**OS Process Management**

OS460
Montana State University
Dillon Tice
Jakob Frank

## Abstract

In every modern operating system, there exists software to handle the spawning, execution, and termination of processes such that each process is allocated the necessary resources for execution, while sharing and synchronizing these resources with other processes that may need them at the same time. This report focuses on the various implementations of user level process management utilities provided by Windows, Mac, and Linux operating systems and how they compare to one another with regards to performance and usability. Since they are proprietary systems, the documentation on Mac and Windows operating systems are lacking in detailed information about process management, as opposed to Linux which is open source.

## Overview

In the Linux kernel, processes are maintained within a structure known as task_struct which contains many fields used in the representation of processes. The Linux kernel source code is openly available and written in C so those who wish to develop for it can do so using more abstracted wrapper functions like fork() and exit() integrated into the kernel or get very low-level into the actual system calls (syscalls) by the same name. Theoretically, there are no limits to process management in Linux as a normal developer. One could write their own process scheduler, overwriting the Linux kernel's existing scheduler, and build the source code for their own use. Multithreaded processes can be achieved using the library pthreads which allows for multiple user-space threads. Conditional compilation is achieved in sched.h with the use of ifdef and ifndefs which are triggered by architecture-specific implementations of hardware.

Windows manages processes in a way similar to Linux in that there are structs, representing the information associated with a process, and interfaces for interacting with these structs, however there are some key differences. There are a few different libraries and APIs used for creating Windows applications, namely Win32 and the .NET framework, which is built on top of Win32. There are similarities and differences in the ways that both of these APIs handle process creation, interprocess communication, and termination.

The Mac Operating System is built on top of a kernel known as XNU (X is Not Unix) which takes ideas from both monolithic and microkernels and implements them in a way that allows the core of the OS to be run as multiple, separate processes. The kernel takes code from Mach 3.0, an architecture originally designed at Carnegie Mellon University, and is implemented for such tasks as preemptive multitasking, virtual memory, interprocess communication (IPC) and real-time support. There are some drawbacks to the way that XNU was implemented; the XNU team had to balance the tradeoffs between the micro and monolithic kernel solutions in order to optimize both flexibility and performance.

## Windows

The basic block of information regarding a process is called the Process Control Block (PCB), which contains information about a process such as its priority class, external code segments in DLLs, access to the process heap, Thread Local Storage (TLS) that contains pointers to memory, giving each process the ability to allocate storage, and much more. The EnumProcesses function in the Win32 API provides a way to enumerate all the processes, by id, currently running on the system, which then allows the programmer to retrieve more specific information about each process by passing the id to GetProcessInformation(). Similar to Linux, each PCB contains pointers to both the process that called the current process, and any children of the current process.

