Operating Systems Notes

Contents

1 OS Architecture

Different architectures dictate the way that an operating system may conduct its business.

As different architecture platforms have different instruction sets and ASPLOS features, it determines what are viable methods for accomplishing tasks such as *memory protection* and *control interrupts*.

The OS acts as an intermediary between software and hardware. It allows mediation of access and gives developers a consistent API to access hardware and low-level operations. This abstraction is achieved via *traps* and *exceptions*. Similarly, devices can gain attention via the use of *interrupts*.

An OS is intended to be a good balance between *user needs* and *system needs*.

A user wants an OS to be *fast, reliable and safe*; an OS developer wants it to be *efficient, error-free, and easy to maintain and implement*.

The design of an operating systems makes a distinction between two underlying principles:

Policy, which is *what* will be done; **Mechanism**, which his *how* to do it.

1.1 Structure of an OS

A capable operating system is home to a multitude of components: memory management, I/O, file systems, command interpreters and so on. There are multiple ways to link to structure how all these components are connected to each other.

1.1.1 Monolithic

An early type of design for an OS, where the *entire* operating system would function in *kernel mode* (see the *Privileged Instructions* section). However, this design carried many issues with maintainability and reliability, and was hard to make sense of.

1.1.2 Layering

As opposed to monolithic, this method implemented the OS as a *set of layers* instead, where a layer could present itself to the corresponding layer above.

5	**Job Managers**	Execute a user's programs
4	**Device Managers**	Handle devices, provide buffering
3	**Console Manager**	Provide virtual consoles
2	**Page Manager**	Implement virtual memories for each process
1	**Kernel**	Implement a virtual processor for each process
0	**Hardware**	

Figure 1: Dijkstra's THE System

Layering provides a *hierarchical* structure but provides performance issues as the overhead increases with the addition of each layer. Systems can be modelled as layers, but tend not to be implemented in such a way.

1.1.3 Hardware Abstraction Layer

More common in modern OSs. The idea is to have a *core OS*, components such as the file system and scheduler, which don't depend on hardware. Then, you have the *hardware abstraction layer* which specifically deals with the hardware through *drivers* and *assembly routines*.

This allows for more portability, as the layer deals with translation between the core and the specifics of the hardware; developers don't have to worry about specific traits of some hardware and can instead deal with specifics of the core.

1.1.4 Microkernel

This concept aims to minimise what the *kernel* is responsible for, and have the rest dealt with by processes at the user level. The Windows Kernel (NT) is a good example.

This isolation allowed isolation between the components, but provides poor performance as the number of user/kernel crossings greatly increases.

1.1.5 Loadable Kernel Modules

Utilised by the *NIXes (Linux, BSD, other UNIXes, Solaris), this concept has core services within the kernel code itself, and then other modules can be loaded in *dynamically*.

This style avoids some of the issues that come with layering, as it can call any other module. As the loading is dynamic, there is no need for reboots.

1.2 Privileged Instructions

These instructions are restricted to use by the OS *only*, and includes direct access to *I/O devices* and *memory state management*.

This is achieved by the implementation of two *modes of operation*; *user* and *kernel* modes.

Privileged instructions can only be executed in kernel mode.

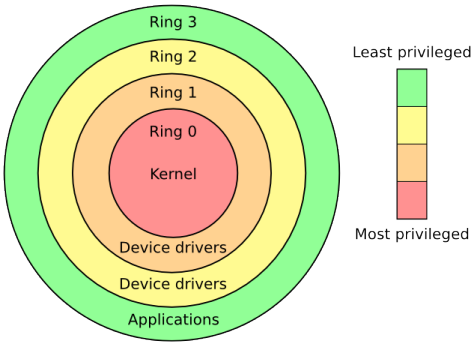

Figure 2: x86 Architecture Levels of Privilege
Courtesy of Hertzsprung of English Wikipedia

1.2.1 System Calls

If code running from within user mode tries to execute a privileged instruction, it will trigger the *illegal execution trap*, which allows for such code to gain access to privileged instructions and resources. These are called *system calls*.

An OS will define a set of system calls that makes it possible for such an interaction to take place.

