Operating Systems

Jacob Denson

December 16, 2015

Table Of Contents

Chapter 1

Introduction

Managing the hardware of a computer is a fragile task, not for the faint of heart. In computer science, we desire abstraction and modulization, to simplify tasks and reduce the load on our brain, so we are able to attack more complicated and interesting issues. An **operating system** is a program which enables this abstraction. It manages a computer's hardware, acting as an intermediary between the core resources of the computer and higher level programs, enabling a simpler interafce for the computer. Specifically resources are managed in the **time** a process takes to use a resource, and the **space** of a resource, management of units of the same resource. It is up to the operating system to enable **Time Sharing**, **CPU Scheduling**, **Memory Management**, and **Communication**.

Software can be grouped into two categories. **Systems Software** is the software operating on computer hardware, explicitly managing the physical resources of the computer. If your software requires detailed knowledge of the specific system you are using (memory endian, etc), it is probably systems software. Of course, all operating system software is systems software, but UIs and Utility software (like a virus scanner) control computer hardware but are not often identified as part of the operating system. Systems software provides a platform for running user-opearted software, known as **Applications Software**. This software is used concientiously by the user, unlike systems software, which definitely aids the user in using the computer, but runs in the background. To further divide terms, the operating system can be broken up into the **kernel**, which are the core, unremovable components of a computer, and the **transient** components which are still important but come and go as the computer is running.

At the CPU level, the computer just follows instructions one by one, as they appear in memory. How then, does an OS control the computer, if it must give CPU control to another set of instructions to run a program? The secret is a precise use of interrupts – events which force the switch from **user mode** to **kernel mode** – where the operating system can do the dirty work of the computer. When the CPU acknowledges the interrupt, it reports to an interrupt vector, normally at the bottom of memory, which tells the CPU where to go for the next instruction. The CPU also saves all the registers to the stack, and then jumps to the kernel code. From then on out, its up to the kernel to do what it wants. The OS begins by saving register entries, then uses the temporary stack to execute its code. If the interrupt is synchronous, it is known as a **trap** (a system call, a page fault, or an error).

Chapter 2

Process Management

The atomic components of a computer in motion are **processes**, which can be understood as a program in execution. A process is not a static object, in the sense that one can 'store', 'retrieve', or 'edit' a process in memory. A program is a description of an algorithm, whereas while a process does contain such a description (part of its **text section**), it also contains all other junk that is gathered when running a program: for instance, the status of the process as it is executing (the program counter, process number etc.), collectively called the **process control block**, the **data section**, and **address space**.

It is up to the processer to manage how the processes operate on the computer. In the past, systems used a **Uniprogramming** scheme, where only one program was ever on the go at any point in time. **Multiprogramming** is the modern approach, where multiple processes can be ran at any point in time – this is what enables you to read the PDF copy of this book, while playing music, while an update to your computer is being downloaded, all at the same time, on the same device. Furthermore, it means processes can run much more efficiently; if a process needs to wait for I/O, or some other event, it doesn't block other processes use the CPU. Back in the day, this approach was unviable due to memory constraint, but the obvious advantages of the multiprogramming approach are now easily obtainable.

Our first concern is how an OS manages processes. Firstly, the OS must know all useful info about a process, so it may intelligently manage them. This information is often kept in a **process table**, an array containing structures for each process. The process control block collects all of the status of a process into one place. That is, it contains that:

1. **State**: If the process is new, ready, running, waiting, halted, etc.
2. **Program Counter**: Address of the next instruction for a process.
3. **CPU registers**: Basic memory slots used to manipulate a program.
4. **Memory-management information**: Additional registers used by the OS, as well as additional memory allocated in RAM.
5. **Accounting Information**: Amount of CPU/real time used, job/process numbers, etc.
6. **I/O status information**: I/O processes allocated to the process, plus access priviledges to other files on the system.

