Final Paper

Taylor Falhman

CS444 Fall 2015

1 Processes and Threads

A process is the execution of a program and a fundamental feature of an operating system. Operating system handles these programs at many levels, including creation, execution, scheduling and threading. Process are given memory and other various metadata attributes so the kernel can reference it.
In FreeBSD, a process has at least one thread, with the possibility of having multiple threads. Each thread has two options for execution: user mode and kernel mode, which in turn have corresponding resource structures. The user mode resources are usually general purpose registers, the program counter and the like found in the CPU for all process. The kernel mode in contrast uses the registers and other counters that the hardware typically uses. The kernel state divides these resources into two structures, the process structure and the thread structure. The process structure holds the main information that the kernel needs to know about the process, while the thread structure only holds the information needed at the time of execution, and may not always have information in it.
A process in FreeBSD will contain the following: a process ID, a signal state, tracing information, and timers to keep track of real time and CPU time. Substructures include the calling user's credentials, memory mapping, file descriptors, system call mapping, statistics, and the thread structure to name most of them. Processes are categorized into two groups; a zombie list and an all processes list. Process can be in only one list at a time, and this is only used to make the zombie killing time more efficient. Threads are also categorized into one of three 'queues': the run queue, where runnable threads are placed, the sleep queue, or the turnstile queue.
Once a thread has finished its allotted time executing, or is finished, the CPU uses context switching. Context switching means that the CPU is switching execution from one thread to another, which means that the done process will be placed on the sleep or turnstile queue, and one will be move from one of those queues to the running queue.
FreeBSD initially used a threading model called 'N:M threading',but this caused extra overhead. In the end, the developers decided to implement the POSIX threading standard, pthreads. This includes creation, scheduling, and signaling. Threads are created using the pthread create system call. Pthreads from the same parent process all share the same shared memory. Pthreads use the mutex as the main method of multi-threaded programing control. There also exists the fork system call in FreeBSD, in which a process can create a completely separate process, as opposed to just a thread.
The default scheduler for FreeBSD is called the ULE scheduler. It's divided into a low and a high level scheduler. The low level part gets the next highest priority thread in the run queue and sends it to execute. The high level part orders the threads by priority for each CPU. [2]

Windows have a slightly different model for what processors and threads do, but there are a multitude of similarities with FreeBSD and POSIX implementations.
Processes are represented by the 'executive process' structure, which in turn holds many other structures full of information about a process. Each executive process also has a corresponding kernel mode data structure, called a W32PROCESS. After all the process structures have been created, a thread must be created. This shows a major difference between the Windows and FreeBSD implementations. FreeBSD treats threads and processes the same in terms of creating a process and execution. But in Windows, a thread is created using a completely different system call, and processes aren't executed, only the threads within them. Processes only provide the context in which the thread will use in its execution in Windows.
This difference probably came about in an attempt to simplify and separate ideas during the implementation of Windows. Since the difference between a process and a thread and small for FreeBSD and it's POSIX-using UNIX cousins, the two can get confusing. Microsoft probably wanted to simplify and break up responsibilities between the process and thread. This also allows developers and users to know exactly what they are looking for in debugging or inspecting the Task Manager.
Windows uses a priority system for its CPU scheduling that FreeBSD does. The priority of affinity is determined by a processor group, unless otherwise set by the developer, which can set the group or affinity using the scheduling API. Thread priority isn't affected by the process it belongs to; the scheduler only sees the total number of threads and tries to give each an equal amount of time on the CPU. [6]
Priority is based off of two factors: the priority class given at creation time given to the process, and the individual thread priority. Priority can also be increased (or less commonly, decreased) for a number of reasons. Most of these are related to normal scheduler and I/O reasons. However, since Windows is such a GUI-focused operating system, some of the built in reasons are related to the GUI. For example, a process get a priority boost after the GUI thread wakes up and starts execution. Another example is that CPU intensive programs, like video games and video players gets a large boost, and even a special service dedicated to making sure that all of their respective execution requests are seamless. This is again due to the unique focus of Windows on the GUI.

The main differences between processes, threads and scheduling in Windows and FreeBSD overall seem to stem from the developers that are using Windows; in general, many of the ideas and constructs are reimplemented to fit the Windows philosophy. In this case, the focus on the graphical utilities and the want to simplify established terminology and ideas seems to have motivated Microsoft's changes. The reason the scheduler is the same is because the priority-based scheduler has been a tried and true method of mostly-fair scheduling.

