Final

Sam Quinn

December 11, 2014

1 CPU Scheduling

Because operating systems run multiple processes concurrently the kernel needs to ensure each process is scheduled for execution. While many processes can run together not every process can run at once. This is where the CPU Scheduler comes into play. To understand how the kernel schedules these processes it is helpful to know the different types of "Multitasking" *cooperative Multitasking* and *preemptive multitasking.* Most modern operating systems implement a preemptive algorithm for scheduling which means an external entity dictates what process should be scheduled next. I will not be covering *cooperative Multitasking* in which the processes schedule themselves.

1.1 Scheduling Algorithms

As you can imagine allowing each process to execute can be a hard task. Even with modern systems the kernel has to be wary of how it allocates its resources. One of the main concepts behind how the scheduling algorithms work is that when a process does not have any work to be done that process can go into a *block* or *sleep* state to allow for other process that have work to be done to execute.

Linux

In earlier versions of Linux the kernel implemented an *O(1)* algorithm that as the name suggests, ran at constant time. This was said to be a great algorithm for larger systems but failed to perform well on I/O heavy devices. During the rise of Linux personal computers the *CFS* (Completely fair scheduler) was created to handle blocking I/O operations better and is the default is today [1].

The Linux scheduler, like many other Linux features, can be customized by selecting different algorithms. Linux allows changing of the CPU scheduling algorithm either system wide or per process basis. The is done with the implementation of a *Scheduler Classes. Scheduler Classes* allows different classes of processes to utilize different algorithms with different policies concurrently [1].

The way the processes are scheduled by the selected algorithm is based on their priority. In Linux there are two types of priorities we need to worry about *real-time* priority and *nice* priority. The *nice* value is supposed to essentially rank the process worthiness for processor time. Linux has sort of a fun way of implementing the *nice* priority system. The scale ranges from [-20 to 19] with the highest priority being -20, and 19 being the lowest or "the most nice" to other processes. The *real-time* priorities are implemented opposing the nice implementation with a scale from [0 to 99] with 99 being the highest priority. In normal configurations *real-time* processes have a greater priority over the system than normal processes [1].

Windows

Windows also schedules processes with a priority base preemptive scheduling algorithm with a few differences. Windows has had the same reliable scheduling algorithm since the first release of Windows NT with scalability improvement with each release thereafter. The Windows scheduler works by trying to equally share the processor or processors only among other processes with the same priority. This causes the problem that the Windows scheduler doesn't work well with different higher-level requirements like distribution of CPU time with multiple users in a terminal environment. The new scheduler named *DFSS* (Distributed Fair Share Scheduling) solved this problem later on. Windows determines what algorithm to use during the *PsBootPhaseComplete* final post-boot stage [2].

Like mentioned before, Windows shares the use of priories with Linux but the values are implemented differently. Windows seems to have a more understandable priority value system than Linux's implementation. Windows separates scale of 32 values into two groups, real-time levels and variable levels. The real-time values occupy the upper 15 levels of the priority [16 to 31]. The variable levels range from [1 to 15] reserving the priority 0 for the system which is one per system. The Windows priority levels go in one uniform direction with the higher the priority value the higher the priority [2].

Context Switches

When a new process has a turn in the queue or if their time slice expired the kernel initial a *context switch*. A *context switch* is the live swapping of one runnable task to another. Change a running process out has two aspects that must be done. First is that they need to map their own virtual memory. Second is reimplement the processes stack information and registers. This context switch takes place every time a new process begins or resumes.

Linux

In the Linux system, context switches are executed by the **context_switch()** system call. Context switches are initiated by the Process scheduling algorithm with the **schedule()** call. The **context_switch()** system call, like mentioned above, needs to remap the virtual memory and copy the stack information. To remap the virtual memory the **context_switch()** system call issues a call for **switch_mn()**, that removes the old processes mappings and replaces them with its own. Copying of the stack information is completed with a different call named **switch_to()** which both saves and replaces the stack information when a context switch occurs [1].

```
context_switch(struct rq *rq, struct task_struct *prev,struct task_struct *next){
   struct mm_struct *mm, *oldmm;

   prepare_task_switch(rq, prev, next);
   my_trace_sched_switch(rq, prev, next);

   mm = next->mm;
   oldmm = prev->active_mm;
...
```

Windows

Windows also needs to perform the same general principle of switching execution to a separate process. The Windows kernel must first save the information of the old thread by creating a stack pointer to the underlaying *KTHREAD* structure. Once the old stack is safe the new process then pushes its own data on the current running process stack. If the new process does not share the same address space as the last running process it must also load the its page table. The new process copies the old address and stores it for latter while loading its new page table into one of the processors registers [2].

