OS CS219 Notes

February 22, 2024

Lecture 1

Part I

Introduction to Operating Systems

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

[1]process is defined later

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.

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.

[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

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 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

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

Lecture 2

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 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

[5] syscalls can't be accessed directly usually. They are in a language's standard library
[6] This means saving context, handling the interrupt and setting context of the past status
[7] This is part of the OS

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]

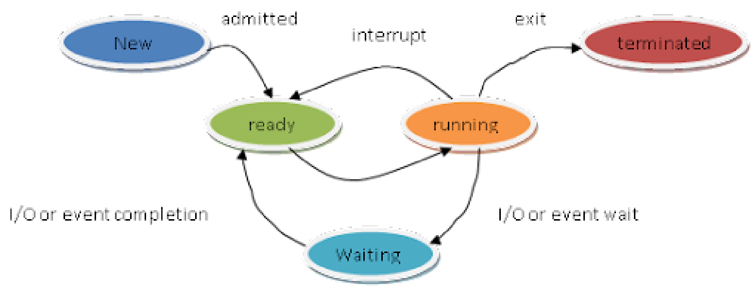


Figure 1: Figure to show process state switching

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
  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

[8] Since the change is done by the kernel's OS scheduler

[9] Parent discussed later

[10] refer Section 3

- **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. Each process has its own kernel stack

4.1 Process Control Block

All the above mentioned attributes of a process along with more necessary information is stored in a data structure called the process control block(PCB)[11]

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

[11]Stores the details about one process

Lecture 3

Part II

Process Management

1 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[12] 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[13] since they return immediately (eg. getpid())

If every OS has different syscalls *portability*[14] is an issue. For this purpose all the OS providers decide on an universal set of syscalls [15] 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:

$$\text{User code} \rightarrow \text{Standard library functions} \rightarrow \text{Syscall in the function} \rightarrow \text{Syscall in assembly instruction} \rightarrow \text{OS}$$

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[16]. Hence we need to understand that syscall is **NOT** a regular function call.[17]

[12] Like when you wanna read from disk which takes time
[13] The process which calls the syscall
[14] ability to run same code on multiple machines
[15] The implementation of said syscall may differ
[16] Done using INT in assembly
[17] Lottt more detail on this further

1.1 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[18] of the parent is given to the child
- 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 way to create process to perform very different functions as compared to parent. We will see another way to create processes for that case 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 as in 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

[18] the heap,stack,instructions,data is the memory image of a process

Lecture 4

1.2 Exit and wait

When a process is done it calls exit() to terminate. Exit is called at the end of main() automatically. Exit doesn't 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 and returns the PID of said zombie process(or -1 if no child). If wait() is called in the parent before child is a zombie the parent is suspended and waits till the child is running. You can also call waitpid() which reaps only a process with a particular pid whereas in normal wait() some arbitrary process out of all zombied ones processes are reaped. Note that one wait() reaps only one child. So we need a wait() for every fork().

What if parent dies without calling wait(). Then the child continues to run as an orphan process. In this case `init.` adopts the orphan process and becomes its parent and eventually reaps[19] it. This is done when the parent calls exit which makes all the parent pointers of its children point to `init.`. Note that `init.` comes into play iff the parent dies, not when it is alive in anycase. So `init.` keeps checking if there is an orphan to adopt and eventually reap. If parent is alive but doesn't call wait then system memory fills with zombies[20]

1.3 Exec

As we saw the fork() method seems complicated with if-else blocks for parents and children and there seems to be redundant code. We create child to do similar work as parents lots of times. If this is not the case (ie) parent, child are doing different things(and we don't want the parent around) we can use `exec`. exec() is used to get a new memory image(using that of the old process) which is used to run an executable which it takes as an argument. It is not similar to fork() since it doesn't create a new memory image it just replaces the parent's memory image and it copies the executable's memory image to run the executable. The ptable is also updated with the new details of the process[21]

So whatever code is given after exec is never run by the child since it overwrites the parents memory image with the executable given as a argument. However if exec fails

[19] calls wait() and erases the memory of the zombie
[20] ahhhhhhh apocalypse
[21] but the PID,PPID is the same

then the parent's image copy only is present in the new process and thus all the code after exec in the parent is run by the child once it gets access to the CPU.