2 Memory Management

One of the most important components of an operating system in the memory management. Memory is a necessary and fundamental part of computing, so managing memory is essential to a functional kernel. In the FreeBSD kernel, the memory is organized into three categories: main memory (usually RAM), secondary storage (disk drives) and backing storage.
Process, when created, are given a virtual address space that exists outside of the context of the physical memory. While this virtual memory is mapped in physical memory, it is not necessarily in sequential order, and does not all have to reside in main memory. When the virtual space is addressed, the kernel converts that into its corresponding physical memory address. This can cause problems when there a many processes existing in main memory as we don't want to access the memory of another process. The memory management unit is the service responsible for this, which keeps track of the start and end of each process memory segment, as well as handles the virtual to physical address translation so as to validate the request.
Virtual memory is most efficient when based in hardware, which uses a fixed-size memory unit called a page. The MMU takes the virtual address, translates it into a page reference found in main memory and an offset within that page to find its corresponding physical address. A common way to implement this is forward-mapped page tables. This design has a table of pages tables that reference a physical address. This design uses a logical address that has the index of the outer table, the offset to use in the page table and the offset to use in the physical address. This allows a 4 MB physical space to reference a 4 GB address space assuming a 4 KB virtual page size and a 32-bit page table. While most processes don't use this much, this allows for a great amount of virtual memory, especially on 64-bit architectures.
These pages exist in both main memory and secondary memory. Therefore, there is a designation as to which ones live where. If a page is a 'resident', then the request is validated and the physical memory can be read from or executed. However, if the page is a 'non-resident', an event called paging occurs. Paging can happen for several reasons, but often indicated memory shortage. The vm fault routine handles and attempts to fix these faults. It finds where the page fault occurred, and tries to allocate the correct page the process needs. Then it asks another part of the kernel for validation and starts the process again. This following code from Chapter 6 of the FreeBSD book [4] shows the page-fault-handling algorithm.

```
int vm_fault (
        vm_map_t map,
        vm_offset_t addr,
        vm_prot_t type)
{
    RetryFault:
        lookup address in map returning object/offset/prot;
        first_object = object;
        first_page = NULL;

        //Traverses list of vm_objects until find item with page, or reaches final and reques
        for(;;){
            page = lookup page at object/offset;

            //The page is found!
            if(page found) {
                //If the page is busy, another process could be using it
                //So we goto the retryfault until its not busy
                if(page busy)
                    block and goto RetryFault;
                //Otherwise, get the page
                remove from paging queues;
                mark page as busy;
                break;
            }

            /*
             *Omitted some non-relavant code
             */

            //Makes sure the first page is remembered
            if(object == first_object)
                first_page = page;
            next_object = next object;

            //Makes the first object handle the fault if the right page isn't found
            if(no next object) {
                if(object != first_object) {
                    object = first_object;
                    page = first_page;
                }
                first_page = NULL;
                zero fill page;
                break
            }
            object = next_object
        }

        //The found or allocated page
        orig_page = page;

        /*
         *Ommited some non-relavant code
         */

        //Now the page can safely be made writeable
        if(prot & WRITE)
            mark page not copy-on-write;
        enter mapping for page;
        enter read-only mapping for clustered pages;

        //Now processes waiting on this page can use it
        activate and unbusy page;
        if(first_page != NULL)
            unbusy page;
}
```

Windows also has a memory management service, called the Memory Manager. It functionally does the same thing as the MMU; translating virtual address requests to the corresponding physical address, and handling paging when there is a fault.

Pages conceptually are the same, where they represent the smallest unit of virtual memory. However, it differs from FreeBSD in that it has two types of pages, large and small. The large pages are pages that have more data in them, and as a result, less entries in the page table are used, and because of the small size of the table, more data is able to fit into virtual memory. To make use of this, Windows maps most of its bigger components, as well as applications memory needs to large pages. The kernel allows device drivers to explicitly be mapped with large pages. But large pages do not always work. The longer a system is turned on, the less physical space in general is available. Because of the amount of contiguous memory a large page requires, a large enough space may not be available due to fragmentation over time. Small pages do not suffer from this. Large pages are also not pageable, so they cannot

be written to the filesystem. This makes small pages more versatile. Another downside of large pages is that all data must be read/write. If an application has a mix of read-only and read-write data, using a large page will allow writes into the intended read-only data. This poses a security hole and, while small pages do reduce efficiency, Microsoft recommends using small pages to solve this.
In term of virtual to physical address translation, Windows implements the same model as FreeBSD, with the forward-mapped page tables. This is probably the most efficient way or possible the only viable way for virtual memory to work in a 32-bit architecture as it is found in both kernels and their implementations are virtually identical.
Windows also has a page fault handler that tries to fix invalid pages and page table entries. One large difference is that in order to preempt these invalid requests, the kernel prefetches a large amount of pages and stores them in the page cache. This allows for a quick and efficient fix when a page fault occurs, and costs no overhead in term of system resources. If a page is in the cache and is requested, then you get fast turn around. If it's never requested or already exists in memory, then it doesn't cost the system anything. It's a very convenient feature to have. [5]
While the concept of pages and the implementation of virtual memory is similar between the two kernels, the page fault handler and pages themselves have a few 'fancy' features in Windows. The reason for this is most likely the specialized nature of Windows. Because FreeBSD is open source, the developers attempt to make it as generalized as possible, which leads to basic and efficient implementations that can run well on a variety of systems. Windows on the other hand is much more specialized Operating System and targets specific hardware more often than FreeBSD. Also, because it is enterprise, and has very application based user space, things like large pages, and extra page fault features make sense.