Caller: The user mode process invoking the system call
Callee: The OS handling the system code

- The caller places arguments, especially the *type* of system call, in a specified location
- The callee saves the caller's state
- The callee verifies the arguments
- If valid, the code is run
- When the system call has been satisfied, the program counter is set to the return address
- Execution is returned to user mode

Figure 3: A rough breakdown of a system call

1.3 Exception Handling

A *trap* is a synchronised, intended transition that is initiated by the OS. The OS takes control as the result of a system call.

An *exception* is also synchronised, however it is an *unexpected* problem with some instruction.

An *interrupt* is *a*synchronous on the other hand, and is caused by some external device.

1.4 Processes

A *process* is a program that is being executed.

A process must have a few key aspects for it to work:

Address Space

Contains the *instructions* of the running program, and the *data* for running the program.

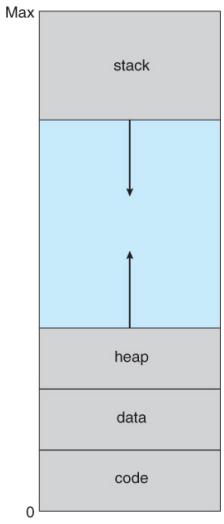

Figure 4: An idealised address space given for a process

CPU State

Contains the *program counter* which indicates the next instruction, the *stack pointer* and other register values.

OS Resources

Contains items that the OS gives the process access to, such as files, network connections and so on.

A process can spawn another process, which are respectively called the *parent* and *child* of each other. When the parent spawns its child, it can choose to either *wait for the child to terminate*, or *run in parallel with the child*.

Spawning of a child makes use of the `fork()` process. It returns `0` to the child, and its process ID to the parent.

1.5 PCB: Process Control Block

The operating system needs a way to keep a track of all processes that are running.

A data structure, called the *process control block*, or *process descriptor*, aims to do this.

This name space includes all the *process IDs* of the processes that are currently present. A process ID is a *unique integer*.

A process is called by the `fork()` function, and can be *killed* (`kill()`), *paused* (`wait()`), and *limited* (`nice()`).

The PCB keeps: the process's *execution state* when the process isn't active; the parent's PID; program counter; and so on. Linux's PCB has over 95 fields for *each* entry.

Ready	Waiting to be assigned by the CPU
Running	The process that is currently running in the CPU
Waiting	Cannot progress until some event happens, e.g. *I/O completion*

Figure 5: Possible execution state of a process

When a process is returned to the *running* state, the values of registers are transferred from the PCB to hardware registers. This is called a *context switch*. These occur many, many times per second. The decision of which process to run next is *scheduling*.

1.6 State Queues

A set of queues contains a queue for a process within each respective state. These are implemented as a set of *linked lists*.

1.7 Sequential Process

The most simple style of process.

A *sequential process* is given some *address space* and a *thread of execution*. The process will then be executed.

1.8 Memory Management Unit (MMU)

Effectively is a hash function from logical address to physical address.

A MMU prevents the need for swapped out processes to be swapped back into the same physical addresses.

Swapping is not typically supported on mobile devices, more likely to to overwrite least used data.

1.9 Partitioning

Main memory is usually broken up into two partitions; The OS and user process.

Each process is contained within a single contiguous section on memory.

Reallocation registers are used to protect users processes from one another and from changing the OS code.

Some old techniques include:

- Fixed Partitions - simple but causes fragmentation often
- Variable Partitions - no internal fragmentation, but can leave holes in the physical memory

Dynamic Storage-Allocation is possible using First-fit, Best-fit and Worst-fit in terms of hole filling.

2 Threading

Instead of using multiple processes for concurrency, we can use threads, which share the same address space and OS resources. This smaller overhead usually results in threads being more performant than the same number of separate processes.

With threads, we want the same program code, data, privileges and resources (open files, sockets...) but we need multiple execution states (each thread has it's own registers and stack)

From the perspective of the parent process to a set of threads, the *heap* is shared, and each thread has it's own *stack pointer* and *PC*.

2.1 Kernel Threading

The kernel presents it's own threading library, with the kernel managing both the process and threads memory. Kernel threads are more performant than just processes, but because they are handled through **system calls** there's still a sizeable overhead.

2.2 User-Level Threading

User threads are built completely transparently to the underlying OS, which only sees the parent processes. User threads can be very performant, though if one thread is blocked due to IO etc., the entire process will also be blocked as there is no way for the kernel to schedule other threads as it isn't aware of them.