Lecture 5

2 Shell

Shell is the process which handles accepting and executing terminal commands

Listing 1: Shell code

```
while(1){
    input(commands);
    int ret =  fork();

    if(ret == 0){
        exec(command);
    }
    else{
        wait();
    }
}
```

The basic working of the shell goes like this:

- Shell forks a child which calls exec to run commands
- Why doesn't the shell call exec() itself. This is since we want the shell program to keep running and not get replaced
- Some commands have code written in the OS itself like `cd`, since they need to maintain the pwd[22] while others have executables called by exec() like `ls`.
- User commands run in *foreground* (ie) can't accept next command till previous one is done
- *Background execution* is when we give a command followed by & the shell runs command but doesn't wait for it to finish. So reaping is taken care later by the shell using a method where wait is invoked without blocking parent.
- We can also run multiple commands in *foreground* sequentially(one after another) using && or parallely using ||

Some things taken care by the shell:

I/O redirection: Every process has some IO channels or "files" open which can

[22] present working dictionary

be accessed by file descriptors[23]. Parent shell can manipulate these files descriptors of child before exec() to do stuff like I/O redirection (ie) by changing the STDOUT file to out desired file or STDIN to desired file. Basically the process still thinks its getting from STDIN or outputting to STDOUT but we change the file descriptors to point to files we want, essentially tricking the process to output/input to/from the desired file

Pipes: Pipes are when the output of one command is given as input to another command. Shell creates a temporary buffer in OS called well a "pipe" and the STDIN-file descriptor of the other command process is made to point to the "pipe". Basically pipe is redirected as input to the next commands.

Signals: Signal is a way to send notifications to process.(eg. kill -9 PID[24]). There are some standard signals available to each OS. SIGINT, SIGCHLD, SIGTERM, SIGKILL[25] ....etc. Note that the kill command can send all signals and which signal it sends is determined by id in "kill -id pid" which conveys the signal. The OS can also generate signals on its own and not from processes(eg. CTRL + C sends SIGINT).

When we send a signal it is sent to all the processes in that process group. A process group is an organisation where sets of processes are treated as group. By default a process belongs to its parent's process group. CTRL + C sends SIGINT to all processes of the *foreground* process group . The syscall `setpgid()` can be used to change the process group of a process.

Signals to a process are queued and delivered to the OS which handles them. It knows to ignore certain signals and to make some processes stop for certain others. User processes can also define their own signal handlers using the signal syscalls to overwrite default behaviour. The process jumps to the process's signal handler and back to the process if it exists after signal is taken care of. However note that some signals like SIGKILL can't be overriden by a process's signal handler since the OS needs to maintain some power over process incase they go rogue.

[23] STDIN.STDOUT,STDERR

[24] SIGKILL

[25] Interrupt,signal to parent when child terminates,terminate,kill respectively

Lecture 6

3 Trap handling

As we saw before the CPU can execute instructions in User mode or Kernel mode with differing level of privileges[26]. The process of going to Kernel mode is called a "Trap" the CPU *traps* into the OS code via the running process[27]. The OS isn't a seperate processes it just runs in the same process which called it (by trapping the OS) just in the Kernel mode of the CPU.

Note: Random doubt, How are interrupts and signals different. Interrupt is a message from a *device* to the system which is handled by OS. Signal is a message sent from one process or another. When we give CTRL + C that is an *interrupt* from the keyboard which the OS handles to create a *SIGINT* signal to the running process

Why is a sycall() even different from a function call?? To understand lets see what happens when a function call is made

- Allocation of memory on stack for function arguments, local variables. Note that this doesn't happen during function definition.
- Push return address and PC jumps to function code
- Save register context of the calling process
- Execute function code and once done pop return address and pop register context

In a syscall some similar things are done

- Push return address and PC jumps to function code(How does the user know where OS code is?)
- Save register context for the calling process
- Execute function code and once done pop return address and pop register context

But the important difference is *where* the memory for the syscall() variables are allocated and what information is exposed to user. We can't expose the PC locations for various syscalls since they can be misused and abused by the User. It also completely takes away control from the OS and leaves the system vulnerable to attacks. Also the OS is *paranoid* in the sense that it is designed to not trust the user. Since the User has access to the user stack the OS doesn't use that stack for allocating variables for its syscall(). It rather has its own secure **kernel stack** which is not accessible to the user.

