates 6 additional multiplexed threads, each of which only prints out its process id, LWP id and thread id. Figure 1 shows the output from a run of the program with concurrency set to 1 - minimum concurrency. Notice that even at that setting, the threads lib created 2 new LWPs (2 and 4) to run our spawned threads (2-7), proof that the concurrency level we set in `thr_setconcurrency` is a hint, not an order. Figure 2 shows the output when we increase the concurrency level by 1. A new LWP (5) appears. Notice also that from one iteration of the thread's main loop to the next, the thread can run on different LWPs.

Another anomaly that leaps out from running listing 1 is that the first run creates three LWPs, the one running thread 1 (LWP1) and the ones running threads 2-7 (LWP2 and 4). Logically, somewhere there must have been an LWP3. What happened in this case is that the thread libs wrapper for the sleep function creates its own bound thread and thus a new LWP. All of which is just to say that you really don't have absolute control over the number of LWPs in a process.

Down in the details, scheduling and concurrency is even more complicated than this, but the bottom line is that two bound threads have the maximum 'chance' of achieving true concurrency, while two multiplexed threads with concurrency level set to 1 have the minimum chance.

What's in a thread?

Now that we've laid the groundwork, it's time to use the thing. The first decision to make in threading an application is what lumps of code should get their own thread. Table 2 shows simple categories of code granularity for threading. What you need to mark for possible threading are the medium and coarse grain functions. A good example of a medium grain function is a signal handler. Typically, a signal handler is a single function that does all its work within that function, or with calls to one or two other small functions. A good example of a coarse grain function would be the serial I/O handler of a communications package. While it contains a huge amount of functionality, and correspondingly huge amounts of code, it needs user input to make a complete program.

You also have to decide whether making two threads of execution concurrent yields any real-time gain to the user. If you have three functions, A,B and C, where B can't start until A is done and C can't start until B is done, then making A and B concurrent gets you nothing. If, however, B can start without A being done, then putting B in a separate thread could be a real-time win.

Creation

Creating a thread is as simple as making a call to `thr_create` with the address of the function that will be the 'main' for that thread. By creating the new thread in a suspended state (`THR_SUSPENDED`) we enjoy the luxury of being able to specify exactly when the new thread begins to run. By calling `thr_continue`, the new thread begins processing at the first line of the function passed to `thr_create`. We can call `thr_suspend` at any time to pause our new thread.

One thing to be careful of with `thr_create` is that you can no longer rely on your stack to auto-grow. The kernel supports auto-growth of stack when you run out of stack-space, but since the kernel isn't handling threads, it doesn't know anything about the threads stack. Thus, you really do have to allocate yourself a big enough stack right from the `thr_create` call.

Threads and Signals

You can set up separate signal masks for each thread in a process. A signal sent to a Unix process from another Unix process via `kill(process_id, signal_id)`, however, will only go to one of the threads that are enabled to catch that signal.

If more than one thread is accepting a particular signal, the signal may be delivered to any of the accepting threads.

For this reason, among others, Novell recommends that instead of dealing with signals on a thread-by-thread basis, applications mask all signals in all threads and dedicate a single thread to wait on incoming signals via `sigwait`. Listing 2 adds a signal handler thread to listing 1. As always, it pays to build an appropriately limited signal set. Two new signals have been defined in UW2.0 to support the threads lib, `SIGWAITING` and `SIGLWP`. `SIGWAITING` happens when all LWPs in the processes LWP pool are blocked interruptibly. In `thread8` this occurs when thread 1 is in `gets()`, thread 2 is sitting in a `sigwait()` and all the other threads are either suspended or sleeping. If you add `SIGWAITING` to listing 2's signal set, the process will stop accepting user input and zombie.

Shared Data

In order to share data among heavy-weight processes you have to use the System V shared-memory IPC. Threads, on the other hand, automatically share all global and static data. You can see this in the trivial example of listing 1 where the variable `ulIterations` is a static in the thread start function. Each thread increments `ulIterations` each time through the loop and you get output like that in Figures 1 and 2.