User threads can be 10 to 100 times faster than kernel threads.

A combination of kernel threading and user-space threading can be used to get the best performance.

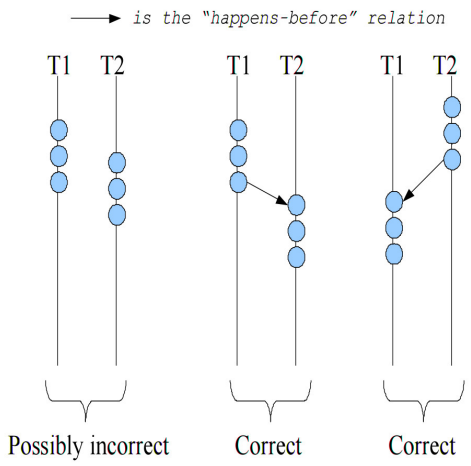

Figure 6: Concurrency Dependencies

3 Mutual Exclusion & Locking

A *critical section* is a sequence of code that may result in incorrect or undefined behaviour if executed simultaneously or pre-empted. Similarly, race conditions occur when the order of execution is unknown and behaviour can be unpredictable.

Mutual Exclusion (MutEx) and locking prevent these problems.

To be safe and operate correctly, a critical section must satisfy the following requirements:

Mutual Exclusion At most one thread is in the critical section

Progress If a thread is outside the critical section, it cannot prevent another from entering

Bounded Waiting & No Starvation If a thread is waiting to enter the critical section, it is guaranteed to eventually do so

Performance The overhead of entering and exiting the critical section is small relative to the runtime of the section.

3.1 Deadlocks

We obviously want to avoid deadlocking (and livelocking). For a deadlock to occur, the following four conditions must exist:

- Mutual Exclusion
- Hold and Wait (Prevent a process that needs two or more locks from holding one and waiting for the other indefinitely - see the Philosopher's Spaghetti problem)
- No Preemption i.e. we can't stop a process/thread from running that may be causing a lockup
- Circular waiting e.g. a process busy waiting on a lock it won't ever get

We can see if a deadlock is possible if there is an *irreducible* cycle in the dependency/resource graph for a set of processes or threads.

3.2 Peterson's Algorithm

```
flag[i] = True;
turn = 1-i;

while (flag[1-i] && turn == 1-i); // spin

/* critical section code */

flag[i] = False;
```

Avoids both *deadlock* and *livelock* for two threads, using `i` and `i-1` as signals. It works but can be tricky to implement.

3.3 Spinlocks

A **spinlock** is a locking primitive, used to build more complex locking mechanisms. The name comes from the behaviour; It `spins' on a condition until it is satisfied, so will acquire the lock as soon as it is available.

```
// do not have lock
while(some_condition);
// have lock
```

Acquiring and releasing locks *must* be an atomic operation, so no context switching occurs during acquisition which could lead to a crash or undefined state. **Test and Set** is an atomic instruction we can use to achieve this.

```
struct spinlock{
    int held;
};

struct spinlock lock;
lock.held = 0;

int test_and_set(int* flag){
    int old = *flag;
    flag = 1;
    return old;
}

void acquire(struct lock* lock)
{
    /* This is the `spinning' */
    while(test_and_set(&lock->held));
}

void release(struct lock* lock)
{
    lock->held = 0;
}
```

Spinlocks are simple to implement, but can be slow and *block* whilst waiting for the lock, wasting CPU time.

3.4 Semaphores

Similar to a spinlock, having the two operations `wait(semaphore)` and `signal(semaphore)`, sometimes given as `P(semaphore)` and `V(semaphore)`. Semaphores can use integer signals, or a boolean - the boolean semaphore has the same behaviour as a lock.

```
void wait(semaphore s)
{
    while(s <= 0); // Busy wait
    s--;
}

void signal(semaphore s)
{
    s++;
}
```

This simple implementation uses *busy waiting*. We can use a wait queue instead, placing ourselves onto the queue and yielding until the semaphore is available.

```
void wait(semaphore* s)
{
    s->value--;
    if(s->value < 0){
        enqueue_process();
        /* Add self to the wait queue and block/sleep */
    }
}

void signal(semaphore* s)
{
    s->value++;
    if(s->value <= 0){
        dequeue_process();
        /* Remove a process from the wait queue and wake it */
    }
}
```

