OS CS219 Notes

January 16, 2024

1 Introduction

An **OS**(operating system) alias **Master Control Program** alias **System software** is software than enables the user to access the hardware resources of a computer in a controlled manner. It acts as a layer of abstraction allowing the user to not worry about hardware level access and just provides access to a few methods needed by the user

The OS has several uses/functions in a computer

- Manages resources as a single **central entity** and hence efficiently
- **Virtualizes** physical resources to be utilized by multiple processes[1]
- **Isolates** and protects processes from one another by not allowing direct access to hardware
- Provides a set of system calls for the user to access hardware resources
- Starts processes and allocates and manages memory required by said process during execution

Thus it is easy to see that the OS has several important functions to perform inorder to enable an abstraction of hardware from the users

2 Process and Virtualization

A process is just the sequence of execution of instructions by the CPU . When we write and compile a program it gets converted into a sequence of instructions whose first *instruction address* is fed to the PC and as we know the sequence of instructions for that program starts getting executed one by one. This is called a process. So when we run a program a *process* is created.

[1] process is defined later

2.1 Context switching

How do we tell the CPU to start a process. First obviously we need to feed the address of the first instruction of the process to the **Program Counter**. We also need to set the stack pointer and other registers with appropriate values. This is called **setting up the context** for a process. This job is done by the OS .After the context is set the OS takes a back seat and allows the CPU to do its work

However this has a few issues. Firstly, when the process requests for data from say the hard drive there is a gap where the CPU is on standby which is wasteful since other processes also require it. Secondly, we also want responsiveness from our system ie when we have a process running we also may want to interact with other processes.

Both of these issues are solved by **concurrent** execution. Basically we run a process for a while and when the CPU is on standby or after a particular interval of time we save the *context* (ie) states of all registers including PC and start the other process by setting up its context this repeats for a while and eventually the partially executed process's context is set and it is continued. This is reffered to as **context switching** and is an important part of Virtualization[2] of the CPU. Note that a part of the OS (ie) *OS scheduler* decides which program to run at what time

2.2 Virtualization

Virtualization refers to the creation of an illusion that each of the processes have full access hardware resources[3]. This enables the hardware to act much more powerfully than they are capable of. For example, as mentioned above context switching creates the illusion of the presence of multiple cores each assigned to one process whereas in reality it is just one core. Infact this is refered to as **Hyperthreading**. Apart from the CPU memory, addresses can be virtualized.

3 Memory allocation and Isolation

3.1 Memory

As we learnt in CS230, memory for a process/program is allocated as a fixed number of bytes. The inital bytes of this memory is the instructions and the global/static variables. Local variables aren't initialised in memory since we do not know the number of times each function is called, instead we have a dynamically growing stack whose starting

[2]Read further to know what it is

[3]It is useful to think of Virtualization as the OS lying to each process about it having full access to a resource

address is stored in the special *stack pointer register*. This stack grows and shrinks as necessary functions are called and they return values. [4].

Apart from this we have a heap which can be accessed by user to store dynamically increasing data structures. We can request the OS to allocate certain number of bytes and return a pointer to said bytes

Here again however the OS plays tricks on the processes. Since it is impractical for the OS to allocate to allocate memory for the process continguosly it allocates them in chunks but returns a **virtual address** (Recall Virtualization) to the program. This "virtual address" is the address returned when the user requests the OS for an address of the data stored. Here again the OS lies to the process creating the illusion that it has access to continguos memory starting from some location

3.2 Isolation

Now we understand that the OS allows multiple processes to run at the same time and share resources But this raises a huge issue since processes are supposed to be independent and processes being to affect other processes would cause problems. The OS takes care of this too by maintaing strict control over access to hardware

The OS is the only entity with access to hardware and process can make specific requests to the OS to use hardware via *system calls*[5]. Infact there are two types of instructions and processer modes corresponding to them

- **Privileged instructions** - special instructions that can interact with hardware. Generated by syscalls,device drivers CPU is in **kernel** mode while executing them
- **Unprivileged instructions** - simple instructions that do not need access to hardware. Given by user processes. CPU is in **user** mode while running them

The CPU is always in user mode except the following cases.

- A syscall is made
- Interrupt occurs
- Error needs to be handled
- Context switching needs to happen for say concurrent running

Note that when a syscall is made the OS code pertaining to it is executed and then control is return back to user code

Interrupt: In addition to running programs a CPU has to handle external inputs from devices like a mouse click. This is called an Interrupt. During an interrupt control is given to the OS(Kernel mode of CPU) which deals with the interrupt and returns

[4]The structure used here is a stack since functions are inherently *LIFO* functions called last return first

[5]syscalls can't be accessed directly usually. They are in a language's standard library

control to the user process [6]

Device Driver: I/O is managed by the device controller(Hardware) and device driver (software)[7]. The driver initializes IO devices and it starts IO operations like reading from the disk. It also handles the above mentioned interrupts

4 Process abstraction and attributes

As we have discussed above about a process it is a sequence of instructions running in the cpu. Also as discussed in section 2.1 a process can run for a while, then be blocked and run again. Hence a process switches from one *state* to another during its execution. We can note that this process state changes only when the kernel goes to user mode.[8]

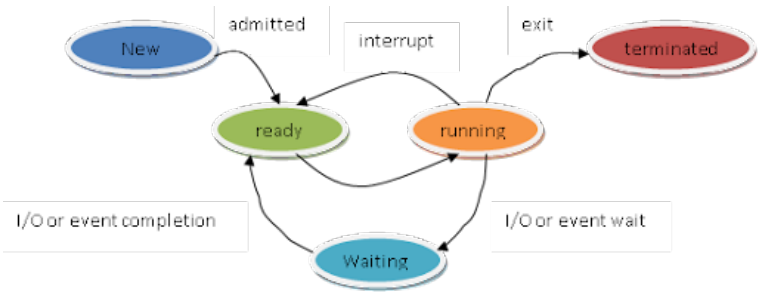

Figure 1: Figure to show process state switching

A process is defined by several *attributes* that define it:

