Writing Topic 2: I/O

Behnam Saeedi
CS444: Operating systems II

Spring 2017

Abstract

In this document we are going to have a look at three operating systems: Linux, Windows and FreeBSD and compare and analyze them based on the data structures, schedulers, algorithms, cryptography and I/O scheduling. We will start by talking about linux as a control and then provide an explanation of many features integrated in Linux Kernel by providing sources. This document will atempts to discuss some of the advantages and disadvantages they have with respect to one another.

CONTENTS

1 LINUX

In this assignment we are going to have a closer look on the I/O aspect of three common operating systems. These Operating systems are Linux, Windows and FreeBSD. In the first section we will cover the Linux in order to hold it a control to compare the other two operating system s to it. In this section we will talk about data structures, I/O scheduling, Cryptography and implementations.

1.1 data structures

According to Linux documentation the Linux Kernel provides various versions of double linked list specifically purposed for I/O. Some examples of these data structures are doubly linked list, B+ tree, priority heap and many many more. [1]. Everything that requires a data structure in Linux is dependent on these few models. The thing to consider is that the Linux actually does not create a data based structure. Instead it provides the pointers needed in order to link them. The developers can create their own structs and include this pointer structure into their data containers. This has an interesting and useful side effect. Due to this method of implementation user has the capability to generate a linked list where the type of the data elements does not need to match. This means a link containing and integer could be followed by a link that contains character.

1.2 Algorithms

There are many different scheduling algorithms. These algorithms include:

- Random scheduling (RSS)
- First In, First Out (FIFO), also known as First Come First Served (FCFS)
- Last In, First Out (LIFO)
- Shortest seek first, also known as Shortest Seek / Service Time First (SSTF)
- Elevator algorithm, also known as SCAN (including its variants, C-SCAN, LOOK, and C-LOOK)
- N-Step-SCAN SCAN of N records at a time
- FSCAN, N-Step-SCAN where N equals queue size at start of the SCAN cycle
- Completely Fair Queuing (CFQ) on Linux
- Anticipatory scheduling
- Noop scheduler
- Deadline scheduler
- mClock scheduler [2]
- Budget Fair Queueing (BFQ) scheduler [3]
- Kyber [4]

1.3 I/O scheduling

Checking the Linux kernel files we can see that the algorithms which Linux supports are NOOP, deadline, and CFQ. In previous assignment we implemented the SSTF version for Linux. (it is good to note that there are 3rd party implementations of that available).

1.4 Cryptography

Linux provides one of the most extensive cryptographic APIs available. It supports a wide variety of algorithms available for memory. There is an extensive list of cypher alogrithms available: [5]

- Symmetric ciphers
- AEAD ciphers
- Message digest, including keyed message digest
- Random number generation
- User space interface

These algorithms are used in order to create the following templates available to the users: [6]

- aes: or The Advanced Encryption Standard is a symmetric block cypher
- ecb(aes): or Electronic Codebook and block mode of operations
- cmac(aes): or Cipher-based Message Authentication Code
- ccm(aes): or Counter with CBC-MAC
- rfc4106(gcm(aes)): Galois/Counter Mode (GCM) in Advanced Encryption Standard
- sha1: or Secure Hash Algorithm 1 which is crypto-hash function
- hmac(sha1): or Keyed-hash message authentication code (HMAC)

As mentioned, the Linux Kernel provides both synchronous and asynchronous operations depending on the users need.

2 WINDOWS

The next element which we are going to look into is the Windows, We are going to look into data structures used in Windows I/O, I/O scheduling, Windows Cryptography and finally a brief comparison of Windows performance stacked next to Linux.

2.1 data structures

Windows has a radically different approach to data structures than Linux. Windows has a specific data structure in place known as IRP or Input/Output Request Packets. [7]. According to windows documentation, Unlike Linux, this data structure contains two elements. a pointer to a DEVICE_OBJECT and a pointer to an IRP. The IRQs are large data files containing all of the necessary driver critical information. This seems to be an inefficient approach to data structure design.

2.2 I/O scheduling

Windows scheduler is a very complex setup that could be throughly covered in a full document of its own. In the source found, it is clear that windows uses a variation of multilevel feedback queue. [8]. As covered in earlier assignment, it works based on setting a specific value for the priority of each task described through a set of classes.[9]. This value could be set through task manager. A similar concept to this exists in Linux which is known as niceness of a program. A program can decide how much of CPU time it would like to have. The nicer a program the lower the scheduling priority becomes. The idea behind this concept is based on the fact that true concurrency does not exist. The CPU creates the illusion of concurrency by switching the process that is running and making sure that all of the processes get access to resources.