3.4.1 Bounded Buffer Problem

In a **producer and consumer** model, we can have many threads wishing to consume some data, and many that produce it. The handover of data can be done with a *bounded buffer*

We can use three semaphores to ensure safety around this bounded buffer:

mutex	1	Mutual exclusion on shared data (head & tail...)
empty	n	Number of empty / available slots in the buffer. Initially all available.
full	0	Number of taken slots in the buffer. Initially none are taken.

```
producer:
    wait(empty)
    wait(mutex)
        add item to buffer
    signal(mutex)
    signal(full)
```

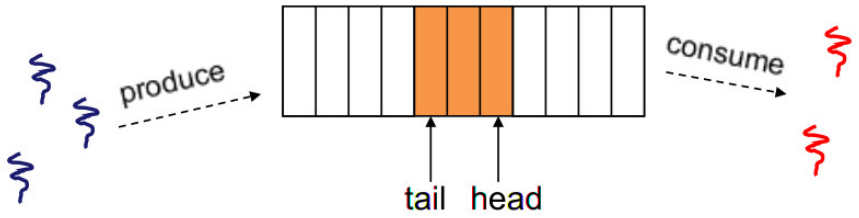

Figure 7: Producer-Consumer model

```
consumer:
    wait(full)
    wait(mutex)
        remove item from buffer
    signal(mutex)
    signal(empty)
```

3.4.2 Pub Sub

The constraints on **publishers** (aka writers) and **subscribers** (aka readers) are different:

1. We *can* allow multiple subscribers to concurrently access
2. We *cannot* allow publishers and subscribers to concurrently access
3. We *cannot* have more than one publisher concurrently writing

This is because subscribers make the promise to not mutate the data, while publishers do not.

semaphore: mutex	1	Mutual exclusion on shared data
semaphore: writing	1	Lock on mutation/writing to the data
integer: read_count	0	The number of subscribers currently reading

```
writer:
    wait(writing)
        perform writes // Satisfies requirement 3
    signal(writing)

reader:
    wait(mutex)
        reader_count++;
        if (reader_count == 1) wait(writing); // Satisfy requirement 2
    signal(mutex)

    do reading

    wait(mutex)
        reader_count--;
     if (reader_count == 0) signal(writing); // Release lock on writers if no further readers
    signal(mutex)
```

3.5 Condition Variables

Has similar operations to Semaphores: `wait()` and `signal()`.

Wait Wait until another thread has signalled and released the lock

Signal Wake a thread from the wait queue

Signals aren't remembered if there are no threads in the wait queue like they are with semaphores.

3.5.1 Bounded Buffer

lock: mutex	1	Mutual exclusion on shared data (head & tail...)
condition: freeslot	n	There is at least one slot free
condition: fullslot	0	There is at least one slot taken

```
producer:
    lock(mutex)
    if [no slots available] wait(freeslot);
        Add item...
    signal(fullslot)
    unlock(mutex)

consumer:
    lock(mutex)
    if [no slots have data] wait(fullslot);
        Pop item from shared buffer
    signal(freeslot)
    unlock(mutex)
    Use item...
```

3.6 Possible Bugs

Most locking mechanisms are prone to bugs; They're shared data structures, and there's never a guarantee a lock will ever be released or acquired when it should be.

The lock is also not strictly associated with the data it protects; Have we got the right lock? Should we have more than one lock? Programming language structures such as classes with built in locking mechanisms go some way to protect against these errors.

3.7 Monitors

A higher level programming language structure, requires some notion of objects (so would be tricky in C, much easier in C++). Each method in the class automatically acquires a lock on entry, releasing it on exit. This is transparent to the implementer of the API exposed by the class. Safeties that come from using classes, such as protected/private methods and structures help protect data.

