by Bob Japenga (USA)

Concurrency in Embedded Systems (Part 4)

Introducing Linux and Concurrency

This is the fourth article in a multi-part series about concurrency in embedded systems. Here you learn how embedded Linux provides the mechanisms to design robust systems with concurrency.

My previous articles examined common pitfalls in systems with concurrency and discussed in general terms how to deal with these pitfalls. The next several articles will discuss the embedded Linux features available to implement a well-designed system with concurrency. Hardware and software concurrency issues were considered in previous articles. This and upcoming articles will strictly examine software. This article discusses the mechanisms to create concurrency in your software through processes and threads. Upcoming articles will show how semaphores, pipes, mutexes, FIFOs, sockets, shared memory, and message queues can be used with these concurrent threads and processes. Keep in mind; we are taking this in thin slices. To learn more, a great resource is Michael Kerrisk's *The Linux Programming Interface*. I'll introduce the features and you can dig deeper with this and other books.

THREADS & PROCESSES

The first concurrent operating system (OS) I used was Digital Equipment Corporation's RSX-11. (You might be interested to know that the second instantiation of Windows NT is a descendent of RSX-11). In RSX-11, concurrent operations were called "tasks." Later, when I wrote my own real-time multitasking OSes (thank goodness those days are over) we always used the word "tasks" to describe the separately executable programs that concurrently ran. When I first started using Linux (actually QNX was my first exposure to a Unix/Linux-like architecture), I needed to learn a new set of terms: processes and threads. For this article, when I use the word "task" I am talking about either a process or a thread. I'll start by defining terms and looking at the differences and similarities.

DEFINITIONS

Processes and threads fall under my old definition of tasks. A task can be defined as an instance of a software program that is utilizing CPU resources to accomplish some purpose. These resources include memory, I/O, the file system(s), and networking. In Linux, a process is a task that obtains these resources from the kernel. All processes have their own memory allocated to them. These consist of: program memory (sometimes called the "text" or code segment), data memory (where variables are kept), heap (i.e., dynamic memory), and the stack. The kernel's memory manager prevents processes from having any access to other processes' memory. This encapsulation is an extremely valuable feature that enables us to isolate the process from problems created by hardware or software memory corruption. If one process goes amuck, there is no chance of corrupting memory in other processes. If hardware or software never failed, this separation would be unnecessary. Figure 1 shows how a process with multiple threads uses virtual memory.

Here is where things get a little tricky. A process can create a separate process called a "child process." The creating process is called the "parent process." The child process inherits copies of all the parent's data, stack, and heap segments. However, the program memory is shared by the parent and child and is set to read only by the kernel's memory manager. Think of the program memory as the read-only DNA inherited from the parent and data and think of stack and heap as life's experiences. You are stuck (for better or worse) with what you get for DNA, but your life experiences are your own and can be shaped independently of your parent's experience. A child process is free to modify variables and use resources however it wishes, with no possibility of interfering with the parent's memory and resources. With child processes, similar tasks that use a common code base can maintain data independence and still share the same code.

Each process (parent or child) can create separate execution threads. These threads share the same code base as the child processes do with their parent in a read-only memory segment, but they also share all memory except the stack and thankfully "errno" (i.e., the global variable that indicates the type of error that occurs with certain function calls). Thus, one thread of the same process can stomp all over another thread's heap and data variables. Or, to put nicely, they can share each other's data. Initial (i.e., minimum) stack sizes can be separately set for each process and for each thread. Think of threads as old-time multitasking with some useful additions.