[26] Refer 3.2

[27] read further to find out what a trap is

3.1 Kernel stack and IDT

The kernel stack is used for running all the OS code. There is a part of the PCB given to these secure OS processes which aren't accessible to the user. The context for a process calling a syscall is pushed onto the Kernel stack and is popped when the syscall is done.

Again how does a process know which PC to jump for a syscall(). We have the **Interrupt descriptor Table** for this purpose. It is a special data structure which has the mapping to the PC at which each syscall()'s code is present. This PC is indexed by n which is the argument passed to the *trap* instruction. Accessing this table can only be done by privileged instructions.

When the user wants to make a system call we can't do it with OS code directly since it involves changing permission to Kernel mode which the OS code needs anyway to run[28]. We obviously can't make the user code do it due to *lack of trust*. The only trust worthy entity which is capable of changing permission to run OS code is the hardware. The hardware creates an interrupt which calls a special "trap instruction" or (INT n) at the assembly code level with an argument which indicates the type of trap(to indicate which syscall,interrupt,error is calling it). After this the CPU is finally capable of running OS code and thus OS can perform whatever the trap was called for.

The **IDT** is setup during Bootup of a system to give the PC's of syscall() depending on which task is to be done. This PC is used by the interrupt instruction to jump to a syscall()'s PC thus maintaining security. Thus, note that the **IDT** is a very important data structure and an ability to compromise it can give access to the entire system since we can locate where all the OS's code and virtual memory is and thus gain access to the entire system

How does trap make the privilege to *Kernel mode*? What all does it do?

- CPU privilege is increased
- CPU shifts it's stack pointer to the kernel stack
- The user context for the calling process is saved(for a syscall)
- The PC (can be obtained from interrupt table) is changed

Now we are in a position to start running the OS code. After the OS code is done handling the syscall/interrupt, it calls a return-from-trap instruction(Trapret and iret).

- CPU privilege is decreased
- CPU shifts it's stack pointer to the User stack
- The user context for the calling process is copied to CPU

User is unaware that it was even suspended and continues as if nothing happened.

IDT Lookup: The IDT is basically an array whose starting index is stored in a CPU register and the interrupt number 'n' given in INT n is the index of the PC we need.

[28] problem of which rat will bell the cat

Lecture 7

3.2 Trap Frame in xv6 and trap handling

- A data structure called the Trap frame is pushed onto the kernel stack when a trap is encountered and then it is popped by return-from-trap
- It contains various register values to be saved and not get lost during trap handling (pushed by OS).
- The "int n" pushes a few registers w(PC,SP etc.) and jumps to kernel to handle the trap and the rest of the registers are pushed by kernel code after which trap handling is done
- **EIP,ESP** has values that get modified as soon as the "int" instruction is completed. So we need to save them with the int n instruction itself and cannot wait for OS code.
- IDT entries for all interrupts set the EIP to point to the kernel trap handler "`Alltrap`" which is common for all traps into the OS regardless of the reason for why the trap was called
- `Alltrap's` assembly code pushes remaining registers to complete trapframe on Kernel Stack
  Note: Why do we have to save registers doesn't the `int` instructions do it for us? The hardware only saves the bare minimum and absolutely required values like **EIP,ESP**. It is upto the OS to save the remaining registers (depending on if it needs to) inorder to restore them after trap handling
- Thus after `Alltrap` is executed our sturct Trap frame has all its values set appropriately
- Note that after the `Alltrap` is completed the ESP points to the top of the trap frame
- After this we invoke the `trap()` function in C which actually handles the specific reason the trap was called for[29]

So we can see that `Alltrap` is written in assembly to do the bare minimum that **must** be done in assembly like pushing registers to Kernel stack. After this we go to a high level language like C enabling us to code the logic in the more compliacted part of actually handling the trap much easier. After the trap handling is done the assembly code `trapret`'s instructions are executed. This pops the trapframe from the kernel stack (things pushed in the `Alltrap`s). It calls the `iret` instruction which is the opposite of int. It pops values which int pushed to kernel stack after which it changes privilege level back to user mode. After this the process which trapped the OS continues running. In xv6 if it was a syscall that called the trap after all the trap handling is completed the

[29]It knows the reason from the value of the eax register which it passesn as an argument, int n

OS puts the return value for the syscall in the `eax` register.

