Operating System Feature Comparison

by Sinan Topkaya

CS 444 - Spring 2016

Abstract: This paper will examine processes, threads and CPU scheduling, I/O, interrupts and memory management for Windows and FreeBSD operating systems. It will mainly be about how these operating systems implement them and how is it different or similar compared to Linux.

PROCESSES, THREADS AND CPU SCHEDULING

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 and executive thread(ETHREAD) structure. To understand how Windows implements Processes and Threads we have to understand the structure EPROCESS and ETHREAD. 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_PROCES`, 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 en capsulated 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.[1]

```
lkd> dt \verb|_kprocess|
\verb|nt!_KPROCESS|
+0x000 Header                          : \verb|_DISPATCHER_HEADER|
+0x010 ProfileListHead         : \verb|_LIST_ENTRY|
+0x018 DirectoryTableBase      : Uint4B
...
+0x074 StackCount                      : \verb|_KSTACK_COUNT|
+0x078 ProcessListEntry        : \verb|_LIST_ENTRY|
+0x080 CycleTime                       :  Uint8B
+0x088 KernelTime                      : Uint4B
+0x08c UserTime                        : Uint4B
+0x090 VdmTrapcHandler         : Ptr32 Void
```

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 a 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 inturrept/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 adress 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.

```
lkd> dt nt!_eprocess
+0x000 Pcb                          : _KPROCESS
+0x080 ProcessLock              : _EX_PUSH_LOCK
+0x088 CreateTime               : _LARGE_INTEGER
+0x090 ExitTime                 : _LARGE_INTEGER
+0x098 RundownProtect     : _EX_RUNDOWN_REF
+0x09c UniqueProcessId    : Ptr32 Void
...
+0x0dc ObjectTable              : Ptr32 _HANDLE_TABLE
+0x0e0 Token                    : _EX_FAST_REF
...
+0x108 Win32Process             : Ptr32 Void
+0x10c Job                          : Ptr32 _EJOB
...
+0x2a8 TimerResolutionLink : _LIST_ENTRY
+0x2b0 RequestedTimerResolution : Uint4B
+0x2b4 ActiveThreadsHighWatermark : Uint4B
+0x2b8 SmallestTimerResolution : Uint4B
+0x2bc TimerResolutionStackRecord : Ptr32 _PO_DIAG_STACK_RECORD
```

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.

```
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;
    }
```

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 jsut because the caller has insufficient privileges to create the process. All windows are associate with the graphical representation of a workspace and again if no desktop is specified, the process in associated with the caller current desktop. This stage mainly deals with validating parameters and converting flags and options to their native counterparts.
2) 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 valued 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. Incase if the file is DLL then the creation fails. Once the executable is found it looks in the registry to see whether a sub key 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.

3) Stage 3: At this point NtCrateUserProcess has opened a valid Windows executable, and created a section object to map it into the new process. Iin this stage PspAllcateProcess is being called to run the image. This internal system function involves seeting up the EPROCESS object, creating the initial process address space, initializing the kernel process strcuture(KPROCESS), setting up the PEB and concluding the setup of the process address space.
4) 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 suing PspAllocateThread and PspInsertThread. PspAllocateThread handles the actual creation and initialization of the executive thread object while PspInsetThread 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.
5) 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 is 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 recieved b the subsystem, Csrss process structure, thread structure, inserts, count of process are all allocated and inialized.
6) Stage 6: This isthe stage when the process is 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.
7) Stage 7: The new thread begins life running the kernel-mode thread startup routine KiThreadStartup. Then PspUserThreadStartup cehcks whether the debugger notifications have already been send for this process, then DbgkCreateThread then waits for a reply from the debugger. After the debugger is notifies, PspUserThreadStartup looks at the result of the initial check on the thread life. After this PspUserThreadStartup checks whether the system wide cookie in the SharedUserData structure has been set up yet. If is has not, then it generates one based on hash system information. Finally PspUserThreadStartup sets up the initial context to run the image-loader[1]

Similarly, Threads are formed with many Data Structures as well. The executive thread object encapsulates an ETHREAD structure, which in return 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 strcuture describing object parameters. `OBJECT_ATTRIBUTES`.
2) CreateThread builds an attribute list with two entries: client ID and TEB address. This allows those values to be recived 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 a 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 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 is can be scheduled for execution. [1]

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 comapred 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. In FreeBSD, similar to Windows, the threads of a process operate in either user mode or kernel mode. The kernel state includes parameters to the current system call, and have 2 primary structures; process structure, thread structure. FreeBSD supposes transparent multiprogramming, it does so by context switching, by swithcing 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. 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 identifying 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.[2] Process Structure:

1) Process identification: the PID and the parent PID
2) Signal Slate: 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 a undergoing process creation, NORMAL specifies that threads will be runnable, sleeping or stopped, and the ZOMBIE state is when the process in 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 return 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. 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 initilized. 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 privilages and limitation inherited from the parent. Thread Structure

1) Scheduling: The thread priority, use-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 identify 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[2]

The `kthread_add()` is used to create a kernel thread. And the new thread run in kernel mode only. It is alter 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 includes 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. Kernel switches among threads in an effort to share CPU effectively; this activity is called context switching. Switching between thread can occurs 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.[2]

In conclusion, in Windows a process holds the address space, handle table, statistics and at least one thread and it does not 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 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 Pthreats are used for Linux for cooperative user-mode threads 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 prioritie 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 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.

References

[1] D. S. Mark Russinovich and A. Ionescu, *Windows Internals, Part 1 6/e*. Microsoft Press, 2012.

[2] M. K. McCusick and G. V. Neville-Neil, *Design and Implementation of the FreeBSD Operating System 2/e*. Addison-Wesley, 2015.