In truth, 'multiprogramming' does not need programs to execute all at one time (though with the rise of multiple core CPUs, this is not impossible) – it needs only keep up this illusion to the user by rapidly cycling between running different processes. The **dispatcher** is that component of the operating system which decides which order processes should execute in. Most of the time, the status of a process follow a simple state transition diagram, cycling between 6 states:

1. New: processes which haven't been through **resource acquisition**.
2. Ready: Processes ready to run, but which are not executing.
3. Active: Processes that are executing at a given moment.
4. Blocked: Processes that cannot execute until a given event occurs.
5. Stopped: A special case of blocked, where a user suspends a process.
6. Exiting: A process about to go through **resource release**.

An operating system implements an array known as a process table to make decisions with multiple processes. Each entry is a process control block. When it needs to, the dispatcher uses one of a selection of strategies to pick a 'Ready' process, and give it control of a CPU.

When a process is running, the dispatcher process has no control over the CPU. The common manner of doing this is to set a timer interrupt (the

alarm clock approach), which will automatically exit out of a process to an **interrupt service routine**, known as the **sleeping beauty** approach. It is up to this routine to save program state, and restore the state to the next process used. Either we can save this to the disk (slow), isolate the memory from the other processes (memory-inefficient), or just trust memory won't be overwritten by other programs (volatile).

In Unix, we create new processes using the *fork()* command, which clones the PCB of an existing process. The process created is the same in all respects (with separate memory for copied variables) except that the return value of the fork command is 0, not 1. The OS uses fork to start running any other process (by using the exec() command, one replaces the program a process is using by another program). In fact, to begin running a computer, a technique known as **bootstrapping** is used, where we initialize a single process, which begins to fork into other process, eventually causing a tidalwave of processes to emerge, 'pulling the computer up by its own bootstraps'. After a parent has forked, it should eventually use the *wait()* command to ensure all its processes have been terminated. Otherwise, a child process may become an **orphan**. Fortunately, Linux has an 'init' process, which adopts all orphans, and then kills them by periodically waiting, ensuring that no processes are left-behind. Nonetheless, their (now defunct) PIDs may still occupy the resource table, a resource leak. If this happens, the process is known as a **zombie** process.

A **thread** is like a process, but more lightweight. When you create a thread, only the minimum functional information is copied for both threads to function (i.e. the stack and program counter), but RAM is not copied. A process is an address space with one thread. All stacks for each thread must be in the same address space, since the address is shared. Threads give us the advantage of a lightweight method of enabling concurrency, but the come at the price of having to look after data much more carefully. Real parallelism can only be achieved by having multiple CPU cores. **Psuedo parallelism** use rapid switching of CPUs to simulate the effect of real parallelism. **Hardware parallelism** is where the CPU has multiple I/O devices running in tandem with the CPU, enabling the computer to do a (albeit more restricted) set of instructions at the same time.

We can either implement threads in user space or in kernel space. In the first case, the implementation is OS independant, but requires much more busywork. Processes need to implement their own thread table to keep track of threads specific to the process, and we can't use interrupts to switch between threads. Nonetheless, this enables processes to schedule their own threads efficiently, and multiple threads can be run in a single time **quanta**, the amount of time a process has before it is interrupted by another process. When thread management is implemented in kernel level, you get all the process management for free, but you have to switch contexts every time you want to switch threads.

Processes **cooperate** if they share resources, and **compete** if they fight for individual resources and CPU time. Competing processes cannot influence side effects with one another.

Chapter 3

Resource Sharing

Processes each have individual memory locations. They compete, which means that no data between the processes is shared. How, then, are processes able to communicate with one another? When they do, how do they avoid buggering each other up? These issues are encompassed under the topic of **Interprocess Communication**.

Even assuming that data is able to be easily shared between processes, we still have the problem of data confliction - how do we ensure that we don't void the other processes's correctness by changing the data when it is running. **Race conditions** occur when the output of a process depends explicitly on when it is run relative to other processes – a big no no, since processes don't get to pick, and cannot ensure which order they will run in! What we desire is **mutual exclusion** – only one process can access or modify shared data at any one time, regardless of the order processes are run. The code that accesses shared memory is a **critical section**.

One way to ensure mutual exclusion is to disable interrupts. Of course, no other program can edit data, if a process can never be switched out until it is done with the data. This is a simple solution, but for security reasons it is unwise to give user access to interrupt control, and with multiple CPUs, it is impossible to disable interrupts, for we must have a way to manage processes on the other cores.