So in conclusion the int instruction and `Alltrap`s work together to save values when a trap is performed. C code(trap handler) takes care of the actual trap. Trapret pops the register context and `iret` returns from trap and control goes back to user

Depending on the value at n we know if the int was called for a syscall or an interrupt and can handle it appropriately. How do we know when a device interrupt occurs? If a particular pin connected to the device has a high value then an `int n` in passed to the CPU with an appropriate "n" value and thus the CPU can handle the device interrupt

3.3 Timer interrupt

We thus understand how a trap can give control to the OS. But it maybe that intentionally or accidentaly that a program never does a trap. So the OS would never get the control of the CPU. What if such a process goes into an infinite loop? How will the OS stop it? Alternately if we want to do a context switch how will the OS get control? One option is to just assume the process is not malicious and it is smartly written to trap into the OS at regular intervals. However the better option is to interrupt every process after a particular amount of time has passes and then trap into the OS to handle this interrupt and check if everything is okay or do a context switch if required. This interrupt is appropriately called an **Timer interrupt**. The Timer interrupt is a special hardware interrupt and is given to kernel periodically.

Lecture 8

4 OS scheduler

OS maintains a list of all active processes in a data structure. Processes are added in during `fork()` and it is removed during `wait()`. The **OS scheduler** is a piece of ccode that runs over this dynamically growing data structure and picks a process to run. Basic Job of the scheduler:

- It saves the context of the currently running process in its PCB
- Loops over all the ready processes and picks a process to run(according to its policy)
- Restore context of the new process from PCB and make it run
- Continue as long as system is active

Why do we want to do context switch? Sometimes when OS is in kernel mode it can't go back to the process which it trapped into. For example, when the process has terminated or when it has made a blocking system call. Another reason could be that the OS may not want to return to the same process due to the necessity to give fair oppurtunity for all process to run. This is done through *time sharing* for which context switching is necessary.

The OS scheduler needs to decide on a policy (*scheduling policy*) for selecting a process to run and needs to have a mechanism to switch to that process. The scheduler can be non pre-emptive where it switches contexts only when a running process does a blocking syscall or gets terminated ie when it willingly gives up control. A pre-emptive scheduler will perform a switch even if a process is ready to run. It does this via timer interrupt[30]

4.1 Mechanism of a context switch

How does a context switch happen? Assume we want to switch from process A to process B. Process A is trapped into OS[31]. Kernel stack has register context of user process A in the trap frame. After running some kernel code assume process A can't be run anymore (Say it asked to read from disk which takes time). Now we want to switch to B.

We now save the *Kernel context* (ie) PC of OS where we stopped, kernel stack pointer, some registers etc.. . Now be careful and observe that there are **two different** contexts

[30]Refer 3.3
[31]Refer 3

the *user context* and the *kernel context* of a process. The user context is saved on the kernel stack by a trap into the OS from that process. The kernel context is saved on the kernel stack during a context switch. After saving both contexts we then switch the value of the stack pointer to the kernel stack of B[32] from the kernel stack of A[33].

What does kernel stack of B have? It trapped into OS at somepoint or was context switched out. So its kernel stack has a trap frame, kernel context etc... . We now restore the kernel context of B and resumes execution in kernel mode of process B at the point(ie at the PC where context switch happened) the CPU was given up by it[34]. Now user process B can run as normal after it returns out of trap. Thus we have completed a context switch.

