

# **Operating System Feature Comparison: Processes and Acheduling**

by Sinan Topkaya

CS 444 - Spring 2016

Abstract: This paper will examine processes, threads and CPU scheduling for Windows and FreeBSD operating systems. It will mainly be about how these operating systems implement them and how is it different compared to Linux. Specifically, it will answer the questions: How do they differ? How are they the same? and Why do I think these similarities or differences exist?

## WRITING ASSIGNMENT - PROCESSES AND SCHEDULING COMPARISON

### *Windows*

Each Windows process is represented by an executive process (EPROCESS) structure. EPROCESS contains and points to a number of other related data structures. Likewise if each process has one or more threads, each of them are represented by an executive thread (ETHREAD) structure. To understand how Windows implements Processes and Threads we have to understand the structure EPROCESS and ETHREAD.

*Processes:* In this section I will write about how various data structures involved in process state manipulation and management. Then I will write about how and when those data structures are created and filled out, while creating or terminating processes.

[1] The EPROCESS and most of its data structures exist in system address space, except for process environment block (PEB), which exist in the process address space. The reason for that is because PEB contains information accessed by user-mode code. For each process that is executing Win32 program, the Win32 subsystem process (Csrss) maintains a parallel structure called the CSR\_PROCESS, kernel-mode part of the Win32 subsystem (Win32k.sys) maintains a per-process data structure, W32PROCESS. The W32PROCESS structure is created the first time a thread calls Windows USER or GDI function implemented in kernel code. Every EPROCESS structure is encapsulated as a process object by the executive object manager, but because processes are not named objects, they are not visible in WinObj tool. Many other drivers and system components, can choose to create their own data structures to track information on a per-process basis. It is also important to take data structure sizes into consideration.

**Process Object:** EPROCESS structure has many key fields. APIs and components are divided into layered modules with their own naming. For example one of the key field (member) of an executive process structure is called Pcb, or in other words process control block. This is a structure of type KPROCESS, for kernel process. Although routines store information in EPROCESS, the dispatcher, scheduler and interrupt/time use KPROCESS. Therefore allowing a layer to exist between high-level functionality and its underlying low-level implementation of functions, this also helps prevent unwanted dependencies between layers. Data structure PEB lives in the user-mode address space of the process, it contains information needed by image loader, the heap manager, and other Windows components that need to access it from user mode. The CSR\_PROCESS structure contains information about processes that is specific to the Windows subsystem. W32PROCESS structure contains information that the Windows graphics and window management code in

kernel needs to maintain state information about GUI processes. Each of these data structures have different process structure and hold important information about the Process.

CreateProcess: Creating a process involves many steps

- 1) Validate parameters; convert Windows subsystem flags and options to their native counterparts; parse, validate, and convert the attribute list to its native counterpart.
- 2) Open the image file(.exe) to be executed inside the process.
- 3) Create the Windows executive process object
- 4) Create the initial thread
- 5) Perform post-creation, Windows-subsystem-specific process initialization.
- 6) Start execution of the initial thread
- 7) In the context of the new process and thread, complete the initialization of the address space and begin execution of the program.

1 Stage 1: In CreateProcess, the priority class for the new process is specified in the CreationFlags parameter. If no priority class is specified for the new process, the priority class defaults to Normal. The most important thing to know here is that CreateProcess does not fail just because the caller has insufficient privileges to create the process. All windows are associated with the graphical representation of a workspace and again if no desktop is specified, the process is associated with the caller's current desktop. This stage mainly deals with validating parameters and converting flags and options to their native counterparts.

1) Stage 2: NtCreateUserProcess specifies the Windows image that will run the executable file. But just because a section object has been successfully created does not mean that it is a valid Windows image. It could be DLL or a POSIX executable. In case if the file is POSIX executable, the image to be run becomes Posix.exe and CreateProcess starts from stage 1. In case if the file is DLL then the creation fails. Once the executable is found it looks in the registry to see whether a subkey with the file name and extension exists there. If it does PspAllocateProcess looks for that value named Debugger for that key. If the value is there the image run becomes the string in that value and the process creation starts from Stage 1 again.

2) Stage 3: At this point NtCreateUserProcess has opened a valid Windows executable, and created a section object to map it into the new process. In this stage PspAllocateProcess is being called to run the image. This internal system function involves setting up the EPROCESS object, creating the initial process address space, initializing the kernel process structure(KPROCESS), setting up the PEB and

concluding the setup of the process address space.

