CS 444 - Spring 2016 - Writing Assignment 1

Cody Malick

malickc@oregonstate.edu

Abstract

Understanding how an operating system handles processes, threads, and CPU scheduling are important topics for an aspiring kernel developer to understand. In this report, we will cover how Linux, Windows, and FreeBSD handle these important tasks in their respective kernels, and contrast them.

CONTENTS

I. Introduction

This report contains descriptions on how each of the operating systems, Linux, Windows, and FreeBSD handle processes, threads, and CPU scheduling. A lot can be learned about the design decisions of the minds behind the operating system by how they handle these core constructs. We will start by describing how Linux handles these, as Linux is the focus of this class.

II. Linux

A. Processes

The first thing that should be defined, is simply what a process is: "A *process* is a program (object code stored on some media) in the midst of execution."[1] Although the Linux kernel refers to processes as tasks, in this report (for uniformity) we will only refer to them as processes.

A process in Linux is ultimately born, somewhere up the line, from the `init` process. The `init` is the absolute root of every process tree on any Linux machine. It has a process ID of 1. All processes are spawned from either the `fork()` or `exec()` function calls. `Fork()` creates an exact copy of the calling process as a child of the process, while `exec()` loads a new executable into the process space and executes it. These are the only ways new processes are created in Linux.

Each process has a process descriptor that describes every aspect of the process, the `task_struct`. The `task_struct` is contained in `<linux/sched.h>`. `Task_struct` describes the current state of the process, the parent process ID, the process ID, exit code, priority, etc. Everything you could possibly want to know about a process in Linux is in that struct. I've included the `Task_struct` as Appendix A as it is a very large structure.

Processes, of course, must have states of execution. On Linux, processes have five states, each represented by flags in the `task_struct`:

`TASK_RUNNING`: The process is either running or in a run-queue ready to be run (run-queues will be discussed in the CPU scheduling section).

`TASK_INTERRUPTIBLE`: the process is sleeping (blocked) waiting for a condition to un-block. Once the condition is met, it transitions to `TASK_RUNNING`.

`TASK_UNINTERRRUPTIBLE`: The same state as `TASK_INTERRUPTIBLE` with the difference being that it does not wake once an interrupt is received.

`_TASK_TRACED`: This flag shows that the process is being traced by another process. An example is GDB.

`_TASK_STOPPED`: Process execution has halted and it is not possible to start running. This state is a result of a halting interrupt from the CPU, or if a debugger sends it a signal.

B. Threads

As with processes, we will define what a thread is: "Threads of execution, often shortened to *threads*, are the objects of activity within the process." [1] In Linux, threads are treated fundamentally the same as processes. The only difference is that they have a shared address space, filesystem resources, file descriptors, and signal handlers. Here is the `thread_info` struct:[1]

```
struct thread_info {
    struct task_struct *task;
```

```
    struct exec_domain     *exec_domain;
    unsigned long          flags;
    unsigned long          status;
    __u32                  cpu;
    __s32                  preempt_count;
    mm_segment_t           addr_limit;
    struct restart_block   restart_block;
    unsigned long          previous_esp;
    __u8                   supervisor_stack[0];
};
```

Compared to the `task_struct`, this is a realitively simple data structure. It has a pointer to the `task_struct` that is unique to that thread, along with a few extra flags and sets of information that seperate threads from processes.

C. CPU Scheduling

Linux ships with a few different schedulers by default, including a real time scheduler.[2] The default scheduler for Linux is the CFS, or the Completely Fair Scheduler. The CFS is only fair in so far as it tries its best to split the processors time into equal bits, 1/n, where n is the number of processes currently running. The CFS weights this with process priority, or nice value, and the higher the weight the larger percentage of processor time that process gets. So it's not completely fair, but that is the idea behind it.

One of the major problems with this system is that a very large chunk of processes creates incredibly small slices of CPU time. For example, if there were six-thousand processes and/or threads that all had the same weight, then the cpu slice for each process would be 1/6000. There is a point where that slice is not large enough to even complete the context switch required to start executing the process again. The CFS handles this by implementing a floor on slice size, known as *minimum granularity*.[1] With this system, every process will always have a minimum amount of run time, but the system can become very slow once this size is reached.

