Processes, Threads, and CPU Scheduling

Nehemiah Edwards

CS 444

Spring 2017

I. Processes

A Process is defined as a container for a set resources used when executing an instance of a program, where a program is a sequence of instructions [5].Every process has an Unique Process ID, or PID associated with it that unique identifies each process. Some aspects of a process are the same for all Operating Systems. Things like the ability to communicate to another process through a shared object or pipe, ability to create child processes, threads, etc [5]. Although there are some similarities, processes running in Windows differ greatly from those that run in Linux or FreeBSD. Processes in Linux are similar to processes in FreeBSD

A. Processes in Windows

In Windows, active running processes can be seen in the Task Manager. Each process has at least one thread, and may have multiple which is known as multi-threading. Additionally processes have a private virtual address the process is executed and later mapped into a physical memory location. They will also have a list of open handles to the system's resources, which includes things such as semaphores, files, and communication ports [5]. Access tokens are used to hold information to prevent users from executing a process without the correct permission to do so. A process in Windows is created like this:

```
CreateProcess( NULL,
        argv[1],
        NULL,
        NULL,
        FALSE,
        0,
        NULL,
        NULL,
        &si,
        &pi )
```

A child process will only store the parent's process id, or PPID, no other information about the child's parent will be stored in the child process unlike in Linux. When a parent process closes any child process associated with it will continue what it was doing and they are considered to be zombies at that point.

B. Processes in FreeBSD

In order to see active running processes in FreeBSD, the ps(1) or top(1) commands are used, which is similar to the way you would do in Linux. A new child process is created with the fork() command which is exactly the same as Linux

C. Processes in Linux

In Linux, active running processes can be seen in either the System Monitor or by running the ps command in the console [5]. Unlike in Windows, If a parent process is closed in Linux, then all child processes associated with the parent must exit as well [5]. A new child process is created by using fork() like this:

```
pid_t child_pid = fork();
```

Here, child_pid will show if the process is the parent if it is not eqault to zero, otherwise it is the parent process. Initially everything from the parent is shared by the child, until the child makes changes to the stack or the heap [1]. An exec command allows the child to be replaced by another program while the parent is continuing the execution of the original program, so fork() and exec() are often used together

II. Threads

A thread is a basic unit of CPU utilization, made up of a stack, a program counter, and a set of registers. A Single-threaded process has only one thread, thus only one sequence of instructions may be executed at any given time. Making use of multiple threads(Multi-threading) is useful when a process has multiple independent tasks associated with them. This is especially useful if a thread is blocking, so that the other threads can continue without having to block. Threads in Windows and threads in Linux and FreeBSD are handled very differently. FreeBSD and Linux are very similar.

A. Threads in Windows

Threads are bulit-in to the Windows Operating System. In Windows threads, each object has only essentially one type, which is HANDLE [2]. To create a new thread, only one funtion call is needed, which is CreateThread and can be used in a manner that's similar to this:

```
threadArray[i] = CreateThread(
            NULL,
            0,
            threadFunction,
            threadArg[i],
            0,
            &threadId[i]);
```

Only one function is needed to make a thread block while it's waiting for an object, which is WaitForSingleObject or to have it for multiple objects WaitForMultipleObjects is used [2]. For Example the code below would wait for the child process to exit:

```
WaitForSingleObject( pi.hProcess, INFINITE );
```

WaitForMultipleObjects allows a programmer to have it wait for any thread out a set to be terminated and efficiently perform "clean up" processing. This can also be set to wait for any combination of threads,events, semaphores, and/or mutexes, in order to give the programmer some flexibility which is quite difficult to acomplish if using Pthreads if at all [2]. When a signal is received in a windows thread it will stay in a signaled state, making it unlikely to be lost and easier to deal with [2].

B. Threads in FreeBSD

Similar to Windows, FreeBSD supports transparent multiprogramming, to give the illusion of concurrency in the execution of multiple processes and programs [4]. This is done through context switching, or switching contexts of threads either

within the same thread or a different one. FreeBSD is able to use POSIX Pthreads like Linux uses(Explained in the Threads in Linux section), so the way the processes are handled are nearly identical.

C. Threads in Linux

Linux uses POSIX Pthreads libary to implement threads. They aren't bulit in like in Windows. In Linux's Pthreads, each object has its own type such as pthread_t, pthread_mutex, pthread_cond_t, etc. [2]. They are created like this:

```
int pthread_create(pthread_t *restrict tidp, const pthread_attr_t *restrict attr,
                   void *(*start_rtn)(void), void *restrict arg)
```

This means that different functions are required for each object type and a programmer must know the number order and type of parameters for many different functions. In order to wait for muliple objects to terminate, each termination has to be done one funtion call at a time using pthread_join [2]. To handle signals in Pthreads a while loop must be set up to test a conditional expression and a mutex needs to be used in order to protect it data and any changes to it. If there isn't a thread waiting on a condition variable and a signal is recieved, then the signal is lost [2]. The Linux kernel there is no such thing as a thread, as it instead views the threads as separate running processes [1].

III. CPU Scheduling

CPU scheduling allows one process to use the CPU while the execution of other process is waiting for some resource to become , in order to make good use of the CPU. The goal is to make the system efficient, fast, and fair to each process. There are many different scheduling algorithms that deal with various different needs. Each operating system utilize a different algorithm all of which have some form of priority handling.

A. CPU Scheduling in Windows

Windows uses a multi-level feedback queue algorithm. The default fixed quantum in Windows Server is 120ms [3]. This ensures less overhead when context switching and provides fairness as all processes are given the same length of time. I/O bound processes are given higher priority over CPU bound ones so that the longer quantum length doesn't affect interations too much [3].

B. CPU Scheduling in FreeBSD

FreeBSD's default scheduler, referred to as a timeshare scheduler, the priority that a process has is recalculated over time by various parameters [4]. Some task require a real-time scheduler in order to finish by a certain time, or in a certain order. FreeBSD's kernel implements real-time schduling in a queue that remains seperate from the one that is used for regular time shared processes [4]. This is similar to Windows in that it gives priority to interactive programs.

C. CPU Scheduling in Linux

Unlike Windows and FreeBSD, Linux has several different CPU schedulers and I/O elevators for a variety of different usages [3]. This allows the programmer to adapt the scheduler based on what is implemented. The default scheduling type is set to round-robin time sharing, ensuring fairly distributed quantums.

REFERENCES

[1] Himanshu Arora. What are linux processes, threads, light weight processes, and process state, Nov 2013.

[2] Clay B. Why windows threads are better than posix threads, Aug 2016.

[3] Dimitris. Processor scheduling and quanta in windows (and a bit about unix/linux), Aug 2007.

[4] Marshal Kirk McKusick, George V. Neville-Neil, and Robert N.M. Watson. 4.1 introduction to process management — informit, Oct 2014.

[5] Shankar Raman. Process behavior in windows and linux, Sep 2014.