Another solution is to have shared lock variables that indicate when a process is being used. The problem with this solution is that the act of reading the variable, changing it, and checking if it set, is not atomic. Both processes might read the variable as empty before changing it, causing a race condition.

An alternative method is **strict alternation**, where a certain integer valued 'turn variable' x is given. Each process is assigned a unique integer, and checks whether x equals its turn, and runs the criticial code if it is. When it is done, it increments x, giving access to the criticial section to the next process. The problem of this solution is that we have to wait our turn for an unbounded amount of time – other processes might never run their turn, and thus never increment x. If you continually test a variable until it is your turn, this is **busy waiting**. A lock which uses busy waiting is called a **spinlock**.

A good solution to the mutual exclusion problem is to have the qualities of

1. **Mutual Exclusion**: No processes run the code at the same time.

2. **Fair Scheduling**: Every process will eventually be able to run the section. That is, **livelock** will not occur, where some processes are eternally blocked (**starved**) while other processes continue to run.

3. **Fault Tolerence**: Failing programs should not cause other processes to fail.

4. **Lack of Assumption**: Makes no assumptions about program interleaving order and CPU speeds.

5. **Efficiency**: Little blocking and busy waiting should occur.

A **preemptive resource** is a resource that can be taken away from a process and given to another process at any time without error. A **non-preemptive** resource is one we are worrying about now. Once a program begins using the resource, it enters a critical section, and the resource cannot be given away until it is finished.

A set of processes is in **deadlock** if each process is waiting for an event that only another process can execute. Because none of the processes can run, they are stuck enternally. Coffman showed that a deadlock only occurs when:

1. **Mutual exclusion**: Each resource can only be held by one process.

2. **Hold and Wait**: Processes holding resources can request new resources.

3. **No Preemption**: Resources granted cannot be forcibly taken away from a process.

4. **Circular Wait**: There must be a cycle of processes, each waiting for a resource held by the next member of the chain.

Dekker's algorithm was the first synchronization algorithm to work property, and works for two processes,

```
while (TRUE) {                  while (TRUE) {
    pAtrying = TRUE;                pBtrying = TRUE;
    while (pBtrying) {              while (pAtrying) {
        if (turn == B) {                if (turn == A) {
            pAtrying = FALSE;               pBtrying = FALSE;
            while (turn == B);              while (turn == A);
            pAtrying = TRUE;                pBtrying = TRUE;
        }                               }

        <CRITICAL SECTION>              <CRITICIAL SECTION>
    }                               }

    turn = B;                       turn = A;
    PAtrying = FALSE;               pBtrying = FALSE;
}                               }
```

The bakery algorithm solves the problem in full, invented by Leslie Lamport.

```
boolean choosing[n];
int number[n];

while (TRUE) {
    choosing[i] = TRUE;
    number[i] = max(number[0],...,number[n-1]) + 1;
    choosing[i] = FALSE;

    for (j = 0; j < n; j++) {
        while (choosing[j]);
        while (number[j] != 0
                && (number[j], j) < (number[i], i));
    }

    <CRITICAL>

    number[i] = 0;
}
```

These solutions work, but are complicated to implement. Luckily, hardware support has enabled these problems quite trivial to solve. Most modern day processers include an atomic **test and set** instruction, which reads the content of some memory word, called the lock, sends the lock value to a register, and sets the lock to one. Since CPU instructions are indivisible, the value of the lock guarentees whether the value is locked or not.

Using the test and set instruction, we can implement a **semaphore**, a variable that is used as a condition to control access to a shared resource. A **counting semaphore** is integer valued: a positive value indicates the number of shared resources that can be accessed as a certain time, a zero indicates a block, and a negative value indicates the number of processes in the queue to access a variable. This is implemented customarily using two operations. The increment, or *V* command frees a process, and the *P*, or decrement operation lcoks the semaphore, or busy waits until its available. A **binary semaphore** takes only two values.

A **monitor** is a high-level abstraction which combines and hides away shared ata, operations on data, and synchronization, so that a programmer does not have to ensure mutual exclusion himself.