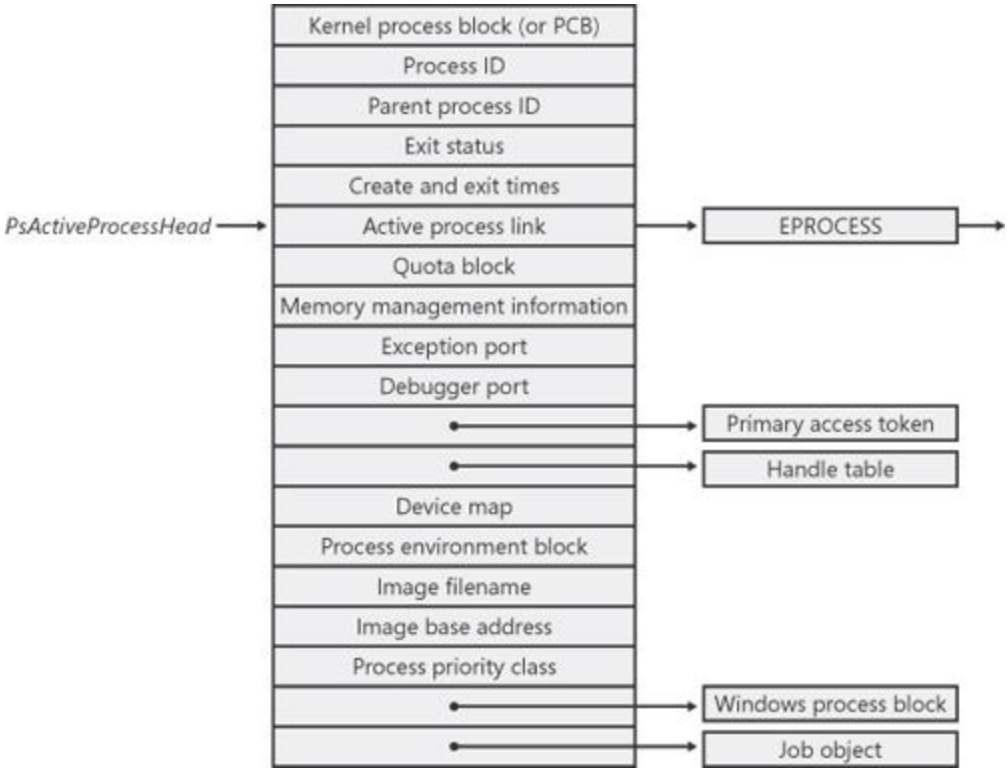


*Figure 1: A chart describing the structure of a Windows PCB*

There are a few different APIs that Windows programmers can use to access functions and structures from the kernel. The Win32 API offers structure to processes through its Processthreadsapi.h header file, which provides a PROCESS_INFORMATION struct; the heart of the Windows PCB. It also offers functions that allow for the creation, modification, and termination of processes that use these structures. Such functions include CreateProcess(), ExitProcess(), and GetCurrentProcess().

The windows.h header file provides access to synchronization objects that allow for the use of mutexes, semaphores, and other synchronization methods. For thread-level synchronization, Slim Reader/Write (SRW) locks are often used when threads need to share the same data. Additionally, there are also Condition Variables and Critical Section functions and objects available, depending on the usage of the threads. For interprocess synchronization, Events are most often implemented to signal to processes whether or not they are able to proceed. An Event object has two states: “on” or “active” to say that waiting threads/processes may proceed and “off” or “not active” to say that waiting threads/processes must continue waiting. Additionally, it provides access to mutex and semaphore functions such as CreateMutex(), OpenMutex(), ReleaseMutex(), and the respective functions for semaphores.

An alternative way of interacting with the Windows kernel is to use the .NET framework, which provides a unified way to access the same functions and structures from different platforms and across multiple languages. By inputting a file path to an .exe file, the API creates a Process object that provides functions such as .Start(), for starting a process, .Kill() for terminating, and .GetProcesses()/.GetProcessByName()/.GetProcessById() for getting relevant process information. In addition, the ProcessPriorityClass allows an enumeration of the priority levels of different processes, as defined by the kernel. Such classes include Above Normal (32768), High (128), Idle (64), and Real Time (256), with Real Time being the absolute highest priority.

The way most users interact with Windows processes is through the Task Manager program. It provides a simple list of information about all currently running processes and allows the user to create new processes (via the 'New Task' button), terminate processes ('End Task'), and assign priorities to them. There isn't much information about the actual inner workings about the Windows Task Manager, but it began as a simple Task List, created by a Windows developer in 1995 as a side project, and evolved into something that is absolutely essential to managing Windows' processes effectively.

## Linux

### *Representation of Processes*

In the Linux kernel (currently 5.4.2), processes and threads are synonymous. A process is represented as a large ( ~1 - 2 KBs) structure known as the task_struct. This structure contains several fields that are necessary to manage processes. Some important ones include state, stack, flags, static_prio, sched_info, tasks, children, sibling, group_leader, thread_info, pid, tgid, and files. States include TASK_RUNNING, TASK_INTERRUPTIBLE, TASK_STOPPED, and more. The flags field defines many indicators. Some include whether a process is allocating memory, being created, or exiting. Each process is given a priority (assigned to static_prio); however, a process's actual priority is calculated dynamically given many different factors. Tasks is a (modified) linked list node that allows all processes in a system to be iterated through via "children" and "sibling" pointers. Thread_info contains architecture-specific data as well as storage for context switches such as hardware registers, program counter, etc. PID contains a unique process identifier. In a multithreaded environment, TGID contains the PID of the process (known as the thread group leader) that spawned additional threads. The init process, which is an ancestor to all other processes, has a task_struct which is static and a PID of 0.