If we had made `ulIteration` an automatic variable, it would have gone on the stack, which is separate for each thread, and thus each thread would get its own, private copy, giving you output like this:

```
Thread1 Iteration 1
Thread2 Iteration 1
Thread1 Iteration 2
Thread1 Iteration 3
Thread2 Iteration 2
```

Interthread Coordination

There are a number of new mechanisms for coordination the activity of threads within a single HWP: locks, semaphores, barriers and conditions. While a detailed discussion of all these would take far too long, a few are so important that they need to be mentioned here.

Mutex Locks

Mutual exclusion locks are used to restrict resource access to a single thread. Lock the resource by calling `mutex_lock`. Any other thread calling `mutex_lock` for that mutex blocks until you call `mutex_unlock`. All the mutex calls take a pointer to a `mutex_t` structure as their first arg in order to identify the mutex. Under the

rules of shared data this `mutex_t` struct must be either global or static in order to be available to all threads.

Reader-Writer Locks

Reader-writer locks are a variation of mutex locks. They allow the application to place two different types of lock on the same resource. When performing a non-destructive operation on the resource (read) the app calls `rw_rdlock` to put a read lock on. Any number of threads can put read locks on a resource (with an important exception). If a thread attempts a write lock on the resource, it will block until the readers unlock. When a thread acquires a write lock, all other readers and writers block until the single writer unlocks. In file-system terms, putting a read-lock on a resource is the equivalent of doing a `chmod 444` on a file (everyone can read, none can write), while putting a write lock is more like a `chmod 600` (one can read/write, no others can read or write).

Conditions

Conditions provide a way for threads to wait on specific conditions without having to 'acquire' a semaphore or a mutex. The following pseudo-code snippet demonstrates:

```
cond_t MyCondition;          // All threads agree that this global condition indicates
                             // that
                             // a line has arrived from the user.
mutex_t MyConditionsMutex;  // All threads agree that this mutex is
                             // associated
                             // with MyCondition.

// this is thread0
BOOL bLineIn = FALSE;
cond_init( &MyCondition ...)
// spawn thread1
gets();
bLineIn = TRUE;
mutex_lock( &MyConditionsMutex );
cond_signal( &MyCondition );
mutex_unlock( &MyCondition );

// this is thread1
mutex_lock( &MyConditionsMutex )
do {
    iRet = cond_wait( &MyCondition, &MyConditionsMutex );
} while ( bLineIn == FALSE );
mutex_unlock( &MyConditionsMutex );
```

From this code snippet it appears that this is a spinning condition, but it really isn't. `cond_wait` blocks until some thread validates the condition (sets `bLineIn`

TRUE) and calls `cond_signal` or `cond_broadcast`. We only sit in this loop re-testing the condition because the condition could have been invalidated again by another thread that was also blocked on this condition and got scheduled before us.

Thread Termination

Terminating a thread is very similar to terminating a Unix process. From inside the thread you call `thr_exit` (which is called implicitly if the start function returns). From outside the thread you have to send the thread a `SIGTERM` signal. If you want to do any cleanup, you can catch the signal, then call `thr_exit`. Suspended threads do not terminate until they are restarted.

A process terminates when all non-daemon threads have terminated. A call to `exit()` or a return from `main()` (which implies `exit`) forces termination of all threads. In listing 3, we precede the call `return(0)` at the end of `main()` with a call to `thr_exit()`. If we run listing3 and start up the threads, they'll start printing their output. While they're running, we hit `q` to exit the main user-input loop. The spawned threads keep running, but `thread0` exits (the `return(0)` never gets executed). Run `thread9` again with the `-d` flag so that all threads are daemon threads and you see that the process (and all daemon threads) terminates when all non-daemon threads (i.e. `thread 0`) terminate.

If, as Novell suggests, you create a separate signal handling thread, either make it a daemon thread, or make sure you have some way of killing it, so that your process doesn't hang waiting for that endless thread to die.

Threads and Libraries

Those of us who suffered through the combination of OS/2 1.0 and MSC5.1 know all about the misery of non-reentrant libraries in a multithreaded environment - traps, mysterious hangs, crazy values. According to Novell, and I've found no reason to doubt them, all the libs delivered with 2.0 and the new SDK are thread-safe. Third-party libs are another story altogether. As usual, there's only one way to know for sure ...

