WRITING ASSIGNMENT 1

CS 444 Spring 2017

Author:
Brandon Dring

Professor:
Dr. Kevin MCGRATH

May 1st, 2017

Abstract

We are going to be looking at the three operating systems that dominate the market, we have Windows, FreeBSD, and Linux. Specifically we are going to analyzing how each OS implements processes, threads, and scheduling. How it compares and contrasts to Linux, and why it might function differently than it does in Linux.

I. Windows

A. Processes

Each process is represented by a EPROCESS (Executive Process) block, much like a struct, (although technically it is a process object)[**1**]. The block has values that point to attributes about it such as process ID, flags, amongst various other data structures with task related values. The EPROCESS block lives in the system address space, apart from the Process address block, which is unique to the process containing it, that is stored in the process address space. An important attribute in the EPROCESS block is the KPROCESS (Kernel Process) field, which contains information on how the kernel schedules the threads of a respective process.

Creating a windows process is very ham handed and contains a total of 7 steps, along with traversing through 3 parts of the Windows OS (Windows client-side library, Executive, and subsystem process). Upon creation, it parses any flags and attributes passed in. Then it opens the image file to be executed, creates a new EPROCESS block, and instantiates the initial thread. From there, it performs post initializations of the new process and threads and loads any DLL's that need to be required as well which are yet a couple more steps[**1**].

Creating a process in windows

```
CreateProcess( NULL, // No module name (use command line)
   argv[1],     // Command line
   NULL,        // Process handle not inheritable
   NULL,        // Thread handle not inheritable
   FALSE,       // Set handle inheritance to FALSE
   0,           // No creation flags
   NULL,        // Use parent's environment block
   NULL,        // Use parent's starting directory
   &si,         // Pointer to STARTUPINFO structure
   &pi )
```

vs. Creating a process in FreeBSD or Linux

```
spawnPid = fork();
execvp(args[0],args);
```

B. Threads

Each process is required to have at least one thread running on it. Much like the processes have the EPROCESS block, threads have the ETHREAD block. The ETHREAD block has an attribute similar to the KPROCESS block for processes called KTHREAD; it contains information for the kernel on how to schedule the threads and synchronize them[**1**]. Alike to the EPROCESS block they both live in the system address space. Inside the ETHREAD block there is also a pointer to the parent EPROCESS block so that it can access pointer to any pending I/O requests that might need to be serviced. Overall the structure of the ETHREAD block is very similar to the EPROCESS block.

Creating a thread is a little more lightweight than creating a process, as the need to create an entire new `EPROCESS` block is gone. Creating a process only requires the call `CreateThread()` function call

C. Scheduling

Windows has a preemptive priority based scheduler, of which there are 32 priority levels and the thread/process that has the highest priority level gets ran first. However, Windows also uses processor affinity for particular threads, so if that processor that has been paired with the process is not available for the ready thread, it might get skipped within the run queue. Digging deeper, Windows also implements a round robin like scheduler that deals quantums, or small units of time. Each running process/thread is given an amount of time varying on system settings, other process states, and a direct manipulation of the process quantum before being switched off and allowing another thread with at least the same priority to run[**1**]. Although, since windows is a preemptive scheduler, if it finds another process that is ready to run with a higher priority, the currently running process might not be able to finish it's quantum do to the scheduler being preemptive, and get placed back in the run queue[**1**]. Generally, when threads are created they have the default priority of 8, but there are certain system based threads like security or session based activities that have a slightly elevated priority level. Having a default higher priority level ensures that OS/Kernel related activities will be ahead of any user threads.

II. FreeBSD

A. Processes

FreeBSD processes are similar to Linux processes in which they are all related from the same tree root, init. FreeBSD uses a smaller struct called `proc` containing the process attributes and thread pointers in comparison Linux. But unlike Linux, and similar to Windows, threads and processes are clearly separated. Similar to Windows processes, FreeBSD processes have at least one thread accompanied with it and runs its code[**2**]. And the process contains the address space available to store data, and kernel resources that it can access and make calls to, such as pipes/sockets, files and the like. Furthermore, the process also has a kernel state that it keeps track of, which can either be process structure or thread structure. The difference between the two being that the process structure keeps tabs on what needs to remain in memory at all times, and the thread structure only keeps track of what it needs while executing. FreeBSD processes have 3 different states they can be in, NEW, NORMAL, and ZOMBIE[**2**]. The new states represents that the process is in the course of initialization and allocation, while the zombie means that the process has died and it and it's resources are waiting to be cleaned by the parent process. While in the normal state, a thread of a process can be in one of three states, which is: Runnable, sleeping, or stopped. Runnable indicates that it is or is being ready to execute, sleeping is waiting on an event typically like I/O, and stopped has been stopped by a signal or parent process.