- 3) Stage 4: At this point Windows executive process object is set up. The process still has no threads so it will not be doing anything. Now we have to create or insert a thread by using `PspAllocateThread` and `PspInsertThread`. `PspAllocateThread` handles the actual creation and initialization of the executive thread object while `PspInsertThread` handles the creation of the thread handle and security attributes. Also `KeStartThread` call can be used to turn the object into a schedulable thread on the system.
- 4) Stage 5: Once `NtCreateUserProcess` returns success, all executive process and thread objects have been created. At this point `Kernel32.dll` send a message to the Windows subsystem so that it can set up information. This message includes information on process and thread handles, entries in the creation flags, ID of the process creator, flag indicating whether if this process belongs to a Windows application, UI language information, DLL redirection, local flags and manifest file information. Once this message is received by the subsystem, `Csrss` process structure, thread structure, inserts, count of process are all allocated and initialized.
- 5) Stage 6: This is the stage when the process environment is determined, resources for its threads to use have been allocated, the process has a thread and the subsystem knows about the new process. Execution of the initial thread starts.
- 6) Stage 7: The new thread begins life running the kernel-mode thread startup routine `KiThreadStartup`. Then `PspUserThreadStartup` checks whether the debugger notifications have already been sent for this process, then `DbgkCreateThread` then waits for a reply from the debugger. After the debugger is notified, `PspUserThreadStartup` looks at the result of the initial check on the thread life. After this `PspUserThreadStartup` checks whether the systemwide cookie in the `SharedUserData` structure has been set up yet. If it has not, then it generates one based on hash system information. Finally `PspUserThreadStartup` sets up the initial context to run the image-loader.

*Threads:* Similarly Threads are formed with many Data Structures as well. The executive thread object encapsulates an `ETHREAD` structure, which in turn contains `KTHREAD` structure as its first member. Like Processes, `ETHREAD` structure and the other structures it points exist in the system address space, while the thread environment block (TEB) exists in the process address space. The Windows subsystem process maintains a parallel structure for each thread created, calling `CSR_THREAD`. The data structures have almost the same attributes as the process, for threads.

A life cycle of a Thread:

- 1) `CreateThread` converts the Windows API parameters to native flags and build native structure describ-

ing object parameters. `OBJECT_ATTRIBUTES`.

- 2) `CreateThread` builds an attribute list with two entries: client ID and TEB address. This allows those values to be received once the thread has been created.
- 3) `NtCreateThreadEx` is called to create user-mode context and capture the attribute list. After this `PspCreateThread` has been called to create a suspended executive thread object.
- 4) `CreateThread` allocates an activation context for the thread used by side-by-side assembly support. It then queries the activation stack to see if it requires activation, and it does so if needed. The stack is saved in TEB.
- 5) `CreateThread` notifies the Windows subsystem about the new thread, and the subsystem does some setup work for the new thread.
- 6) The thread handle and the thread ID are returned to the caller.
- 7) Thread is now resumed so that it can be scheduled for execution.

*CPU Scheduling:* [2] Windows CPU scheduling uses priority-based preemptive scheduling, the highest-priority runs next. The scheduler is Dispatcher. Thread runs until blocks, uses time slice and preempted by higher priority thread. If no run-able thread exists then it runs the idle thread. There is always a queue for each priority. Win32 API identifies several priority classes to which a process can belong to such as: `REALTIME_PRIORITY_CLASS`, `HIGH_PRIORITY_CLASS` or `ABOVE_NORMAL_PRIORITY_CLASS`. For example the high priority class will be the first in queue compared to the above normal priority class. A thread is also given priority class such as `NORMAL` or `BELOW_NORMAL`. In this case the normal priority thread will be the first in queue. The default priority is `NORMAL` within the class.

### *FreeBSD*

Similar to Windows, the threads of a process operate in either user mode or kernel mode.[3] The kernel state includes parameters to the current system call, and has 2 primary structures; process structure, thread structure. FreeBSD supports transparent multiprocessing, it does so by context switching, by switching between the execution context of the threads within the same or different process. A mechanism is also provided for scheduling the execution of threads, that is for deciding which one to execute next.

Now I will first write about how FreeBSD implements Process then I will talk about how it implements threads and how it deals with scheduling.

*Processes:* First I will write about the Process structure then I will briefly explain what they are, later I will write more about the state of a process. Every running command starts at least one new process and there are a number of system processes that are run by FreeBSD. Each process is uniquely identified by a number called PID. Similar to files each process has one owner and group, and the owner and group permissions are used to determine which files and devices the process can open. The command `%ps` allows the user to display a list of the currently running processes, their PIDs, and how much memory they are using, this is the structure of a process. The processes can be scheduled, to specify which process to run first or next.