Comparison

Similarities

- Have options to change algorithms.
- Each process has a priority
- Both have different priorities for real-time and normal processes.
- Context switches are fundamentally the same.

Differences

- Windows Windows implements only one priority value for both real-time and variable processes.
- Linux's priority of nice values is backwards from Windows.

2 Processes and Threads

To compute efficiently one very important concept to understand is how a programs operates in a parallel environment. There are two ways a program may run in parallel, threads and processes. While both treads and processes accomplish the same task of running in parallel each have their correct application. One of the main difference between these two and should be taken into consideration when choosing weather you should use threads or processes, is that threads share the same address space while each process has its own address space [4].

2.1 Processes

Processes are most useful when parallelization is required but sharing of system resources is nonessential. Processes often times do not share the same address space and are independent from all other process. The creation of a process can be an expensive task since each process that is created has to allocate its own address space and force the CPU to execute a context switch.

Linux

To create a process in a Linux operating system the **fork()** system call is used. When the **fork()** system call is called it creates a child process from the currently executing process which is refereed to as the parent. It is common to initiate a new program after the **fork()** command. To create a new namespace and begin executing a new program in Linux the **exec()** system call is used. If the parent process relies on the child process it can inquire about the child's status by using the **wait()** system call. Child processes when completed exit using the **exit()** system call, however, these processes are not completely destroyed from the system quite yet. When a child process calls **exit()** it is put into a “Zombie” state until the parent calls **wait()** [1].

Underneath the hood in Linux each process has its own “Process Descriptor” which keeps all of the processes information stored at the end of its stack, which is linked to a system wide doubly linked list. When **fork()** is called it initiates the **clone()** system call which does just what you would expect clones the parent process. The child and the parent at this point share the same copy of everything including their process descriptors. At this stage of process creation the **clone()** system call will differentiate the child process from the parent and pass required information if the child would like shared address spaces, open files, and namespaces. These flags are really the only thing that differentiates the creation of a process from that of a Linux thread [1].

```
...
    char *stack;
    char *stacktop;
    pid_t pid;

    // Processes
    pid = clone(childFunc, stackTop, CLONE_NEWUTS | SIGCHLD, argv[1]);
    if (pid == -1)
        errExit("clone");
    printf("clone() returned %ld\n", (long) pid);

    // Threads
    pid = clone(childFunc, staktop, CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND);
    if (pid == -1)
        errExit("clone");
    printf("clone() returned %ld\n", (long) pid);

...
```

Windows

Within a Windows operating system the main principles of threads and processes are preserved with process having their own address spaces and threads sharing the parents address space, however, Microsoft Windows does it slightly different. Windows uses the **CreateProcess()** system call which functions similar to the Linux **fork()** implementation. All of the information needed to execute a program is passed in to the **CreateProcess()** function. Since Windows does not clone the parent process like Linux does the **CreateProcess()** function requires many more parameters to achieve the same effect but potentially has more control of the child process [2].

```
...
// Start the child process.
    if( !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 )         // Pointer to PROCESS_INFORMATION structure
    )
    {
        printf( "CreateProcess failed (%d).\n", GetLastError() );
        return;
    }
...
```

2.2 Threads

Threads function very similar to that of processes in regards to performance but with the added benefit of being able to share the same namespace as the parent. Threads are essentially "light weight" process since they do not require their own address space and namespace it is more memory efficient, however, threads can not have threads of their own.

Linux

The Linux kernel views each thread as a unique process that merely shares resources of another process. Every data structure that is used for a process is also used for a thread within a Linux system. While Microsoft Windows has specific kernel support for threads to allow threads to be simple forms of processes, Linux on the other hand does not implement any thread improvements since Linux processes themselves are already considered "Light Weight". The Linux implementation of a thread is the same as that of a process and are created the same way. Linux creates a thread with the same **clone()** system call as with a process, with the exception of a few flags. The flags passed to the **clone()** system call instruct the kernel to share the address space, filesytem resources, file descriptors, and signal handlers to the newly created thread [1].