B. Threads

FreeBSD threads are separate from their processes. Each thread has it's own kernel stack that is up to 2 pages long, but really only one page is used, and the last page is used as a buffer as to not overflow it's stack and overwriting others data structures. A new thread's kernel stack is only the top call frame is copied from the parent, speeding up the copy process from parent to child. [**2**]

An interesting attribute of FreeBSD threads is that they have two modes they can operate in, either user mode or kernel mode. Typically in use, they are running in user mode which gives the thread restricted access to the system. When the thread enters kernel mode, it actually sets a bit in the CPU architecture, then continues on with its duties. Typically, when a thread enters kernel mode it is to make a system call, and after it is done it exits. [**2**]

C. Scheduler

FreeBSD uses a round robin-esque scheduler where it allocates a slice of time for a thread to run on the CPU based on the current priority levels of the threads that are waiting to be ran. FreeBSD has a total of 256 priority levels[**2**], where 0-47 are interrupt threads that have top priority, it's 48-79 which are real time threads that must be completed by a specific deadline. Then 79-120 are threads regarding the kernel but do not deal with interrupts. Past that is general purpose user threads, and idle threads (120-223 and 223-255 respectively). What is interesting about the FreeBSD scheduler is that it doesn't look for complete fairness when assigning its priority. Instead, it looks to see what threads have the most resources, and have requested the most amount of CPU time, like interactive programs with I/O and raises their priority number to service I/O requests as soon as they come back. While CPU hogging threads have their priority number lowered, so if a thread uses all of its' allotted quantum/time slice it's priority is bumped down, as to not slow the system. [**2**] And sleeping threads, or long inactive threads have their priority gradually raised, as to service them as soon as they have been signaled to return.

III. Linux

A. Processes

Linux implements processes as a special kind of task with its own `task_struct`. Each task struct contains a set of pointers to attributes about the process and any of its threads, along with pointers to other `task_struct` within the system (in a doubly linked list). Inside the `task_struct`, there contains two different stacks to each respective process. There is the user stack, and the kernel stack. It is important to keep these separated because if not, user code could alter kernel related data, and thus cause a major security flaw. [**4**] Processes in Linux are cloned from a single very incestious tree. Everything comes from process PID #1 which is init, which is loaded during the booting process. Processes are created with the fork/clone system call, at that point, all process related data is then copied from the parent to the child. So any file pointers, address spaces, and the like, the child will receive from the parent.

B. Threads

Linux threads are unique in that they are treated like processes, and they share resources (Virtual address space, file descriptors, etc), the scheduler treats them no different that a true process.

C. Scheduler

Linux has had a run at various scheduling algorithms throughout its history. Currently since Linux 2.6, they have landed on the Completely Fair Scheduler (CFS). In short, all threads and processes have a relatively equal amount of time for CPU access/resources. When a thread has been neglected it gets a chance to run on the CPU in a fair manner to be put back in balance. [**4**] To determine who gets what priority the Linux scheduler looks at the total time a thread has been able to run on the CPU with `virtual_runtime`. So the lower the runtime, the higher the priority, and vice versa. One interesting thing about the queue that tasks are put in, is that tasks are put in a binary tree, more specifically a red-black binary tree sorted by least CPU time on the left, and the highest CPU time on the right. So, the tree is properly balanced at all times, and any operations on the tree, like an insertion or deletion happen in O(log n) time. [**4**]

There is another unique and very helpful feature in the Linux scheduler such that tasks can be grouped together and treated fairly within that group. For example, a web server might spin off a bunch of threads to handle incoming connections. All the new threads are put into a group where they share virtual runtime. [**4**]

IV. Comparison to Linux

A. Windows

1) Processes: Windows processes are very heavy weight compared to how relatively small Linux processes are. Windows processes can be either a normal process or a protected process with special rights. To create them is a rather extraneous process. For Linux, it is just about 2 calls just `fork()`,`clone()` and `exec()`, but with Windows creating a process behind the scenes is about a 7 step process[**1**].

2) Threads: Threads are distinct in Windows and seen as different entities from their parent processes. While in Linux, threads share the same address space as their parent process and are scheduled the exact same as processes.

Creating a thread in Windows

```
CreateThread(
NULL,              // default security attributes
0,                 // use default stack size
MyThreadFunction, // thread function name
pData,      // argument to thread function
0,                 // use default creation flags
&dwThreadId); // returns the thread identifier
```

vs. Creating a thread in FreeBSD or Linux

```
pthread_t Mythread;
 pthread_create(&Mythread,NULL,fnc,args);
```

3) Scheduling: Windows scheduling is uniquely different because of the processor affinity that a task can have, therefore altering the potential queue that Windows uses. Windows also is different in the way the prioritize the threads. Windows based off of 32 different levels, and Linux with a two factor niceness value, and soft real-time priority.[**1**][**4**]

B. FreeBSD

1) Processes: FreeBSD processes are overall very similar to Linux, processes are all instantiated from a single root with PID #1, and all processes and threads are kept in a struct. However in FreeBSD they are kept in a `proc` struct, as opposed to a `task_struct`. Which is slightly smaller. They are all connected as well by a linked list, from a pointer within the `proc` struct itself.

2) Threads: Similar to Windows, threads are separate from processes. Linux, is unique in the sense that threads and processes are seen as the same.

3) Scheduling: FreeBSD and the Linux scheduler are similar but have a slight, yet important difference between them. Both have a round robin approach in they give essentially loop back in a circle giving all tasks a time

certain amount of CPU time. In Linux, the current scheduler is looking to keep an equal amount of time split between all the processes/thread, no matter what kind of task they are doing. But the FreeBSD scheduler is favored to more I/O bounded and interactive programs. Along with different priority levels between Linux and FreeBSD.

CITATIONS

1.) M. E. Russinovich, D. A. Solomon, and A. Ionescu, Windows Internals Covering Windows Server 2008 and Windows Vista, 5th ed.

2.) M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, The Design and Implementation of the FreeBSD operating system, 2nd ed

3.)M. Jones, “Inside the Linux 2.6 Completely Fair Scheduler,” IBM - United States, 15-Dec-2009. [Online]. Available: https://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/. [Accessed: 02-May-2017].

4.)M. Mitchell, J. Oldham, and A. Samuel, Advanced Linux Programming, 1st ed. Indianapolis , IN: New Riders Publishihng, 2001