Implementation of a monitor class in C++. `invariant` is asserted over the life of a class, throwing an error if the condition is broken. Similarly `precondition` asserts a condition before method execution is allowed.

```
class Account {
  private lock myLock;
```

```
  private int balance := 0
  invariant balance >= 0

  public method boolean withdraw(int amount)
     precondition amount >= 0
  {
    myLock.acquire();
    try:
      if balance < amount then return false
      else { balance := balance - amount ; return true }
    finally:
      myLock.release();
  }

  public method deposit(int amount)
     precondition amount >= 0
  {
    myLock.acquire();
    try:
      balance := balance + amount
    finally:
      myLock.release();
  }
}
```

Hoiked from wikipedia, https://en.m.wikipedia.org/wiki/Monitor_(synchronization)

4 Secondary Storage

4.1 Disk Performance

- Seek : moving the disk arm to the correct cylinder
- Rotation(latency): waiting for the sector to rotate under the head
- Transfer: transferring data from surface to disk controller

Seeks are very expensive, so the OS tries to schedule disk requests that are queued waiting for the disk

- FCFS - okay when load is low
- Shortest Seek Time First (SSTF) - minimizes arm movement, unfairly picks middle blocks
- SCAN(elevator algorithm) - service one direction till done, then reverse. Skews wait time non-uniformly
- C-SCAN - like SCAN, but only goes in one direction (typewriter)
- C-LOOK - similar to C-SCAN, arm only goes as far as final request in each direction

4.2 Paging

Paging is a *memory management scheme* that an operating system utilises to move information from secondary memory to main memory. A *page* is a fixed-size block that resides in secondary storage.

This scheme is found in many modern operating systems as it allows for programs to exceed the size of physically available memory.

Pages are stored in the *page table* which is stored in main memory. An address is generated by the CPU which states the *page number* which acts as a *base index* of the page table, and offset that provides the physical address when combined with the page number.

With a pure page table, we need *two* accesses to retrieve a page, once to the table to get the location of the page, and then to get the actual contents of the page. We can implement a buffer however to speed this up, the *translation look-aside buffer*, or *associative memory*.

The buffer stores identifiers that uniquely identify each process, but keeps the number of entries quite low, usually with only 64 to 1024 entries.

If a request is made for some page but is no in the buffer, referred to as a *miss*, the value will be loaded in for next time.

Physical Address Same as a physical memory address in a simpler page-less system

Logical Address Is the concatenation of a Page Number and Page Offset, and is the addressing scheme exposed to programs.

Page Table Holds the translations from a *page number* to the base address (starting memory address) of a page. Each process has it's own page table.

Page Offset How far into a page the word of memory we're addressing is.

Page Frame The divisions of physical memory that can hold a page; A page frame is the same size as a page, but may or may not contain a (valid) page.

PTBR Page Table Base Register - holds the physical address of the start of the page table

PTLR Page Table Length Register

TLB Translation Look-aside Buffer - effectively a fast cache of mappings from Logical to Physical addresses

The size of a page is usually OS specific, sometimes architecture specific. Sizes in the range 4KiB to 8MiB or more are not unusual; Very small pages, however, are undesirable as the page table itself takes memory to keep track of all pages.

Pages can be shared between processes, in a similar way to shared memory amongst threads. The mechanism for this is simple, as two processes sharing a page will simply have two different (though not necessarily different) logical addresses resolve to the same physical address. Standard concurrency problems can occur.

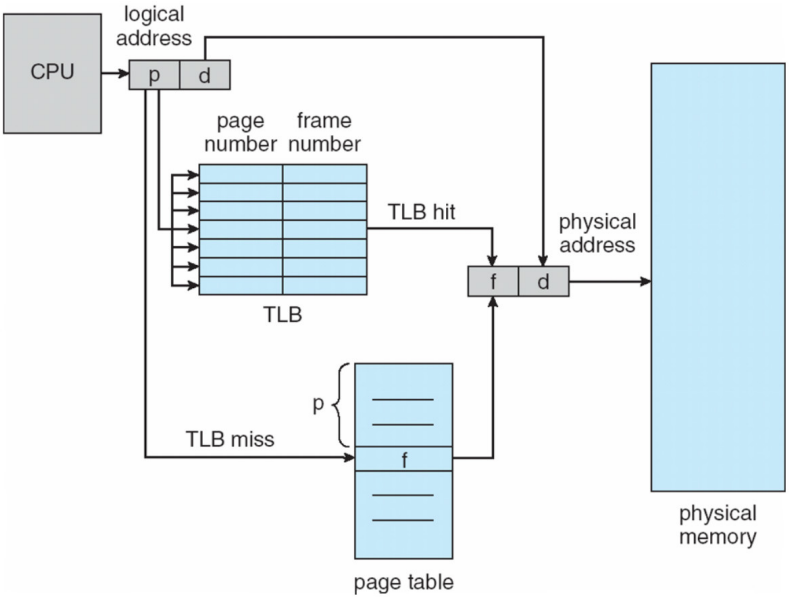

Figure 8: Paging datapath with a TLB

Note that the lookup concurrently runs through the TLB and page table; This is a race but the TLB is significantly faster, so if a hit occurs the Page Table lookup is cancelled.