2.3 Cryptography

Windows has a service known as Cryptographic Service Providers or CSP. The task of this contains a vast library of cryptographic standard and algorithms. [10]. According to this documentation minimum requirement for the CSP assets are DLLs containing implementation the functions in CryptoSPI (a system program interface). The algorithms used are consistent of:

- **Digital signature algorithm:** In this algorithm, every type specifies only one key exchange algorithm. Then using the application's specification, provide type is selected.
- **Key BLOB formats:** Similar to last element with the exception that the tool creates a digital signature by selecting the appropriate type.
- **Digital signature format:** The Type is determined as the format of the key BLOB is used to export keys from the CSP and to import keys into a CSP.
- **Session key derivation scheme:** A session key is generated by hash which leads to selection of the type.
- **Key length:** Public and private keys are used.
- **Default modes:** Block encryption padding.

Some of the implementations include:

- PROV_RSA_FULL
- PROV_RSA_AES
- PROV_RSA_SIG
- PROV_RSA_SCHANNEL
- PROV_DSS
- PROV_DSS_DH
- PROV_DH_SCHANNEL
- PROV_FORTEZZA
- PROV_MS_EXCHANGE
- PROV_SSL

It is not as simple to compare the cryptography methods that are available. However we can definitely see some familiar types that are present in windows version such as AES and SSL.

3 FREEBSD

In this section we are going to dive into the FreeBSD kernel and have a look at the data structures, I/O scheduling, cryptography and implementations of them. Then we are going to compare those to Linux kernel and its features.

3.1 data structures

In the section bellow titled, "Kernel System Queue Synopsis" a synopsis of the system queue library is provided. [11]. This is identical to Linux approach to data structures. This meas FreeBSD too provides only a pointer for the developer to include in their own costume container providing a flexible linked list.

3.2 I/O scheduling

FreeBSD Kernel has two available schedulers. The ULE scheduler first introduced in FreeBSD 5.0 and The 4.4BSD scheduler.

- **4.4BSD scheduler:** his scheduler was originally designed for Linux kernel in order to address interactivity issues. This scheduler has a single run-queue and is variation of Round-Robin. This queue is in fact a double linked list and is shared across all the CPUs. The reason to that is to avoid complex load balance algorithms.[12][Page 2].
- **ULE scheduler:** ULE scheduler is based on the BSD scheduler and improves on some of its features in order to improve efficiency specially with single processor systems. This scheduler is disabled by default in favor of the traditional BSD scheduler. Furthermore, it is fair by default but it could be set to favor specific processes.

3.3 cryptography

The FreeBSD takes advantage of Linux OpenCrypto, crypto and cryptodev libraries. They have been working constantly to significantly reduce the overhead involved with using this API. [13]. Another supported encryption method is the OpenSSL cryptographics based on libcrypto.[14].