An important point to note is that *context swtiching* can only happen in kernel mode. So when we switch context the `EIP` is an OS code PC. When we switch into another process we switch into the OS code PC address which we were about to execute the last time a context switch happened with the process we are switching into. That is the PC of the last OS code B was executing before the last time it got switched out.[35]

What if B is a new process that has never been run before? Since it as never context swtiched before will it have its *kernel context* saved? We can create a new process only by calling `fork()` and fork doesn't create an empty kernel stack. It gives some dummy value to the stack which solves this issue. We will look at this issue in detail later

4.2 OS scheduler in xv6

In xv6 every CPU has a scheduler thread ie a special process for the scheduler code. Also note that xv6 is a pre-emptive scheduler with timer interrupts. It runs over the list of processes and then switches to one of the runnable ones.

The scheduler process runs and then it context switches to another process. When the current process wants to switch to another process it swtiches to the scheduler and then to another process. Direct context switching doesn't happen. The scheduler is always a middle man.

`sched():` xv6 has a function `swtch()` that actually performs the context switch. The `sched()` functions makes the process context switch to the scheduler. It can't be called in user mode. It is a OS function to be run in kernel mode. The calling process is switched out and the scheduler context is switched in. The scheduler then finds out some other process to run and then `swtch()` context switches from scheduler to that process.

[32] The switching of this stack is the exact moment of the context switch

[33] each process has its own Kernel PC

[34] This is why we store the kernel PC when we context switchd out of A

[35] This point is quite confusing. A helpful hint is to try to visualise what happened that last time a context switch happened with B which stopped it from running

When does a process call `sched()`:

- **Yield:** Timer interrupt occurs, process has run enough and gives up CPU
- **Sleep:** Process performed blocking action like I/O from disk
- **Exit:** Process called exit, sets itself as a zombie and gives up CPU

In both the scheduler,`sched()` function the function `swtch()` switches between two contexts.

Context structure: It is the set of registered stored/restored during a context switch from one process to another. As disucssed before it is pushed into the kernel stack during a context switch. The pointer to this structure is stored in the `struct proc` of the xv6 system as a field called "context"(p→context). This could be thought of as similar to the *kernel context*.

Note that the trap frame is a different structure which also has its own pointer in `struct proc` even though it is also stored on the kernel stack. It is saved during a trap into the OS whereas **context structure** is stored only during a context switch. An example is a timer interrupt, when the process traps into the OS itself the **trap frame** is saved whereas the **context structure** is only saved during a context switch to a different process

Lecture 9

Reference: Caller and callee saved conventions

Caller and callee saved registers: Registers can get used and the values they had before maybe lost when they are used to execute code from a function call. Some registers are saved on stack by caller before involving the function(caller[36] saved) and then these can be modified by the function (callee[37]) freely without the callee have to worry about saving them.

Some registers saved by callee function and restored after function ends (callee saved). Caller code can expect them to have same value on return and doesn't have to save them on its own.

An example is that the **return value** is put by callee in `eax` thus the previous value in it is lost. Thus the caller must be save it. The work of saving registers to make sure they aren't lost in the function execution. These things are taken care of by the C compiler so the programmer doesn't need to take care of them.

There is a very specific convention on what registers are pushed and popped in which order when a function call is made. It is as follows:

- Push the caller save registers(`eax,ecx,edx`)
- Push arguments in reverse order
- Return address (old `EIP`) is put on stack by call instruction(callee saved)
- Push old ebp on Stack
- Set `ebp` to current top of stack (base of new "stack frame[38]")
- Push local variables and callee save registers(`ebx,esi,edi`)
- Execute the function code
- Pop stack frame (of the function call) and restore old ebp
- Return address popped and `eip`[39] restored by the `ret` instruction to jump back to the caller instruction

Thus it is easy to see that in a way the callee saved registers are those absolutely essential for the function call to be returned properly like `ebx,esi` and the rest of the registers are saved according to the needs of the caller.
Stack pointers: `ebp` stores base of stack frame and `esp`(changes with growth of stack) stores the current top. Thus we can access the full of the stack this way. We can also

[36] the code which made the function call
[37] the actual function call code
[38] the portion of the stack given for the function execution
[39] instruction pointer

very easily access arguments since they are on the base of the stack

4.3 `swtch()` in xv6

It first saves the registers in the context structure onto the kernel stack of the old process. It now switches ESP(stack pointer) to context structure of the new process. Pops the registers from the new processes context sturcture(the moment context switch is happening). CPU now has the context of the new process. This entire process is coded out in assembly language since we have to push and pop registers from the stack which cannot be done in c.