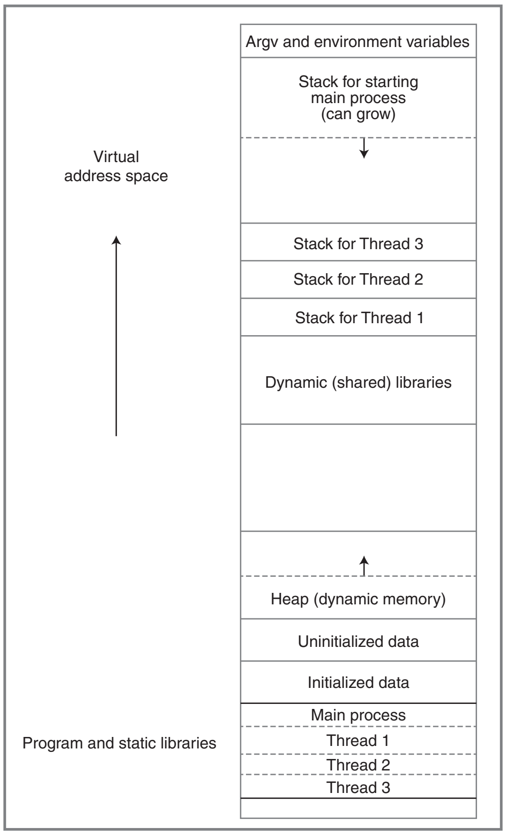

Figure 1—Memory allocation is a multiple-thread process.

SCHEDULING

Any time you use multitasking, you have to know how the kernel performs scheduling. The kernel does not make a distinction between processes and threads with regard to scheduling. In Linux, a single-thread process is treated the same as all other threads. Linux gives the designer significant flexibility by enabling a number of options for the type of scheduling when a process is created. Out of the box, the default scheduling algorithm used for a process is round-robin time slicing (SCHED_OTHER). In round-robin time slicing, every task gets an equal real-time slice. Each task either runs for its allotted time or until it relinquishes control. Although there are priorities, they are more like suggestions to the kernel (i.e., telling children to play "nice"). Appropriately, these are called "nice values." In addition there is round-robin with real priorities (SCHED_RR); first-in, first-out (SCHED_FIFO), which eliminates the time slice (i.e., each task runs until it blocks or terminates); batch mode scheduling (SCHED_BATCH), which tells the kernel this process is "not nice" and gives it less favor when scheduling; and idle mode (SCHED_IDLE), which is the same as SCHED_OTHER, but with a nice value so low, it never has to run.

In addition, Linux now provides the ability to create a group of processes that use what is called "real-time extensions." Each group can be assigned a specific amount of time it can run within a given time period. Any time not allocated to the real-time group will be allocated to the other processes in the standard round-robin fashion. After the kernel is built to support this, the software can dynamically set the scheduling period and a global limit as to how much time real-time scheduling it can use.

Remember, this is in thin slices, there is much more I could say about this. If there is interest, I can certainly write a future article about my company's experience using real-time extensions. A lot of work is being done in Linux in this area—in particular for real-time embedded systems—so you need to be aware of what is supported by your particular version of Linux.

PROCESSES VS. THREADS

For the software system's designer, the question arises as to when to use threads and when to use processes. This choice depends on the differences between the two and the associated advantages and disadvantages.

Data isolation. One cannot speak highly enough about the advantages of eliminating shared data space in creating robust systems. But, the advantages quickly disappear if you design separate processes that require a lot of shared memory. Even with all the isolation processes provide, you can still declare certain memory as a shared resource across processes. If your concurrent task design requires a lot of shared memory, you should use threads. We recently profiled one of our designs that used a lot of shared memory between processes and found that (on an ARM AM3517 running at 600 mHz), each shared variable access cost 50 µs. *Advantage:* Processes, unless you need shared memory.

Context switching times. Any time the kernel switches from one task to another, we call that a context switch. This can happen when a task's allotted time runs out, when the task releases control to the kernel, or when a higher-priority task preempts another. When the kernel's scheduler switches from one thread to another within the same process, the virtual memory space remains the same. During the context switch between processes all virtual memory has to be switched out. An additional cost that is more difficult to measure but can be significant, is that a context switch for processes affects the CPU's caching mechanism. When a context switch happens, the cache's memory addresses are no longer useful. This adds an indeterminate amount of time to the context switch. Published benchmarks on this are sketchy and not helpful. Some give a 10-to-15% advantage to threads. If context-switching time is critical, you may need to rapid prototype your architecture and see what actually happens for your application. *Slight advantage:* Threads.