4.3 Memory Protection Within Pages

Protection is implemented, to preserve the applicable level of access, such as read-only.

A page also has a *valid/invalid* bit. It states if the page is within the *logical* address space of some process. If *valid*, it is, otherwise it is *invalid*.

4.4 Benefits of Page Tables

- There is memory protection between processes as each address space is unique to the process
- Common libraries have a shared copy in physical memory instead of one-per-process, which is faster than making a system call
- Different entries per process, so two different processes get different access rights.

4.5 Paging Table Size

If we were to implement a pure page table, the structure can be very large which would make it rather inefficient.

However, we can use other methods to keep this size down.

4.5.1 Hierarchical Paging

Breaks the logical address space into *multiple page tables*.

The page number portion of the address would then be split further, into another page table and offset value.

This quickly gets silly. A page number in a three level paging system for a 64 bit address space is as follows:

```
| 2nd Outer Page | Outer Page | Inner Page | Offset |
|----------------|------------|------------|--------|
| 63          32 | 31      22 | 21      12 | 11   0 |
```

4.5.2 Hashed Page Table

Where the page number is hashed into a page table, where each element has the virtual page number,r value of the page frame and a pointer to the next element.

This reduces the amount of entries as each entry in a hashed page table refers to *multiple pages* rather than one page.

4.5.3 Inverted Page Table

Rather than each process having a page table, we track all physical pages.

There is one entry in the table for each real page of memory, which contains the virtual address of the page stored in that real location, and information about the process that owns that page.

This reduces the amount of space that is required to store the table but it takes longer to search the table when a page is referenced.

5 Memory Management

Programs must be brought from disk into memory and then placed into a process.

The CPU can only access data from memory, not disk.

Register access takes at most 1 CPU cycle, but Main Memory can take many cycles, causing a *Stall*.

5.1 Base and Limit Registers

A set of *base* and *limit registers* define the logical address space. The CPU must check every memory access is valid between the base and the limit for that user. Failure causes a trap in the OS, and usually results in a program being killed (e.g. a *Segmentation Fault*)

5.2 Virtual Address Space

Logical/Virtual addresses are independent of physical memory.

Hardware translates virtual addresses into physical ones.

Logical/Virtual addresses a process can reference is called the address space.

6 Virtual Memory

6.1 Page Fault

If there is a reference to a page that isn't stored in memory, then there will be a trap to the OS, a page fault.

The OS then finds a free frame, swaps page into frame via a scheduled disk operation then resets the tables for validity.

Pages are only brought into memory when referenced, this is called demand paging.

6.2 Page Replacement

There are a couple ways to pick which page to replace.

- Pick a page that won't be needed any time soon
- Pick a page that hasn't been used in a while

The goal of all of this to reduce the fault rate by selecting the best victim page to remove. Best being one that will never be touched again.

Belady's proof: "evicting the page that won't be used for the longest period of time minimizes page fault rate"

Thrashing is when the system spends most of its time serving page faults, and little time doing useful things.

This could mean that there is not enough memory or that the memory is over-committed.