Windows

Like mentioned above Windows has specific implementations to create a thread unlike the universal **clone()** system call that Linux uses. To create a thread in Windows the **CreateThread()** is used. Each thread created in Windows shares the thread context of the main thread (parent process). The thread context includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space. Windows threads also acquire the same security context as that of their parent unless specifically setup to receive otherwise [2].

```
// Create the thread to begin execution on its own.

        hThreadArray[i] = CreateThread(
            NULL,                   // default security attributes
            0,                      // use default stack size
            MyThreadFunction,       // thread function name
            pDataArray[i],          // argument to thread function
            0,                      // use default creation flags
            &dwThreadIdArray[i]);   // returns the thread identifier
```

Comparison

Similarities

- Support both Threads, and Processes.
- Processes do not share an address space while threads do.

Differences

- Linux uses the **clone()** call to interact with the kernel directly where as Windows uses the **CreateProcess()** which must go through the Windows API first.
- Windows threads are viewed as threads while Linux threads are just process with shared memory.

3 File Systems

File systems are the fundamental principle behind how data is transfered in I/O situations. Every file store in a computer is stored and retrieved via a file system. Without a file system all of the data on the disk would be a pile of bytes unknowing where one file ends and another begins. Microsoft Windows has a very different way of approaching filesystems than the Linux implementation.

3.1 Filesystem Support

For each filesystem there needs to be a way for the data structures to be written and read from the physical volume. This is where the filesystem drivers come in. Both Linux and Windows have the need for Drivers but the way they are implemented with in the operating systems are very different.

Linux

Linux supports dozens of different filesystems with the most common being Minux, Ext[2,3,4] and JFS. Linux has one of the most versatile filesystem support all because of the *VFS* (Virtual Filesystem). The *VFS* creates the abstraction from real underlaying file I/O operations. With all of the I/O operations flowing through the *VFS* for Linux to support different filesystem all that is needed is a driver to translates standardized *VFS* data operations to the current filesystem operations. This allows the Linux system to perform I/O operations without even needing to know what the actual filesystem of the disk is. [1].

```
...
#include <linux/fs.h>

// Registering a Filesystem
extern int register_filesystem(struct file_system_type *);
// Unregistering a Filesystem
extern int unregister_filesystem(struct file_system *);
...
```

Windows

Windows on the other hand only supports a handful of filesystems. Since Microsoft does not take advantage of a standardize virtual file system like Linux the operating system must know each filesystem and how to specifically talk to the media. Windows achieves this with an advanced *I/O Manager* that in turn uses drivers just like the Linux kernel to perform the actually operations on to the filesystem [2].

Windows either at boot or when a new volume is mounted first tries to identify what type of filesytem the volume contains. Every Widows supported filesystem contains the information need for the *FSD* (Filesystem Driver) in the volumes boot sector. If the filesystem format is unrecognized or boot sector has been corrupt rendering the information Windows uses to identify the volumes filesystem unreadable Windows will default to the *RAW FSD*. When Windows declares a filesystem as *RAW* the user is prompted to see if they would like to format the volume. If the boot sector is intact and Windows identifies the filesystem as a supported type Windows creates a *Device Object* that the operating system will use to map the requested I/O operations to the physical media. After a *FSD* claims a volume all of the I/O operations to that volume is passed trough the *FSD*. All of the I/O operations are mapped from the *Device Object* by the *VPB* (Volume Parameter Block) to the volumes responsible *FSD*. When Windows mounts a volume and assigns a drive letter to it, it is really just a symbolic link to the *Device Object* [3].

Comparison

Similarities

- Read information from the MBR(master Boot Record).
- Each filesystem gets a file descriptor mapped to unique location.
- Need a filesystem driver to write to actual volume.

Differences

- Windows kernel needs to be aware of what the underlaying filesystem is.
- Volumes in Windows get assigned letters, Volumes are assigned devices with numbers in Linux.

References

[1] Robert Love. *Linux Kernel Development, 3rd Edition.* Pearson Education, Inc, Crawfordsville, Indiana, 2010.

[2] ALex Lonescu Mark Russinovich, David A. Solomon. *Windows Internals, Part 1, 6th Edition.* Microsoft Press, Redmond, Washington, 2012.

[3] ALex Lonescu Mark Russinovich, David A. Solomon. *Windows Internals, Part 2, 6th Edition.* Microsoft Press, Redmond, Washington, 2012.

[4] Varron Sahgal. Programmer interview, 2013.