### *Process Creation*

Creation of a process in Linux always derives from the fork() syscall. A very basic example of how this can be used is demonstrated here

Parent Process

```
void main() {                pid=312
      int pid = fork();
      if (pid == 0) {
      child_process();
      } else {
      parent_process();
      }
}
void child_process() {

}
void parent_process() {

}
```

Child Process

```
void main() {                pid=0
      int pid = fork();
      if (pid == 0) {
      child_process();
      } else {
      parent_process();
      }
}
void child_process() {

}
void parent_process() {

}
```

Note that the code for the two programs is exactly the same. Execution will begin for the child process from the line after fork(). However, the pid inside of the child process will be 0 and the pid inside of the parent process will be 312 - allowing different execution paths despite being the same code.

To specify further on how fork() works, fork() is no longer a pure system call since kernel version 2.3.3, it is a glibc wrapper that calls the clone() syscall with flags that provide the same effect. Clone() then calls the _do_fork() syscall which executes copy_process(). Copy_process() takes the clone_flags that are passed to it in fork() and determines if the user has proper permissions to duplicate the process. If copy_process() returns true, the process's task_struct is duplicated and credentials are copied into the new process. Resources and CPU time are reset to zero in the child process and it will not inherit semaphore adjustments, record locks, or timers from the parent process. In cases where resources are limited or processes after forking will immediately execute a new process via execve(), vfork() can be used to duplicate a process without copying the page table of the parent process. It achieves this by suspending the parent process until a call to execve() or exit() is made in the child process.

*Process Termination and Signals*

Process termination is most often done with the exit() syscall. exit() calls do_exit() which does the majority of the work of setting the task_struct's flags (to EXITING), releasing

semaphores, decreasing file count references, and setting the task's state to ZOMBIE. Immediately after, all objects being maintained by the process are freed and the only memory remaining of the process is its kernel stack, thread_info struct, and task_struct (without many empty fields). This is solely to provide information to the parent process. After the parent process has accepted the information or signified disinterest, the remaining memory is freed.

Signals are used within Linux to further manage process execution. When a signal is sent to a process, the operating system halts normal execution of the process and delivers the signal. If the process has been written to handle certain signals, it can do so; otherwise, the default Linux handler will be invoked. The kill -15 command sends the signal SIGTERM to a process as a request to terminate that process. This can be handled by a process and is typically done by closing resources used in a safe, clean manner. The kill -9 command sends the signal SIGKILL to a process, which instructs the operating system to forcibly terminate the process and is one of two signals that cannot be handled by the process itself. Other signals mostly relate to improper usage or access of resources by a process. Some of these include SIGTRAP, SIGPIPE, SIGALRM, and SIGUSR1.

### *Process State*

Processes and their corresponding task_struct state field have five possible states. These include two sleeping states: TASK_INTERRUPTIBLE (INT), TASK_UNINTERRUPTIBLE (UNINT) as well as three other states: TASK_RUNNING, TASK_STOPPED, and TASK_ZOMBIE. INT indicates that a process is blocked and waiting for a certain condition, such as a time slot or event. When that condition is true, the process will switch to the RUNNING state. UNINT is the same as INT, but the process will not respond to signals. This is typically used when a wake up event is expected to occur quickly or a process must wait for a certain period of time without being interrupted. In the ZOMBIE state, the task has terminated but the parent process has not yet chosen to retrieve or discard the child process's information. In the STOPPED state, the process has received one of the four signals SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU. Alternatively, it can occur when a process receives any signal while being debugged. In the RUNNING state, a process is running or in the runqueue waiting to be ran.

### *Context Switching*

When a different process takes control of the processor and a resulting context switch occurs, or system mode changes, the first process needs to be easily accessible. To achieve this, the thread_info (which contains a pointer to the task_struct) for the process is contained at the bottom of the kernel stack. This makes the process easy to retrieve as the bottom of the stack can be computed with two lines of assembly:

```
movl $-8192 %eax
andl %esp %eax
```

The **esp** register points to the current stack pointer. The first line moves the size of the stack ($2^{13}$ ==8192) into the **eax** register. In the second line, we AND the current stack pointer register with these bits to move to the bottom of the stack. This occurs because $2^{13}$ in binary is all ones with 13 zeros at the end.

A process switch is done within Linux with the schedule() function. The macro switch_to() is used to switch to kernel mode and is hardware-dependent. The third parameter of switch_to(), "last", preserves the "prev" descriptor after process switching.

