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*.

1.1 OS Structure

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*.

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.

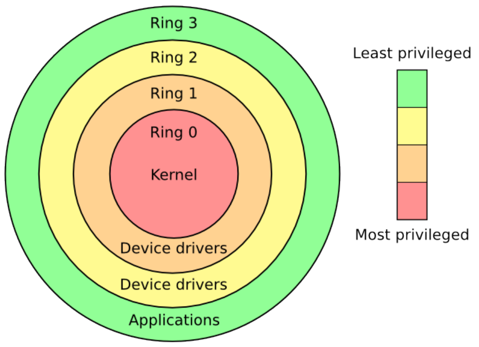

Figure 1: x86 Architecture Levels of Privilege
Courtesy of Hertzsprung of English Wikipedia

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 2: 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.

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.

Realocation 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 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.

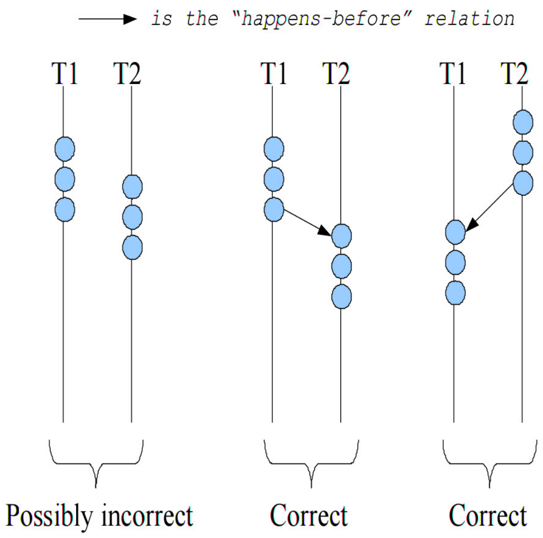

Figure 3: Concurrency Dependencies

3.1 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.

3.2 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.

3.3 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 */
    }
}
```

3.3.1 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*

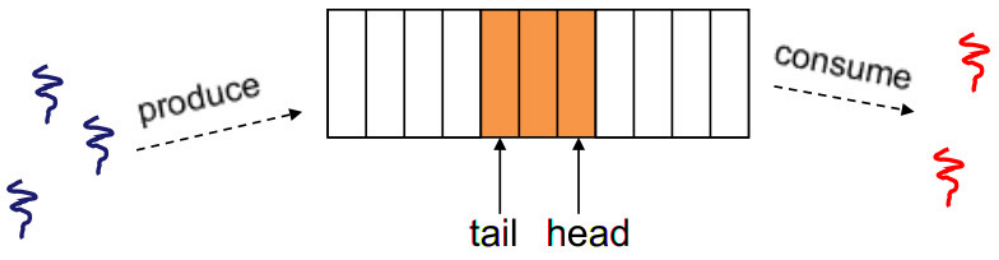

Figure 4: Producer-Consumer model

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)
```

3.3.2 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)
```