Process Structure:

- 1) Process identification: the PID and the parent PID
- 2) Signal State: Signals pending delivery and summary of signal actions
- 3) Tracing: Process tracing information
- 4) Timers: Real-Time timer and CPU-utilization counters

Process states include NEW, NORMAL and ZOMBIE states. NEW specifies that there is an ongoing process creation, NORMAL specifies that threads will be runnable, sleeping or stopped, and the ZOMBIE state is when the process is undergoing termination. A process may create a new process that is a copy of the original by using the fork system call. The fork call returns twice: once in the parent process, where the return value is the process identifier of the child, and once in the child process, where the return value is 0.[4]

The kernel begins by allocating memory for the new process and thread entries. These thread and process entries are initialized in the steps: One part is copied from the parent's corresponding structure, another part is zeroed, and the rest is explicitly initialized. The zeroed fields include recent CPU utilization, wait channel, swap and sleep time, timers, tracing, and pending-signal information. The copied portions include all the privileges and limitations inherited from the parent.

*Threads:* Thread Structure

- 1) Scheduling: The thread priority, user-mode scheduling priority, recent CPU utilization and amount of time spent sleeping; the run state of a thread; additional status flags; if the thread is sleeping, the wait channel, the identifier of the event for which thread is waiting and a pointer to a string describing the event

- 2) TSB: the user- and kernel-mode execution states
- 3) Kernel Stack: the per-thread execution stack for the kernel
- 4) Machine state: The machine-dependent thread information

The `kthread_add()` is used to create a kernel thread. And the new thread runs in kernel mode only. It is later then added to the process specified by `procp` argument. If this thread wanted to be started then `kthread_start()` is used. The structures used by this function include name of the thread, the function for this thread to run. Later the caller must `sched_add()` to start the kernel. POSIX threads can also be created using FreeBSD.

*CPU Scheduling:* Kernel switches among threads in an effort to share CPU effectively; this activity is called context switching. Switching between thread can occur synchronously or asynchronously. Mutex Synchronization can be used for short-term thread synchronization.

There are number of ways the scheduling can be done, like timeshare thread scheduling, multiprocessor scheduling, adaptive idle and Traditional Timeshare Thread Scheduling.

For example multiprocessor scheduling is used when a runnable as a result of wakeup, unlock, thread creation, or other event is called by `sched_pickcpu()`. The best CPU to run is chosen (if there are multiple processors.) The whole system is searched for a least loaded CPU, and also checks to see the priority of the threads its running. Using several algorithms it decides the best fit CPU for the work.

*Answer to the questions*

*Windows*

*How is it different compared to Linux? and How is is similar to Linux? and Why?:* Process and Threads: In Windows a process holds the address space, handle table, statistics and at least one thread and it does not have an inherent parent/child relationship. While Linux process is called a task and although it also holds the information basic address space, handle table and statistics it does not have to have a thread. Linux does not have an inherent parent/child relationship and uses basic scheduling unit to schedule tasks. In Linux there are no threads per-se, tasks can act like Windows threads by sharing handle table, PID and address space. Windows uses basic scheduling unit to schedule threads and Fibers are used for cooperative user-mode threads while Pthreads are used for Linux for cooperative user-mode threads.

CPU Scheduling: In Windows there are 2 scheduling classes: Real-time and Dynamic, while Linux can have 3 scheduling classes, normal, fixed round robin and fixed fifo. Windows favors higher priorities while threads favor lower priorities. In linux priorities of normal threads decay as they use CPU, and priorities of interactive threads boost. In Windows priority of dynamic threads get boosted on wakeups and thread priorities are never lowered. In both operating systems the newly created thread starts with a base priority.

Although they have differences between each other, a user can ultimately accomplish the same things. It is a matter of what you need and if possible to know, want to use. We must understand that although the functions work differently they ultimately try to accomplish the same end result. For example `pthread_create` and `CreateThread` are really similar however they have different parameters.

### *FreeBSD*

*How is it different compared to Linux? and How is it similar to Linux? and Why?:* FreeBSD and Linux have different CPU scheduler, this mainly impacts the process management, specifically with regards to multicore scenarios. As mentioned above FreeBSD uses a lot of different kinds of algorithms for CPU scheduling. The similarity between each other is that they are both POSIX systems. If we are to compare default scheduler for both systems Linux uses CFS while FreeBSD uses ULE. Although ULE is similar to CFS it can be instructed to favor interactive processes.