File I/O

The feature of sharing open file descriptors introduces an atomicity problem that is almost certain to blow up any pre-SVR4.2MP third party lib that does file IO. Consider the case where you have two threads, X and Y which share an open file descriptor. X wants to do a seek, read on that file. Simple enough. Problem is, `seek()` and `read()` are separate instructions. Between those two calls, X could be preempted. During that preemption, Y could also call `seek` against that file descriptor, putting the descriptors internal pointer someplace other than where

X wanted it. When X regains control, it will read at the offset Y sought to, not the one X wanted. You could get around this in a number of ways, by locking the file, or surrounding all file ops with a semaphore but those are pretty big hammers to use on such a small problem.

To deal with this UW2.0 introduces `pread()` and `pwrite()`. These are atomic combinations of `lseek/read` and `lseek/write`. The calls are identical to `read` and `write` except that they take an extra argument - the offset from beginning of file to seek to. These calls do not, as `lseek` would, change the file descriptors internal file pointer.

Other Considerations

Now that you've been freed from that ugly old `fork/exec` thing, the temptation is to go out and write a new thread for everything (16 million threads! One for every color in the rainbow!) but you should check that impulse just a little. There is a kernel enforced limit on the number of LWPs that one user-id can have. This is a kernel-tuneable called `MAXULWP`. It has a range of 1 to 65000 and defaults to 200, which should suffice for all but the most esoteric programs (that I can think of right now).

In the listing 1 there is a kludgy, highly unsupported method for obtaining `MAXULWP`. According to Novell, there is no supported way for a non-root user to obtain `MAXULWP`.

The Bottom Line

UW2.0 threads are very easy to get running, and once you get used to it, a much more natural way of viewing problems than the old sequential model. Keeping in mind a few of the concepts and caveats mentioned here should put you well on your way to writing the maximum multithreading program.

END TEXT - BEGIN TABLES AND FIGURES

Thread-specific calls

`thr_create`
`thr_exit`
`thr_join`
`thr_kill`
`thr_setprio`
`thr_sigsetmask`
`pread`
`pwrite`

Process-specific analogue

`fork/exec`
`exit`
`wait`
`kill`
`nice`
`sigsetmask (BSD)`
`lseek/read`
`lseek/write`

getpid thr_self
 Table 1: some thread-specific calls and their process-specific analogues

Code granularity levels	code item	
fine	loop	may be threaded by parallelizing compiler thread
medium	standard one-page function	
coarse	background serial IO communications handler	thread
super-coarse/gross	program	separate heavy-weight process

Table 2: Code granularity

```
P1688 LWP2 - Thread 2 iteration 0
P1688 LWP2 - Thread 3 iteration 1
P1688 LWP2 - Thread 4 iteration 2
P1688 LWP4 - Thread 5 iteration 3
P1688 LWP4 - Thread 6 iteration 4
P1688 LWP4 - Thread 7 iteration 5
P1688 LWP2 - Thread 2 iteration 6
P1688 LWP2 - Thread 3 iteration 7
P1688 LWP2 - Thread 4 iteration 8
P1688 LWP2 - Thread 5 iteration 9
P1688 LWP4 - Thread 6 iteration 10
P1688 LWP4 - Thread 7 iteration 11
```

....

Figure 1: Listing1 output at concurrency level 1

```
P1688 LWP2 - Thread 2 iteration 0
P1688 LWP2 - Thread 3 iteration 1
P1688 LWP2 - Thread 4 iteration 2
P1688 LWP4 - Thread 5 iteration 3
P1688 LWP4 - Thread 6 iteration 4
P1688 LWP4 - Thread 7 iteration 5
P1688 LWP5 - Thread 2 iteration 6
P1688 LWP2 - Thread 3 iteration 7
P1688 LWP2 - Thread 4 iteration 8
P1688 LWP5 - Thread 5 iteration 9
P1688 LWP5 - Thread 6 iteration 10
P1688 LWP4 - Thread 7 iteration 11
```

....

Figure 2: Listing1 output at concurrency level 2