Operating Systems Notes

1 OSs and Architectures

Different architectures dictates the way that an operating system must conduct its business.

As different architecture platforms have different instruction sets, 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 to mediate access and gives developers a nicer way of completing lower-level actions. 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 overhead increases at each layer. Systems can be modelled as layers, but tend not to be implemented in such a way in reality.

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.

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 Linux, 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.

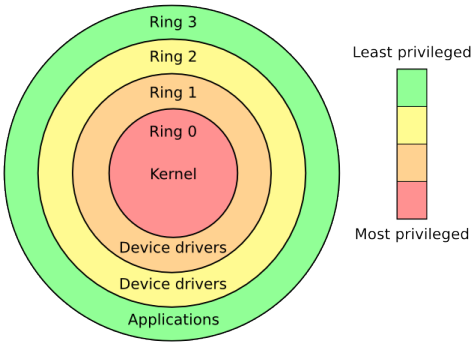

Figure 2: x86 Architecture Levels of Privilege
Courtesy of Hertzsprung of English Wikipedia

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.

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 them to work:

Address Space

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

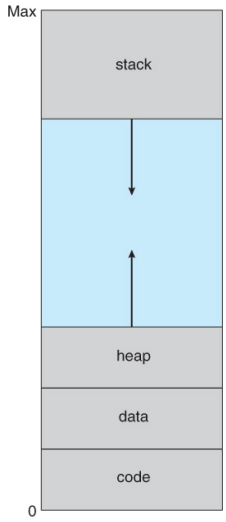

Figure 4: An idealised address space given for a process

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.

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' *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.

1.6 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.

2 Memory Management

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

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

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

2.1 Base and Limit Registers

A set of *base* and *limit registers* define the logical adress space. the CPU must chech every memory access is valid between the base and the limit for that user. Failure causes a trap to the OS monitor

2.2 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.

2.3 Memory Managment Unit (MMU)

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

a MMU prevents the need for swapped out process to be swapped back into the same physical addresses.

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

2.4 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 canging 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-fir and Worst-fit in terms of hole filling.

3 Virtual Memory

Paged virtual memory allows larger logical address space than physical memory. Not all pages of the address space need to be in memory. All needed pages are transfered to free page frames. If there are no free frames, then we evict one.

3.1 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 ten finds a free frame, swaps page into frame via a sheduled disk operation then resets the tables for validity.

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

3.2 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 oor that the memory is over-commited

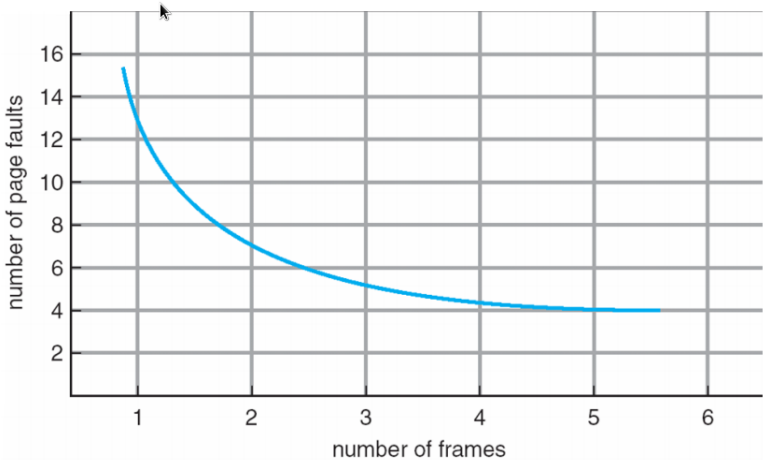

Figure 5: Page Faults vs #Frames

3.2.1 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 cirlce and rotate round

FIFO and LRU can each be implimented locally or globaly.

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

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

4 MutEx

A *critical section* is a sequence of code that may result in incorrect or undefined behaviour if executed simultaneously or preempted. 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 Exlusion 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.

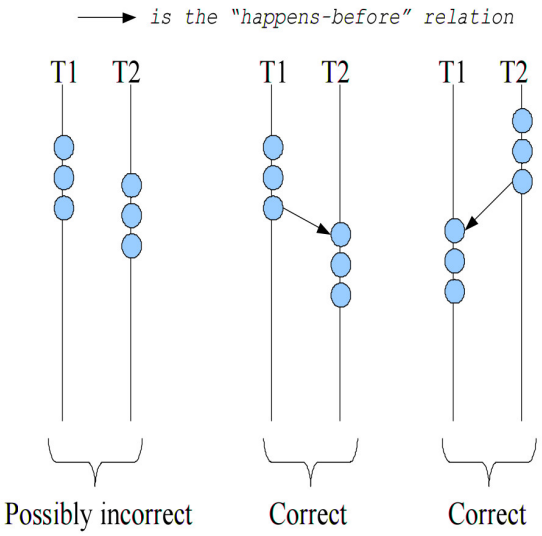

Figure 6: Concurrency Dependencies

4.1 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.

4.2 Spinlocks

A **spinlock** is a locking primative, 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 contect switching occurs during acquisition which could lead to a crash or undefined state. **Test and Set** is an atomic instruction we can use to acheive 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.

4.3 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 */
    }
}
```

4.3.1 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*

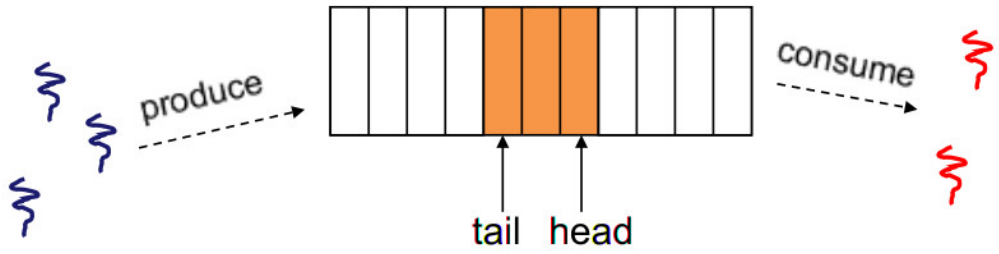

Figure 7: Producer-Consumer model

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

mutex	1	Mutual exlusion 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)

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

4.3.2 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 exlusion 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)
```

4.4 Condition Variables

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

Wait Wait until another thread has signaled 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.

4.4.1 Bounded Buffer

lock: mutex	1	Mutual exlusion 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...
```

4.5 Possible Bugs

Most locking mechanisms are prone to bugs; They're shared data structures, and there's never a guarantee a logck 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.

4.6 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)