3.4 Kernel System Queue Synopsis

The following code could be found at FreeBSD library manual. [11]. This tool is capable of providing kernel and user-level to hardware crypto device for hash handling purposes.

```
#include <sys/queue.h>
    SLIST_CLASS_ENTRY(CLASSTYPE);
    SLIST_CLASS_HEAD(HEADNAME,        CLASSTYPE);
    SLIST_EMPTY(SLIST_HEAD *head);
    SLIST_ENTRY(TYPE);
    SLIST_FIRST(SLIST_HEAD *head);
    SLIST_FOREACH(TYPE        *var, SLIST_HEAD *head,        SLIST_ENTRY NAME);
    SLIST_FOREACH_FROM(TYPE *var, SLIST_HEAD *head, SLIST_ENTRY NAME);
    SLIST_FOREACH_FROM_SAFE(TYPE *var,        SLIST_HEAD *head, SLIST_ENTRY NAME,
        TYPE *temp_var);
    SLIST_FOREACH_SAFE(TYPE *var, SLIST_HEAD *head, SLIST_ENTRY NAME,
        TYPE *temp_var);
    SLIST_HEAD(HEADNAME, TYPE);
    SLIST_HEAD_INITIALIZER(SLIST_HEAD head);
    SLIST_INIT(SLIST_HEAD *head);
    SLIST_INSERT_AFTER(TYPE *listelm, TYPE *elm, SLIST_ENTRY NAME);
    SLIST_INSERT_HEAD(SLIST_HEAD *head, TYPE *elm, SLIST_ENTRY        NAME);
    SLIST_NEXT(TYPE *elm, SLIST_ENTRY NAME);
    SLIST_REMOVE(SLIST_HEAD *head, TYPE *elm, TYPE, SLIST_ENTRY NAME);
    SLIST_REMOVE_AFTER(TYPE *elm, SLIST_ENTRY NAME);
    SLIST_REMOVE_HEAD(SLIST_HEAD *head, SLIST_ENTRY NAME);
    SLIST_SWAP(SLIST_HEAD *head1, SLIST_HEAD *head2, SLIST_ENTRY NAME);
    STAILQ_CLASS_ENTRY(CLASSTYPE);
    STAILQ_CLASS_HEAD(HEADNAME, CLASSTYPE);
    STAILQ_CONCAT(STAILQ_HEAD *head1, STAILQ_HEAD *head2);
    STAILQ_EMPTY(STAILQ_HEAD *head);
    STAILQ_ENTRY(TYPE);
    STAILQ_FIRST(STAILQ_HEAD *head);
    STAILQ_FOREACH(TYPE *var, STAILQ_HEAD *head, STAILQ_ENTRY NAME);
    STAILQ_FOREACH_FROM(TYPE *var, STAILQ_HEAD        *head, STAILQ_ENTRY NAME);
    STAILQ_FOREACH_FROM_SAFE(TYPE *var, STAILQ_HEAD *head, STAILQ_ENTRY NAME,
```

```
    TYPE *temp_var);
STAILQ_FOREACH_SAFE(TYPE *var, STAILQ_HEAD        *head, STAILQ_ENTRY NAME,
    TYPE *temp_var);
STAILQ_HEAD(HEADNAME, TYPE);
STAILQ_HEAD_INITIALIZER(STAILQ_HEAD head);
STAILQ_INIT(STAILQ_HEAD *head);
STAILQ_INSERT_AFTER(STAILQ_HEAD *head, TYPE *listelm, TYPE        *elm,
    STAILQ_ENTRY NAME);
STAILQ_INSERT_HEAD(STAILQ_HEAD *head, TYPE        *elm, STAILQ_ENTRY NAME);
STAILQ_INSERT_TAIL(STAILQ_HEAD *head, TYPE        *elm, STAILQ_ENTRY NAME);
STAILQ_LAST(STAILQ_HEAD *head, TYPE *elm, STAILQ_ENTRY NAME);
STAILQ_NEXT(TYPE *elm, STAILQ_ENTRY NAME);
STAILQ_REMOVE(STAILQ_HEAD *head, TYPE *elm, TYPE, STAILQ_ENTRY NAME);
STAILQ_REMOVE_AFTER(STAILQ_HEAD *head, TYPE *elm, STAILQ_ENTRY NAME);
STAILQ_REMOVE_HEAD(STAILQ_HEAD *head, STAILQ_ENTRY        NAME);
STAILQ_SWAP(STAILQ_HEAD *head1, STAILQ_HEAD *head2, STAILQ_ENTRY NAME);
LIST_CLASS_ENTRY(CLASSTYPE);
LIST_CLASS_HEAD(HEADNAME, CLASSTYPE);
LIST_EMPTY(LIST_HEAD *head);
LIST_ENTRY(TYPE);
LIST_FIRST(LIST_HEAD *head);
LIST_FOREACH(TYPE *var, LIST_HEAD *head, LIST_ENTRY NAME);
LIST_FOREACH_FROM(TYPE *var, LIST_HEAD *head, LIST_ENTRY NAME);
LIST_FOREACH_FROM_SAFE(TYPE *var, LIST_HEAD *head,        LIST_ENTRY NAME,
    TYPE *temp_var);
LIST_FOREACH_SAFE(TYPE *var, LIST_HEAD *head, LIST_ENTRY NAME,
    TYPE *temp_var);
LIST_HEAD(HEADNAME, TYPE);
LIST_HEAD_INITIALIZER(LIST_HEAD head);
LIST_INIT(LIST_HEAD *head);
LIST_INSERT_AFTER(TYPE *listelm, TYPE *elm, LIST_ENTRY NAME);
LIST_INSERT_BEFORE(TYPE *listelm, TYPE *elm, LIST_ENTRY NAME);
LIST_INSERT_HEAD(LIST_HEAD        *head, TYPE *elm, LIST_ENTRY NAME);
LIST_NEXT(TYPE *elm, LIST_ENTRY NAME);
LIST_PREV(TYPE *elm, LIST_HEAD *head, TYPE, LIST_ENTRY NAME);
LIST_REMOVE(TYPE *elm, LIST_ENTRY NAME);
LIST_SWAP(LIST_HEAD *head1, LIST_HEAD *head2, TYPE, LIST_ENTRY NAME);
TAILQ_CLASS_ENTRY(CLASSTYPE);
TAILQ_CLASS_HEAD(HEADNAME,        CLASSTYPE);
TAILQ_CONCAT(TAILQ_HEAD *head1, TAILQ_HEAD        *head2,        TAILQ_ENTRY NAME);
TAILQ_EMPTY(TAILQ_HEAD *head);
TAILQ_ENTRY(TYPE);
TAILQ_FIRST(TAILQ_HEAD *head);
TAILQ_FOREACH(TYPE        *var, TAILQ_HEAD *head,        TAILQ_ENTRY NAME);
TAILQ_FOREACH_FROM(TYPE *var, TAILQ_HEAD *head, TAILQ_ENTRY NAME);
TAILQ_FOREACH_FROM_SAFE(TYPE *var,        TAILQ_HEAD *head, TAILQ_ENTRY NAME,
    TYPE *temp_var);
TAILQ_FOREACH_REVERSE(TYPE        *var, TAILQ_HEAD *head,        HEADNAME,
    TAILQ_ENTRY NAME);
TAILQ_FOREACH_REVERSE_FROM(TYPE *var, TAILQ_HEAD *head, HEADNAME,
    TAILQ_ENTRY NAME);
TAILQ_FOREACH_REVERSE_FROM_SAFE(TYPE *var,        TAILQ_HEAD *head, HEADNAME,
    TAILQ_ENTRY NAME, TYPE        *temp_var);
TAILQ_FOREACH_REVERSE_SAFE(TYPE *var, TAILQ_HEAD *head, HEADNAME,
```

```
    TAILQ_ENTRY NAME, TYPE        *temp_var);
TAILQ_FOREACH_SAFE(TYPE *var, TAILQ_HEAD *head, TAILQ_ENTRY NAME,
    TYPE *temp_var);
TAILQ_HEAD(HEADNAME, TYPE);
TAILQ_HEAD_INITIALIZER(TAILQ_HEAD head);
TAILQ_INIT(TAILQ_HEAD *head);
TAILQ_INSERT_AFTER(TAILQ_HEAD *head, TYPE *listelm, TYPE *elm,
    TAILQ_ENTRY NAME);
TAILQ_INSERT_BEFORE(TYPE *listelm,        TYPE *elm, TAILQ_ENTRY NAME);
TAILQ_INSERT_HEAD(TAILQ_HEAD *head, TYPE *elm, TAILQ_ENTRY        NAME);
TAILQ_INSERT_TAIL(TAILQ_HEAD *head, TYPE *elm, TAILQ_ENTRY        NAME);
TAILQ_LAST(TAILQ_HEAD *head, HEADNAME);
TAILQ_NEXT(TYPE *elm, TAILQ_ENTRY NAME);
TAILQ_PREV(TYPE *elm, HEADNAME, TAILQ_ENTRY NAME);
TAILQ_REMOVE(TAILQ_HEAD *head, TYPE *elm, TAILQ_ENTRY NAME);
TAILQ_SWAP(TAILQ_HEAD *head1, TAILQ_HEAD *head2, TYPE, TAILQ_ENTRY        NAME);
```