4.3.1 Parameters of `swtch()`

Both the CPU thread and the *struct proc* stores the pointer variable to the context structure. `swtch()` has two arguments(two pointers a context** and a context* pointer). It is the pointer to the old context (ie) current process's context pointer (Note that this is the pointer to the context pointer) and the context pointer of the process we want to switch into.

We take the pointer to the context pointer(&(p→context)) of the process we are switching from and not the context itself(p→context) for an important reason. We want to update the context to its latest status. What we mean is that when we context switch out of a process it already has the last context which it had when it was switched out previously. We want to update this context to the current value since when we come back to the process in the future we want to start executing it from this point which we are leaving it at.

We however needn't modify anything about the context of the process we are switching into but rather just wants its value. Thus we just take its context* pointer as an argument directly.

4.3.2 Mechanism of `swtch()`

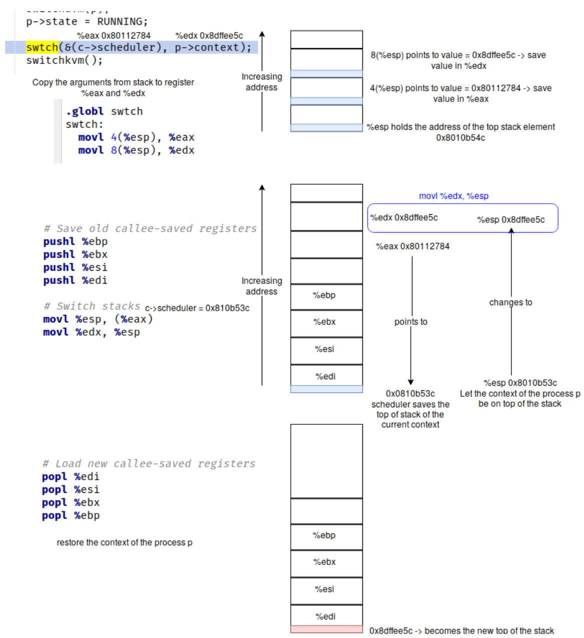
Note that only the caller saved registers and the `EIP` is present on the kernel stack when `swtch` is called. So what must switch do?

- Push remaining (callee save) registers on old kernel stack
- Save pointer to this context in old process PCB(for which we use the first argument of context** type as explained above)
- Switch ESP from old kernel stack to new kernel stack
- ESP now points to saved context of new process
- Pop callee-save registers from new stack(which restores some of the context)

- Return from function call (pops return address, caller save registers which completes restoring the context)

4.3.3 xv6 assembly code for `swtch()`

The assembly code explaination given below is redundant if you understand the high level of what is happening but helps if you are trying to understand specifics



- When swtch function call is made, old kernel stack has return address (eip) and arguments to swtch (address of old context pointer, new context pointer)
- Store address of old context pointer into eax
- Store value of new context pointer into edx
- Push callee save registers on kernel stack of old process
- Top of stack esp now points to complete context structure of old process
- Update old context pointer (eax) to point to updated context
- Switch stacks: Copy new context pointer from edx to esp
- Pop registers from new context structure
- Return from swtch in new process

Figure 2: Assembly code in xv6 for `swtch()`

4.3.4 `swtch()` for new processes

Our way of working assumes the existence of a context structure of a process. What if the process just started and has never run before (ie) a newly forked process? The kernel stack of such new processes are setup[40] in such a way that the EIP of function where the process starts from is saved in the context structure to mimic that the process was switched out from where we want to resume in kernel mode(this location is just after the fork wherever our PC is pointing to). Apart from this the trap frame is copied from the parent(`eax` or return value alone is changed for the `fork()`) so it has the proper

[40]They are artificially setup with appropriate context sturcture and trap frame

register context to resume in user mode just after wherever `fork()` call was made. Process resumes execution in kernel mode and it returns from trap to user space(to the instruction right after the `fork()`).

allocproc: This is the function that creates a new *struct proc* for our newly forked process. It finds an unused entry in the ptable and marks it as an embryo and changes it to runnable after the process completion is well "completed". It also allocates a **PID** for this process along with new memory (ie) a kernel stack and also a stack pointer pointing to the bottom of this stack. What else?