3 Cryptography

Cryptography is an important part of digital security. Because of this, most kernels, including FreeBSD and Windows, implement a cryptographic API for use within the kernel.
FreeSD calls their crypto API the 'crypto subsystem'. It supports both symmetric and asymmetric cryptography, but symmetrical encryption is more common in the case of the kernel. There are two sets of APIs that make up the subsystems: the software and the device driver APIs. [3]
The model of using the crypto subsystem involves a session where the requesting process starts a session, puts the work it wants done on a queue and then waits for the work to be done and returned. This is shown in the following code example.

```
#include <opencrypto/cryptodev.h>

    //DRIVER SIDE SETUP
    //Recieves a device id
    device_id = crypto_get_driverid(0)
    //Registers a crypto algorithm
    data_value = crypto_register(device_id, ALG_ID, MAX_OPERATOR_BITS, ALG_FLAGS)


    //Begins a new sessions with the algorithm and the device
    session_id = newsession(data_value, device_id)

    //Handle work in here

    //Ends session thats refered to by session_id
    freesession(data_value, session_id)

    //Unregister support for an algroithm
    crypto_unregister(device_id, ALG_ID)
```

Windows also has a cryptographic API within the kernel. While it provides support for cryptographic service providers, or CSPs, only the Microsoft RSA base provider comes in the kernel. Each CSP can implement the API differently, but all have a similar architecture. Each CSP has a key database which stores all the key information for users and applications.
The various API calls allow users to encrypt and digitally sign files. It also allows for generating keys, using hashes

and other standard cryptographic functionality. There is not much special here other than the use of the key database to store everything.[1]
Again, this is most likely due to enterprise nature of Windows, where architectures like this make sense because Microsoft builds and maintains the features. If FreeBSD had a similar design, it would be difficult for users to maintain such a database most likely.
The API itself also doesn't worry about sessions and requests as long as the context handle is present in Windows. This shows how Windows treads handles as a special thing; it's very focused on files and file handles as a way of protection and addressing it.
The following code exemplifies and highlights the differences between the Windows and FreeBSD CryptoAPI calls.

```
#include <wincrypt.h>        // CryptoAPI definitions
#define MS_DEF_PROV          "Microsoft Base Cryptographic Provider v1.0"
#define PROV_RSA_FULL        1
#define CRYPT_NEWKEYSET      0x8

BOOL bResult;
HCRYPTPROC hProv;


 bResult = CryptAcquireContext(
              &hProv,              // Variable to hold returned handle.
              NULL,
              MS_DEF_PROV,         //Using the built in CSP
              PROV_RSA_FULL,       //Using RSA
              CRYPT_NEWKEYSET);    //The documentation says this will initialize the database

 //Once the handle is acquired, do the work here

 // Release handle once we're done with it
 CryptReleaseContext(hProv);
```

Overall, despite differences in the models used, the two APIs do the same thing. Cryptography is such an essential component of modern computing that it needs to be as portable as possible. This is why standards such a RSA exist. As for the differences between the implementations, as stated above Microsoft implemented theirs differently because the could, and it worked in the context of their kernel and their programming paradigm. FreeBSD did it their way due to the architecture their kernel supported and the need for it to be as generalized as possible.

References

[1] Microsoft developer network, the cryptography api. `https://msdn.microsoft.com/en-us/library/ms867086.aspx`. Accessed 12/02/2015.

[2] Marshall Kirk McKusick, George V Neville-Neil, and Robert NM Watson. *The design and implementation of the FreeBSD operating system, Chapter 4*. Pearson Education, 2014.

[3] Marshall Kirk McKusick, George V Neville-Neil, and Robert NM Watson. *The design and implementation of the FreeBSD operating system, Chapter 5*. Pearson Education, 2014.

[4] Marshall Kirk McKusick, George V Neville-Neil, and Robert NM Watson. *The design and implementation of the FreeBSD operating system, Chapter 6*. Pearson Education, 2014.

[5] Mark E Russinovich, David A Solomon, and Alex Ionescu. *Windows Internal, Part 2, Chapter 10*. Pearson Education, 2012.

[6] Mark E Russinovich, David A Solomon, and Alex Ionescu. *Windows Internals, Part 1, Chapter 5*. Pearson Education, 2012.