III. Windows

Windows is a very different beast. The process and thread structure are quite a bit different from Linux. The CPU scheduler Windows uses has the same basic idea as the CFS, but varies greatly when it comes to how it handles priority and division of CPU time.

A. Processes

Processes in Windows are similiar to Linux in only one way: the both have a process ID. Otherwise, processes in Windows handle very differently than Linux. Here is the basic process struct in Windows, `_PROCESS_INFORMATION`: [3]

```
typedef struct _PROCESS_INFORMATION {
  HANDLE hProcess;
  HANDLE hThread;
  DWORD dwProcessId;
```

```
    DWORD dwThreadId;
} PROCESS_INFORMATION, *LPPROCESS_INFORMATION;
```

The `HANDLE` data type is a 32-bit unique identifier stored in the Windows kernel to identify individual processes.[4] The `hThread` points to the primary thread a process is executing on. These first two of these identifying fields are used internally by the kernel to specify what functions are performing operations on the process and thread objects. The second set of identifiers, `dwProcessId` and `dwThreadId` are used as unique identifiers for the process and thread themselves. These are the equivalent of the PID used by the the Linux `tast_struct`.

Process creation is also differnt in Windows. Instead of the constantly traceable tree of child-parent relationships, a function called `CreateProcess()` is invoked, resulting in a new process completely independent of the parent process. [5]

Process state in Windows is determined by thread state. Here are the possible thread states in Windows:[6]

`Initialized`: A state that indicates the thread has been initialized, but has not yet started.

`Ready`: A state that indicates the thread is waiting to use a processor because no processor is free. The thread is prepared to run on the next available processor.

`Running`: A state that indicates the thread is currently using a processor.

`Standby`: A state that indicates the thread is about to use a processor. Only one thread can be in this state at a time.

`Terminated`: A state that indicates the thread has finished executing and has exited.

`Transition`: A state that indicates the thread is waiting for a resource, other than the processor, before it can execute. For example, it might be waiting for its execution stack to be paged in from disk.

`Unknown`: The state of the thread is unknown

`Wait`: A state that indicates the thread is not ready to use the processor because it is waiting for a peripheral operation to complete or a resource to become free. When the thread is ready, it will be rescheduled.

This is interesting for a few reasons. First, there is an 'unknown' thread state. In Linux, it is isn't possible for a thread or process to be in an unknown state, only the ones listed. Next, it's interesting to see how Windows has so many more states than Linux. One of the differences that sticks out most is that Windows has a state when a process is queued to enter the processor in the `Standby` state. By having all these different states available, you can really see the real time state of the process, as apposed to Linux where you have fewer states that cover less situations. Then again, this could just be a result of Linux having better processor and thread management, requiring less states.

B. Threads

Threads, in Windows, are handled as subcomponents of processes. They are grouped into thread pools, where multiple threads execute under one process. MSDN explains the idea behind this structure is to reduce the number of application threads, and provide management of worker threads. Threads are are spawned through the `CreateThread()` function call.[7]

C. CPU Scheduling

The Windows CPU scheduler, by default, schedules processes in much the same way that the CFS schedules processes. A process is given the default weight, and the scheduler tries to divide the CPU time evenly among the running processes.

Once we introduce weights or priority, the scheduler's behavior is very different.

While the system is assigning CPU time slices to each process, if a higher priority thread becomes available, the CPU stops the current running process, mid-time slice, and starts running the higher priority threads.

This is vastly different from the CFS in that regard. The CFS will allow a current running thread to complete it's time slice before moving it out of the processor. By introducing this kind of behavior, the power of priority states becomes glaringly clear.

Windows has thirty-two levels of priority, each divided into subdivisions:

`IDLE_PRIORITY_CLASS`

`BELOW_NORMAL_PRIORITY_CLASS`

`NORMAL_PRIORITY_CLASS`

`ABOVE_NORMAL_PRIORITY_CLASS`

`HIGH_PRIORITY_CLASS`

`REALTIME_PRIORITY_CLASS`

Each of these classes has subdivisions inside of them, seven in each, that has steadily increasing higher priority, starting from one, rising to thirty-two. As a side note, the zero priority class is reserved for a thread that goes through and zeroes out memory when a process terminates.[8] Besides this large difference in priority, the two CPU schedulers are very similar.