- Leave space for trapframe (copied from parent)
- after that pushes return address of "trapret"
- Push context structure, with `eip` pointing to function "forkret". This is done since, when new process is scheduled, it begins execution at forkret, then returns to trapret, then returns from trap to userspace continuing to execute instructions after fork
- Thus we have hand-crafted kernel stack to make it look like process had a trap and context switch basically lying to the scheduler to make it like look like any process which was context switched in the past. So the OS scheduler can schedule it as it does any other process

Lecture 9

4.4 OS Scheduling Policy

Now that we have understood the mechanism of the OS scheduler let's see how the scheduler chooses which amongnst the ready processes it should run. An important point to note is that only when a process is in kernel mode for a trap a context switch happens. However not on every trap does the scheduler perform a context switch The first classification among schedulers as we saw before are pre-emptive and non-premptive schedulers.

- **Pre-emptive:** Performs *involuntary* context switches using timer interrupts[41] even if the process is in ready/runnable mode.
- **Non-preemptive:** Performs only voluntary context switches when a process gives up the control via a trap.

What are the goals of a scheduling policy?

- Maximise utilization: Efficient use of CPU hardware so that the throughput of the CPU is maximised
- Minimize the time it takes for a CPU to complete a process. It is also called as

[41] Refer 3.3

turnaround or completion time

- Minimize time it takes for the process to start running after it is ready to run which is called response time[42]
- Fairness is to be maintained (ie) all processes get a fair share of CPU (account for priorities also)
- We also want to minimize overhead. That is we don't want the scheduling itself to take too much processing power/time. We can't have unnecessarily large number of context switches (since it takes ∼1 microsecond to switch) or have too much time needed to take a decision even with larger number of processes to be scheduled

4.5 Different scheduling policies

1. **FIFO(First in first out):**
   We put all our processes in a queue and then we run them one after another. Note that FIFO is non-preemptive (ie) a processes runs to termination or till it is blocked. In case of a blocked syscall, after it is done with handling a blocking syscall we add it to the end of the process as a new process with a new context. When it is added back it counts as a fresh CPU burst [43]

Process	**CPU time needed**	**Arrives at end of**	**Execution time**
P1	5	0	1-5
P2	3	1	6-8
P3	2	3	9-10

Table 1: FIFO

   The disadvantages with FIFO is the convey effect (ie) processes which are shorter tend to get stuck behind much longer ones resulting in unnecessary increase in the turnaround time due to longer waiting time

2. **Shortest Job First(SJF):**
   Asssuming that the scheduler has the knowledge on the CPU burst of all processes (time it runs for one continuous stretch). Now we can pick the process with the shortest burst time first and run it and continue doing so. This can be done by maintaining all the processes in a heap ordered by burst time needed. This can be proved to be the optimal solution when we assume all the processes arrive at the same time. However this is an unrealistic assumption. Also due to its non-preemptive nature short jobs can still get stuck behind longer ones (different arrival times).

[42] Very important for processes that are highly interactive and constantly take input from users
[43] A CPU burst is the time for which a process runs in a continuous stretch

Process	CPU time needed	Arrives at end of	Execution time
P1	5	0	1-5
P2	3	1	8-10
P3	2	3	6-7

Table 2: Shortest Job First

3. **Shortest remaining time First(SRTF):** This is a preemptive version of **SJF**. A newly arrived process can preempt[44] a running process, if CPU burst of new process is shorter than remaining time of the running process. This avoids the problem of shorter process getting stuck behind longer ones if they don't arrive at the same time.

Process	CPU time needed	Arrives at end of	Execution time
P1	5	0	1,*preempted* then 7-10
P2	3	1	2-4
P3	2	3	5-6

Table 3: Shortest Remaining Time First

However we are still assuming knowledge on the run times of processes before they arrive (unrealistic assumption).