The **bounded buffer** problem, or **producer-consumer** problem, is as follows. A producer has to send information down a fixed-size buffer to a consumer of the data. We can solve this using semaphores, as follows:

```
wait(empty);                    wait(filled);
wait(accessOK);                 wait(accessOK);
    get empty buffer                get full buffer
    from pool of empties            from pool of buffers
free(accessOK);                 free(accessOK);
    produce data                    consume data
wait(accessOK);                 wait(accessOK);
    add full buffer to              add empty buffer to
    pool of fulls                   pool of empties
free(accessOK);                 free(accessOK);
free(filled);                   free(empty);
```

Chapter 4

Interprocess Communication

Interprocess communication is effectively sharing information between processes. For instance shared memory explicitly shares information. **Message passing** uses a messaging system to explicitly send information without sharing memory locations. For instance, UNIX pipes implement a form of message passing.

Just like memory sharing, we need to have ideal properties our methods should have. We need to decide on

1. **Form of Communication**: Do we send messages **directly** or **indirectly**.

2. **Buffering**: How and where message are stored.

3. **Error Handling**: Dealing with exception conditions.

We need a send() command and a recieve() command to send messages. **Synchronous** forms of communication ensure that send and recieve block. A process only returns from a send() command when it is guarenteed that some other process has recieved the message, and vice versa. This is known as a **rendezvous**. An **Asynchronous** method does not block on a send command, but may block on a recieve (possibly many variants of the recieve command exist, to be used at the programmer's whim).

Processes **directly communicate** if one process is sending a message to one other process. The reciever knows exactly where the message is coming from. Indirect messaging is less secure, but requires less synchronization between programs. Indirect communication is often implemented as a **mailbox**, which are special repositories where information can be stored and accessed. In the case one reciever uses a mailbox, the mailbox is called a **port**. A port usually needs a buffer to store info. When the port is **bounded**, the buffer has a limited capacity, and a sender must wait until an empty spot is available before sending the message. When a buffer has **indefinite capacity**, the sender never waits, whereas if the buffer has **zero capacity**, message sending must be synchronized, the sender must send the message directly when the recieved gets it. In the asynchronous case, the sender must have additional mechanisms to guarantee the message is sent correctly.

We need to handle errors in message communication. The most common errors are:

1. The process could terminate before a message is processed.

2. A message could be lost in the communication network.

3. A message could be corrupted in transmission.

We won't discuss how to avoid these errors, but keep them in mind.

One cannot use semaphores or monitors to pass a message, since semaphores require globally shared memory, whereas monitors require centralized control. Nonetheless, message passing, in tandem with blocking, may be used for synchronization. In Unix, one may use signal between processes. A signal is **generated** when the event first occurs, and **delivered** when the process takes action on that signal. A signal is **pending** when generated but not delivered. Signals, also called *software interrupts*, generally occur asynchronously. A process sends a signal via the *kill()*, *raise()* (to send the signal to itself). To choose what code is executed when a signal is delivered, one uses the *signal()* command.

Chapter 5

Scheduling

When we learnt about processes and psuedo-parallelism, we acknowledged that a dispatcher needs to be able to decide which processes should run, given that there are multiple ‘ready’ processes at a given time. This is the task of (job) scheduling, controlling the order of work to be performed by a computer system. A process alternates between a **CPU burst** and an **IO burst**, using the processor, and aquiring more information from external devices. The CPU burst is of primary importance, since all processors must use it. One must maximize CPU burst efficiency for scheduling.

Long term scheduling determines when processes are allowed to aquire resources. When resource utilization is low, more jobs should be admitted, whereas the opposite should be true if resource utilization is too high. Switching between processes requires some small overhead, but this overhead is multiplied when switching between more and more processes. **Thrashing** occurs when the CPU spends more cycles switching between processes than actually running programs.

One controls the **degree of multiprogramming**, the number of processes in memory, to prevent throttling and spinlock. The job of deciding when to suspend or resume a process is **Medium term** scheduling. **Short term** scheduling decides which ready processes should have CPU next.