Resources. Since each process uses its own instance of memory for program and data (child processes are different), a thread can use significantly less virtual memory than a process. In addition, semaphores, timers, and file handles are all shared between threads of the same process. *Advantage:* Threads.

Libraries. Linux enables you to use static or dynamic libraries. A static library is linked into your code space and everything I have said concerning your code space (i.e., "text" segment) applies to any static library. For dynamic or shared libraries, all threads share the same dynamic library in their virtual address space. Dynamic libraries add an additional requirement on the designer concerning a library function's "thread safety." With processes, you don't need to be concerned about whether or not the library is "thread safe" since you have a completely separate instantiation of the library. *Resource-wise advantage:* Threads. *Ease of use advantage:* Processes.

Dynamic starting and stopping. Many times, a design has a requirement to start and stop a task. The overhead for stopping and starting a process is significantly greater (a factor of 10) than starting and stopping a thread. *Advantage:* Threads.

What about profiling tools? There are tools available for profiling both threads and processes, so I see no clear advantage of either. From the command line, processes are a little easier to profile than threads. *Slight advantage:* Processes.

Software updates. Since processes are separate executables that can be separately updated, there is some advantage to making every task a process. *Slight advantage:* Processes.

What about multiprocessing? I expect sometime in the near future to design our first embedded system with a multicore processor. But, to date, it is only a feature we appreciate on our desktops and servers. One huge advantage of using Linux for embedded systems is that the Linux community is getting the kinks out of this OS feature on desktops and servers. When I am ready to incorporate it into my first embedded design, it will have had years of experience. The OS can run any thread on any core (of course, under careful guidance). *No advantage.*

QUESTIONS & ANSWERS

Since I am not usually afraid to go where angels fear to tread, I will give my opinion concerning processes and threads in embedded real-time systems. I think you need to start with some questions.

How important is keeping the data isolated? There are some designs where this is absolutely critical. Processes are the way to go, in that case. End of discussion. I have experienced few problems in systems caused by errant pointers that need the kind of memory protection processes provide. The data sharing threads permit, when rightly designed, is more efficient in the kind of systems we design.

What do you do when a thread crashes? Unlike a server or even a desktop use of Linux, our embedded systems usually cannot tolerate any task failing. On a desktop, if your browser locks up, you just restart the browser. What you don't want is for the browser failure to bring down the whole system. In an embedded system, if one task were to crash, should the rest of the system keep working? Would you want to design the system to restart just the failed process or thread? My opinion, which we have implemented wherever possible, is to force a restart of the entire system in the case of a single thread failure. Recently, some code we wrote got passed off to a customer who modified it in an attempt to restart a failed thread. This was done in the watchdog logic. We immediately jumped all over the unsuspecting maintainer of this code (nicely, of course) about the risks of doing this. In our case, all processes were talking over queues that did not recover well from a restarted process. But more importantly, when something unexpected happens that causes a process or thread to crash, why risk restarting the processes when the system is in an unknown state.

Which to choose? For these reasons, I find that a design with a few processes and many threads is better than a design with many single-threaded processes. We use a sophisticated watchdog mechanism to force a complete reboot rather than restarting an individual process. I use processes when development needs (e.g., software updates or number of project designers) make them easier to use.

DESIGNING CONCURRENT SYSTEMS

Embedded Linux provides the system designer with an arsenal of weapons for designing a system with concurrency. The very mature kernel and tasking model gives most designers more than they need. Next month, I'll examine additional tools to make your tasks "play nice."

Bob Japenga has been designing embedded systems since 1973. In 1988, along with his best friend, he started MicroTools, which specializes in creating a variety of real-time embedded systems. With a combined embedded systems experience base of more than 200 years, they love to tackle impossible problems together. Bob has been awarded 11 patents in many areas of embedded systems and motion control. You can reach him at rjapenga@microtoolsinc.com.

RESOURCE

M. Kerisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*, No Starch Press, 2010.