**/kernel/sched/core.c**

```
static __always_inline struct rq *
context_switch(struct rq *rq, struct task_struct *prev,
             struct task_struct *next, struct rq_flags *rf)
{
      prepare_task_switch(rq, prev, next);
      arch_start_context_switch(prev);
      ...
      ...
      switch_to(prev, next, prev); <------------ SWITCH TO
      barrier();

      return finish_task_switch(prev);
}
```

## MacOS

The XNU kernel is a unique kernel that is actually made up of components of 2 different kernels: the Mach 3.0 kernel, often considered the first microkernel, and FreeBSD, a more monolithic kernel. The combination of these two kernels allowed for operating systems to be built in a more modular, process oriented approach offered by microkernels, while maintaining the speed of a monolithic kernel. As such,

The Mach 3.0 kernel was originally developed by Carnegie Mellon University (CMU) for the purposes of operating system research, intended to replace the current Accent kernel that they were using. Mach took an existing BSD kernel and re-implemented it based on the experimental concepts of interprocess message-passing, as defined in Accent. By rewriting the BSD kernel from the ground up, with a more IPC-based approach, the team was able to achieve a large degree of modularity with their kernel, while at the same time maintaining compatibility with UNIX based systems. This approach allows for multiple different operating systems to be running as a set of processes at the same time on top of the Mach layer, easily being distributed over a multiprocessor architecture.

The Mach API for interprocess communication offers a few different interfaces for working with IPC concepts, namely the Task Interface, the Lock Interface, the Semaphore Interface, and the Scheduling Interface. Similar to the interfaces provided by the Win32 API, these interfaces provide the necessary structures and functions to achieve synchronization between processes. The Task Interface provides constructs for creating, suspending, and terminating processes (called ‘Tasks’ by Mach) in addition to listing information such as priority level, process id, and related processes through the use of Task Info. The Lock Interface comes with functions for creating, acquiring, and releasing locks to prevent processes from colliding with one another when working with the same data. Lastly, the Semaphore Interface, intuitively, provides access to functions that facilitate the creating, signalling, and destruction of semaphores. These interfaces are vital to the IPC abilities of Mach and, in turn, the performance of XNU, which is at the heart of MacOS, iOS, iPadOS, and tvOS.

Although the XNU kernel provides the flexibility of a microkernel with the performance of a monolithic one, there are some performance issues that arise when splitting an OS into a series of processes. The main issue that came about with the advent of the XNU kernel was the large amount of context switching necessary for interprocess messaging. This caused performance hits by as much as 25%, in some cases, when compared to traditional monolithic kernels.

## Conclusion

Comparing these three different implementations of process management and synchronization, we can clearly see basic OS process constructs, such as locks, semaphores, and signals are implemented in all of them, albeit in different ways for each. Windows provides a multitude of different frameworks and APIs for programmers to interact with the basic kernel functions. The XNU kernel for MacOS provides APIs very similar to those found in Windows. For the Linux kernel, the API is generally wrapper functions of the C library, libc. The intended use of these APIs is execute the correct syscall and to pass arguments to the appropriate register in an abstracted fashion. These syscalls will transfer control to the kernel where the majority of process management is done for these three kernels. Process management should be especially security-restricted and this is emphasized through most process protocols being executed within kernel space. Due to the Linux kernel source code being openly available, we can see some of the security mechanisms in place within process creation with copy_process() and clone_flags. The proprietary nature of Windows and MacOS abstracts some of the details of their security process, but use of APIs indicate they do not want programmers getting too close to the hardware without checks and balances.

# References


"Linux Documentation." *Linux Documentation*, https://linux.die.net/.

Drewbatgit. "Programming Reference for the Win32 API - Win32 Apps." *Win32 Apps | Microsoft Docs*, https://docs.microsoft.com/en-us/windows/win32/api/.

"Informit." *InformIT*, https://www.informit.com/articles/article.aspx?p=370047.

William. "Linux Kernel Vs. Mac Kernel." LinuxAndUbuntu, 6 Dec. 2019, http://www.linuxandubuntu.com/home/difference-between-linux-kernel-mac-kernel.

Mach Kernel Interface Reference Manual, http://web.mit.edu/darwin/src/modules/xnu/osfmk/man/.