- **PID**: Unique process identifier given to each process
- **PPID**: PID of a parent process[9]
- **Context**: Context is saved when switching happens. We have discussed what the context of a process refer 2.1
- **File descriptors**: A record of all the files open by a process is stored in form of pointers in an array. Elements at index 0,1,2 refer to std input, std output and error files. As we open more files for a process a pointer to it is created and added to the end of the array. This poiner is what is returned as a *file descriptor* for the user to perform read or write operations.
- **State**: A process can be in 3 states
  1. **Running**: The CPU is currently executing instructions of the process
  2. **Blocked/suspended**: The process cannot run for a while. Maybe it requested data from drive and is waiting for its arrival

[6]This means saving context, handling the interrupt and setting context of the past status
[7]This is part of the OS
[8]Since the change is done by the kernel's OS scheduler
[9]Parent discussed later

3. **Ready/runnable**: Process can be run and is waiting for *OS scheduler* to give it a CPU/core to use

- **Memory**: Each process uses/is allocated a fixed amount of memory by the OS and its locations are stored
- **Page Table**: The OS lies to the process about memory addresses(Virtual address [10]). The real address mapping to the corresponding virtual address is stored in the page table. The page table can be used to get the real address corresponding to each virtual address
- **Kernel Stack**: The context of a CPU is saved in this kernel stack when context switches occur. This stack is stored in a seperate memory and it isn't accesible by user code The OS uses this stack since it doesn't trust the user stack

4.1 Process Control Block

All the above mentioned attributes of a process alone with more necessary information is stored in a data structure called the process control block(PCB)

It is called by different names in different OSes:

- *struct_proc* in xv6
- *task_struct* in linux

The above mentioned attributes of various processes are stored in the PCB in form of the **ptable** or process table which is a data structure storing all the *proc structs* each of which has all the data corresponding to each process

In **xv6** the ptable is just an fixed size array since it is a simple system. However in real world kernels it is a dynamically expanding data structure.

The **OS scheduler** iterates over the ptable picks a *ready* process and assigns it a processor to run it .A process which needs to be put to sleep (Eg. IO from disk) will be put to sleep and another is picked from processor

5 Booting

We need some system to load up our OS into the CPU during start of the system. The **BIOS**(Basic input output system) is present in the non-volatile memory of our system which loactes boot loader in the boot disk. It is a simple program whose job is to locate and load the OS. It is present in the first sector of the boot disk It sets up context for the kernel and gives control to kernel

BootLoader must fit in 512 bytes of the Boot disk to be easily located which isn't sufficient to load up current complicated systems . So the 512 bytes(simple bootloader)

[10] refer Section 3

load up a more complex BootLoader which loads OS onto the CPU

6 API:Application Programming Interface

System calls provide an interface to the OS called the Application Programming Interface (ie) the set of syscalls given to the user constitute the **API**,

Two types of syscalls are:

- **Blocking:** Syscalls that block the process that called it[11] and the OS comes back to the user process after a while
- **Non Blocking:** Syscalls that are called with the user instructions which acts along with user process without blocking the calling process[12] (eg. getpid())

If every OS has different syscalls *portability*[13] is an issue. For this purpose all the OS providers decide on an universal set of syscalls [14] to provide called the **POSIX** API. Interestingly, since the instructions for syscalls maybe different this is why we may have to recompile to run code on another OS

Hence the hierarchy of a syscall is somewhat like:

User code → Standard library functions → Syscall in the function → Syscall in assembly instruction → OS

In xv6 we are directly given syscalls in the standard library in a user friendly function call. Usually we are given syscalls at the assembly code level since we usually need to change the privilege of CPU[15]. Hence we need to understand that syscall is **NOT** a regular function call

6.1 Fork

Each process is created by another process. Such a process emerging from another is called the *child* of the *parent* process. The syscall used to create such a process is called *fork*. `Init` is the initial process from which all other processes are *forked* When you call fork:

- New child process is created with new **PID**
- Memory image[16] of the parent is given to the child

[11] Like when you wanna read from disk which takes time
[12] The process which calls the syscall
[13] ability to run same code on multiple machines
[14] The implementation of said syscall differ
[15] Done using INT in assembly
[16] the heap,stack,instructions,data is the memory image of a process

- They run copies of same code

Note that while the child may share the virtual memory with parent. It is in a different physical memory location

What is the point of running the same process as a child again? There is none. They aren't the same process due to one key difference. The `fork()` returns 0 in the child process and returns the PID of child to the parent. Hence we can make the parent and child run different code using the different value returned by the `fork()` function. Note that `fork()` returns -1 when forking fails. This process seems to be generating different process running some redundant code. This is not the case usually due to reasons we will see later. Interestingly as of yet the parent also needn't run before the child since they are independent processes

We can also have nested forks asin multiple forks() in a program. This will make a parent and the child each of which will also call another fork and so on.
xv6 fork() code:

- Allocates memory for new process and gets PID
- "np" a pointer to struct proc of child is created
- "currproc" points to struct proc of parent
- Copies info from currproc to np
- Child is made runnable and put on ptable and PID is returned in parent and 0 in child

6.2 Exit and wait

When a process is done it calls exit() to terminate. Exit is called at the end of main() automatically. Exit does'nt clean up the memory of a process and the process is in a dead **Zombie** state.

Parent process of a child calls wait() syscall which cleans up the memory of a zombie child. If wait() is called in the parent before child is a zombie the parent is suspended and waits till the child is done running.