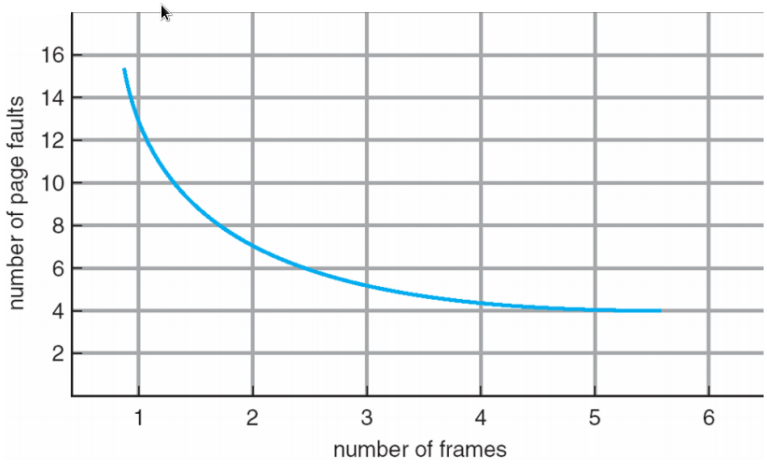

Figure 9: Page Faults vs Number of Frames

6.2.1 Replacement Algorithms

- First-In-First-Out
- Belady's Optimal Algorithm - Replace frame that will not be used for the longest time. (Not possible as we can't see into the future)
- Least Recently Used(LRU)
- Replace a page that is 'old enough' - logically, place all pages in a circle and rotate round

FIFO and LRU can each be implemented locally or globally.

Local :- Each process has a limited number of frames and operates within them.

Global :- the victim is chosen from all page frames in the system, regardless of owner:

6.3 Segmentation

Memories are usually split into distinct segments by programs -- the stack, heap, data section and code section. Segmentation provides hardware support for enforcing these distinct memory segments.

Segmentation is another memory model, that we could contrast with paging. Under segmentation, a virtual address is `segment number` and `offset`. Segmentation allows for the sharing of segments between threads and processes in the same way Paging allows shared pages.

Segments are managed with a *Segment Table* that holds base and limit paris, indexed by segment number. Segmentation gives rise to the Segmentation Fault, everyone's favourite C crash. In modern systems, segmentation is rarelt used alone, and is generally used in conjunction with paging.

Segment Registers are a feature on some more CISC processors, like Intel x86 and hold base pointers to segments. The x86 segment registers are **SS** stack segment, **CS** code segment, **DS** data segment, **ES** extra segment and the additional **FS** & **GS** generic segments.

7 File Systems

The OS acts as an intermediary between the FS and an Disk. It should hide hardware specifics and allocate disk blocks for the FS.

The file system is responsible for the organisation of files.

For example, Windows' file system NTFS relies on the OS making distinct difference between what disk it is reading from, such as the hard disk or a connected USB thumb drive: `C:`, `D:`, etc.

On the other hand, Linux makes no distinction and instead maintains a sole *root*, `/`, where everything is seen as a 'file', such as a hard disk (`/dev/sda`), or a media file you may have (`/home/user/swedgefulBasslines.mp3`).

A file is a *collection* of data; its *contents*, *size*, *owner*, and so on.

7.1 Access Methods

There are a few types of *access methods* that an FS can utilise:

- **Sequential Access** - read byte by byte
- **Direct Access** - random access given to block/byte
- **Record Access** - file is array of records
- **Indexed Access** - one file contains an index of a record in another file

7.2 Directories

Directories, which can also be referred to as folders, are *nodes* of the file system tree.

A directory contains a unordered list of all files and their attributes within itself. Sorting is usually done by some other utility, such as `ls`.

One way to find files within these directories, is via *path name translation*.

A file system finds a file when given some *path*, such as `/home/proper/legit/filepath/ooooo.sh` by walking through the path, directory by directory. At the root, it would first look for `home`, get its location, then look for `proper`, get its location, and so on.

7.3 File Protection

The file system is responsible for checking of permissions that users have for some given file.

Depending on what the user wants to do to the file, the file system needs to check *which* kind of way it wants to access it, e.g. *read*, *write* or *execute* it. Then, it must check has the rights to perform said action.

This is represented via the *access control matrix*, that lists what users have what access to what files.

	/home/hiroshi	/home/boris
Hiroshi	rw	-
Boris	r	rw
Root	rw	rw

Figure 10: An Access Control Matrix

Access control matrices are stored as *lists*, which are easier to manage, but grow in size depending on the number of users, or collection of users (*groups*).

7.4 UNIX inodes

Stands for index node. An inode contains a file's metadata and the location of it's first blocks on disk. An **indirect block** is a block that itself contains pointers to further blocks; With this an inode can represent larger files. Similarly a **double indirect** block contains pointers to indirect blocks that then pointing to data blocks.

So for a filesystem with 512B blocks, and inodes with 12 blocks, 1 indirect and one double indirect block, the maximum filesize an inode can represent is $12 \times 512B + 512 \times 512B + 512^2 \times 512B = 128.3MiB$

7.5 Organising Blocks

A file on a disk is stored as a series of *linked blocks*.