IV. FreeBSD

FreeBSD is an open source OS based off Unix, specifically the Berkeley variant, BSD. It looks and feels similar to Linux in a few ways, but has some subtle differences in process and thread structure. The scheduler is quite a bit different from the CFS.

A. Processes

Processes in FreeBSD are very similar to Linux's. Similiar to Linux's process struct, `task_struct`, the FreeBSD `proc` struct is large and contains a lot of information. Processes are created using the same forking and exec process that Linux uses. After a process is forked, a new process that is identical, except for the PID, is created. The process has a copy of the parent's resources and addresses space.

Process states are handled in a much more simplified manner than either Windows or Linux in FreeBSD. Instead of having eight or five different states for a process, it only has three:

`NEW`: Process is undergoing creation

`NORMAL`: Thread or threads will be runnable, sleeping, or stopped

`ZOMBIE`: The process is undergoing process termination

As per above, `NORMAL` embodies three subdivisions, runnable, sleeping, or stopped, but overall the classification is simplified. [9]

B. Threads

A notable difference between Linux and FreeBSD is that it does not treat threads and processes the same way. Threads are handle in a more simliar fashion to Windows. Each thread is attached to a parent process. If multiple threads are requested

for one process, new processes are spawned, but they have the same PID. When a user looks for a process in the OS, they will only see the single entry representing the parent process.

Thread states are dictated by the list in the processes section, but have a caviat. When a thread is not running on the processor, which there can only be one of, all threads are in one of three queues: the run queue, the sleep queue, or the turnstile queue. As the names dictate, the run queue contains threads that are in a runnable state. Threads that are blocked and are awaiting events are in the sleep queue or the turnstile queue.

The turnstile queue is exclusively for short blocks on threads. In FreeBSD, short blocks are exclusively the result of read/write blocks. Because these kinds of blocks are common, each thread creates a turnstile queue when it is initialized, and creates a list of all threads that are blocked because of a read/write block on a certain piece of memory. When that block is lifted, the turnstile queue manages the order in which the blocked threads are resolved. Allocating turnstile queues on each thread as apposed to each lock results in lower memory usage by the kernel.

Sleep queues are similar to turnstiles in that they contain lists of threads where blocking is occuring, but the sleep queue is exclusively for medium to long term locks that can send interrupts if the lock is held for too long. [9]

C. CPU Scheduling

The default CPU scheduler for FreeBSD is the ULE scheduler. ULE is not an acronym. [9] The scheduler is contained in the namespace `syskernsched_ule.c`, and if you remove the underscore in the namespace, you will see the reasoning behind the name.

The ULE scheduler is split into two parts in FreeBS: a low-level scheduler, and a high-level scheduler. The low-level scheduler runs often, and the high-level scheduler runs a few times a second.

The low-level scheduler runs extremely often, whenever a thread is blocked. When a block occurs, the low-level scheduler pulls the highest priority process from a set of run queues. These queues are organized from low to high priority. The high-level scheduler decides what the priority of each thread is, and sorts it into the appropriate run queue. Each core has it’s own set of these run queues to prevent two cores from trying to access the same queue at the same time.

The different cores run a round robin system on the threads in the queues. Each thread gets equal runtime in the form of time quantums. These time quantums are the same as time slices. If a process uses an entire time quantum, it is moved to the back of the the queue that it came from, and a context switch occurs to the next highest priority thread.[9]