REFERENCES

[1] "Data structures in the linux kernel," https://0xax.gitbooks.io/linux-insides/content/DataStructures/, accessed: 2017-05-19.
[2] "Vmware academic program," https://labs.vmware.com/academic/publications/mclock, accessed: 2017-05-19.
[3] "Budget fair queueing (bfq) storage-i/o scheduler," http://algo.ing.unimo.it/people/paolo/disk_sched/, accessed: 2017-05-19.
[4] "blk-mq: Kyber multiqueue i/o scheduler," https://lwn.net/Articles/720071/, accessed: 2017-05-19.
[5] "Cipher algorithm types," https://www.kernel.org/doc/html/v4.10/crypto/architecture.html#cipher-algorithm-types, accessed: 2017-05-19.
[6] "Ciphers and templates," https://www.kernel.org/doc/html/v4.10/crypto/architecture.html#cipher-algorithm-types, accessed: 2017-05-19.
[7] "I/o request packets," https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/i-o-request-packets, accessed: 2017-05-19.
[8] "The microsoft press store," https://www.microsoftpressstore.com/articles/article.aspx?p=2201309&seqNum=3, accessed: 2017-05-19.
[9] "Scheduling," https://msdn.microsoft.com/en-us/library/windows/desktop/ms685096(v=vs.85).aspx, accessed: 2017-05-19.
[10] "Cryptographic service providers," https://msdn.microsoft.com/en-us/library/windows/desktop/aa380245(v=vs.85).aspx, accessed: 2017-05-19.
[11] "Freebsd library functions manual," https://www.freebsd.org/cgi/man.cgi?query=queue&sektion=3, accessed: 2017-05-19.
[12] "Bfs by con kolivas," https://vellvisher.github.io/papers_reports/doc/BFS_FreeBSD.pdf, accessed: 2017-05-01.
[13] "Hardware crypto support status," https://www.freebsd.org/news/status/report-2002-11-2002-12.html, accessed: 2017-05-19.
[14] "Freebsd man pages," https://www.freebsd.org/cgi/man.cgi?crypto(3), accessed: 2017-05-19.