V. Appendix A

The `task_struct`:[10]

```
struct task_struct {
/* these are hardcoded - don't touch */
 volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
 long counter;
 long priority;
 unsigned long signal;
 unsigned long blocked; /* bitmap of masked signals */
 unsigned long flags; /* per process flags, defined below */
 int errno;
 long debugreg[8]; /* Hardware debugging registers */
 struct exec_domain *exec_domain;
/* various fields */
 struct linux_binfmt *binfmt;
 struct task_struct *next_task, *prev_task;
 struct task_struct *next_run, *prev_run;
 unsigned long saved_kernel_stack;
 unsigned long kernel_stack_page;
 int exit_code, exit_signal;
 /* ??? */
 unsigned long personality;
 int dumpable:1;
 int did_exec:1;
 int pid;
 int pgrp;
 int tty_old_pgrp;
 int session;
 /* boolean value for session group leader */
 int leader;
 int groups[NGROUPS];
 /*
  * pointers to (original) parent process, youngest child, younger sibling,
  * older sibling, respectively. (p->father can be replaced with
  * p->p_pptr->pid)
  */
 struct task_struct *p_opptr, *p_pptr, *p_cptr,
                    *p_ysptr, *p_osptr;
```

```
    struct wait_queue      * wait_chldexit ;
   unsigned short          uid, euid, suid, fsuid;
   unsigned short          gid, egid, sgid, fsgid;
   unsigned long           timeout, policy, rt_priority ;
   unsigned long           it_real_value , it_prof_value , it_virt_value ;
   unsigned long           it_real_incr , it_prof_incr , it_virt_incr ;
    struct timer_list      real_timer ;
   long                    utime, stime, cutime, cstime, start_time ;
/* mm fault and swap info: this can arguably be seen as either
     mm-specific or thread-specific */
   unsigned long           min_flt , maj_flt , nswap, cmin_flt , cmaj_flt , cnswap;
    int swappable:1;
   unsigned long           swap_address;
   unsigned long           old_maj_flt ;      /* old value of maj_flt */
   unsigned long           dec_flt ;          /* page fault count of the last time */
   unsigned long           swap_cnt;          /* number of pages to swap on next pass */
/* limits */
    struct rlimit          rlim[RLIM_NLIMITS];
   unsigned short          used_math;
   char                    comm[16];
/* file system info */
    int                    link_count ;
    struct tty_struct      * tty ;            /* NULL if no tty */
/* ipc stuff */
    struct sem_undo        *semundo;
    struct sem_queue       *semsleeping;
/* ldt for this task - used by Wine. If NULL, default_ldt is used */
    struct desc_struct     * ldt ;
/* tss for this task */
    struct thread_struct   tss ;
/* filesystem information */
    struct fs_struct       *fs;
/* open file information */
    struct files_struct    * files ;
/* memory management info */
    struct mm_struct       *mm;
/* signal handlers */
    struct signal_struct   *sig;
```

```
#ifdef __SMP__
    int                 processor;
    int                 last_processor;
    int                 lock_depth;     /* Lock depth.
                                           We can context switch in and out
                                           of holding a syscall kernel lock ... */
#endif
};
```

VI. Appendix B

FreeBSD `proc` struct: [11]

```
/*
* Description of a process.
*
* This structure contains the information needed to manage a thread of
* control, known in UN*X as a process; it has references to substructures
* containing descriptions of things that the process uses, but may share
* with related processes. The process structure and the substructures
* are always addressable except for those marked "(PROC ONLY)" below,
* which might be addressable only on a processor on which the process
* is running.
*/
struct proc {
        struct proc *p_forw;            /* Doubly-linked run/sleep queue. */
        struct proc *p_back;
        struct proc *p_next;            /* Linked list of active procs */
        struct proc **p_prev;           /*      and zombies. */

        /* substructures: */
        struct pcred *p_cred;           /* Process owner's identity. */
        struct filedesc *p_fd;          /* Ptr to open files structure. */
        struct pstats *p_stats;         /* Accounting/statistics (PROC ONLY). */    struct plimit *p_limit;
                /* Process limits. */
        struct vmspace *p_vmspace;      /* Address space. */
        struct sigacts *p_sigacts;      /* Signal actions, state (PROC ONLY). */

#define p_ucred         p_cred->pc_ucred
#define p_rlimit        p_limit->pl_rlimit

        int     p_flag;                 /* P_* flags. */
        char    p_stat;                 /* S* process status. */
        char    p_pad1[3];

        pid_t   p_pid;                  /* Process identifier. */
        struct proc *p_hash;    /* Hashed based on p_pid for kill+exit+... */
        struct proc *p_pgrpnxt; /* Pointer to next process in process group. */
        struct proc *p_pptr;    /* Pointer to process structure of parent. */
```

```
        struct  proc *p_osptr;  /* Pointer to older sibling processes. */

/* The following fields are all zeroed upon creation in fork. */
#define p_startzero     p_ysptr
        struct  proc *p_ysptr;  /* Pointer to younger siblings. */
        struct  proc *p_cptr;   /* Pointer to youngest living child. */
        pid_t   p_oppid;        /* Save parent pid during ptrace. XXX */
        int     p_dupfd;        /* Sideways return value from fdopen. XXX */

        /* scheduling */
        u_int   p_estcpu;       /* Time averaged value of p_cpticks. */
        int     p_cpticks;      /* Ticks of cpu time. */
        fixpt_t p_pctcpu;       /* %cpu for this process during p_swtime */
        void    *p_wchan;       /* Sleep address. */
        char    *p_wmesg;       /* Reason for sleep. */
        u_int   p_swtime;       /* Time swapped in or out. */
        u_int   p_slptime;      /* Time since last blocked. */

        struct  itimerval p_realtimer;  /* Alarm timer. */
        struct  timeval p_rtime;        /* Real time. */
        u_quad_t p_uticks;              /* Statclock hits in user mode. */
        u_quad_t p_sticks;              /* Statclock hits in system mode. */
        u_quad_t p_iticks;              /* Statclock hits processing intr. */

        int     p_traceflag;            /* Kernel trace points. */
        struct  vnode *p_tracep;        /* Trace to vnode. */

        int     p_siglist;              /* Signals arrived but not delivered. */

        struct  vnode *p_textvp;        /* Vnode of executable. */

        char    p_lock;                 /* Process lock (prevent swap) count. */
        char    p_pad2[3];              /* alignment */

/* End area that is zeroed on creation. */
#define p_endzero       p_startcopy

/* The following fields are all copied upon creation in fork. */
```

```
#define  p_startcopy        p_sigmask

         sigset_t  p_sigmask;       /* Current signal mask. */
         sigset_t  p_sigignore;     /* Signals being ignored. */
         sigset_t  p_sigcatch;      /* Signals being caught by user. */

        u_char    p_priority;       /* Process priority. */
        u_char    p_usrpri;         /* User-priority based on p_cpu and p_nice. */
        char      p_nice;           /* Process "nice" value. */
        char      p_comm[MAXCOMLEN+1];

         struct    pgrp *p_pgrp;    /* Pointer to process group. */

         struct    sysentvec *p_sysent; /* System call dispatch information. */

         struct    rtprio  p_rtprio;        /* Realtime priority. */
/* End area that is copied on creation. */
#define  p_endcopy          p_addr
         struct    user *p_addr;    /* Kernel virtual addr of u-area (PROC ONLY). */
         struct    mdproc p_md;     /* Any machine-dependent fields. */

         u_short   p_xstat;         /* Exit status for wait; also stop signal. */
         u_short   p_acflag;        /* Accounting flags. */
         struct    rusage *p_ru;    /* Exit information. XXX */
};
```

VII. Bibliography

References

[1] R. Love, *Linux Kernel Development*, 3rd ed. Developer's Library, 2010.

[2] R. Hat. (2016, jan) 4.2 cpu scheduling. [Online]. Available: https://access.redhat.com/documentation/en-US/RedHatEnterpriseLinux/6/html/PerformanceTuningGuide/s-cpu-scheduler.html

[3] MSDN. (2016, jan) Process_information structure. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms684873

[4] ——. (2016, jan) Handles and objects. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms724457

[5] ——. (2016, jan) Creating processes. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682512

[6] ——. (2016, jan) Threadstate enumeration. [Online]. Available: https://msdn.microsoft.com/en-us/library/system.diagnostics.threadstate(v=vs.110).aspx

[7] ——. (2016, jan) Creating threads. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682516

[8] ——. (2016, jan) Scheduling priorities. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms685100

[9] K. McCusick and G. V. Neville-Neil, *Design and Implementation of the FreeBSD Operating System*, 2nd ed. Pearson Education, 2015.

[10] L. Torvalds. (2016, mar) sched.h. [Online]. Available: https://github.com/torvalds/linux/blob/master/include/linux/sched.h

[11] FreeBSD. (2016, jan) struct proc structure. [Online]. Available: http://people.freebsd.org/ meganm/data/tutorials/ddwg/ddwg63.html