4. **Round Robin(RR):**
   This is the first practical policy we are seeing. We fix a time slice (called a quantum). The quantum is a multiple of the length of the timer interrupt. So when the time interrupt happens the scheduler checks if the process has already run for the allocated time slice. If that is not the case, control is simply handed over back to the process. If not so, control is handed to another runnable process by context switching into it. Thus the timer interrupt is used to enforce periodic scheduling. This policy is good for maintaining both response time and fairness, however it is not the most efficient for turnaround time. `xv6` uses this policy

   What effect does time slice's length have?

   - A small time slice makes it inefficient since too many context switch happen and the time spent to context switch isn't amortized properly
   - If the time slice is too big there is no responsiveness as one process hoards the cpu for a longer time

5. **Weighted Fair Queueing(WFQ):**
   This is a modified version of Round Robin policy. We set weights to every process and give preferences according to it. The weight setting is done by both user and the OS itself to different extents. The time slice is for a given a process is

[44] terminate this process and make CPU context switch to itself

proportional to this weight. Real life schedulers may not be able to maintian the time slice enforcement accurately. Think about cases where the time interrupt doesn't align with time share or a process is blocked before its slice. A practical modification to accomodate such things is to keep track of the run time of a process and switch in the process which has used the least fraction of its fair share. This compensates for deficiences/excess in the running time.

6. **CFS scheduler**:
   The Completely Fair Scheduler used in linux is a variant of WFQ. It uses red-black to keep track of who all runs and keeps track of who all have used up least amount of their fair share. However it is impossible to be fully fair whilst being reasonable but this policy is close to optimal fairness.

7. **Multi-level feedback queue (MLFQ):**
   This is another practical algorithm with realistic assumptions. We would prefer the property of the SJF which gives priority to smaller jobs. However we can't assume we know the running time before. We also want to ensure that the response time is lesser as we see in Round robin.

   We have multiple queues corresponding to multiple priorities. Scheduling of processes happen first for higher priority queues and within a queue Round Robin is used for scheduling. If a process has used up more of its fair share of CPU we decay it's priority and push it down. When we add a process we add it to the high priority queue. This helps us have an idea on which process is running for a longer time and which ones haven't. We run and finish up shorted processes quickly (Like I/O) even without prior knowledge of CPU bursts. Periodically reset the priority level to make sure even longer process gets some chance to run.

Lecture 10

4.6 Multicore Scheduling

Scheduling decision is supposed to be made seperately for each core. Do we bind a process to a particular CPU core always or do we let a process, run on any CPU core that is free? Making sure that the process runs on the same core as much as possible is better due to improved CPU cache performance. However balancing is important, if the core is highly overloaded then we ensure a process runs on a different free core to not waste time.

Overall however having a single queue which assigns the ready process to the first free CPU out of all of the multiple CPU's present is more efficient than having multiple queues for each CPU and choosing to put a process in one of these queues. This is due to the high cost and thus inability to switch queues once a process is assigned a queue in the multiqueue case.

5 Inter Process Communication

Why do we need IPC[45]? In real life systems it is not ideal or feasible to maintain all of the functionality in just a single process. Rather we modularise code (which can be in different languages, frameworks and written by different teams of people) and need some method for them to work along with each other.

Processes in a system needn't share memory with each other by default. However they still need the ability to communicate. This is done via IPC mechanisms, which are available via the OS whcih provides several syscalls enabling the exchange of information between processes. The communication can also be done by two processes agreeing to share a particular portion of the memory image to enable communication. Communication can also be done by having a particular memory area (buffer) specifically meant for communication. It is important to note that it is an extremely dumb idea to just let two processes have access to the entire memory images of each other since this destroys the ability to maintain isolation of processes.

5.1 Sockets:

Sockets are an abstraction to enable communication between processes. Each process opens its own socket and two sockets of two processes can be connected. The process

[45]Interprocess Communication

to open the socket first is the *server* and the process which connects its socket to the server is the *client*. A socket is a way of bidirectional communication, thus a process can write into its socket which can be read by other processes and vice versa. Processes communicating using socktes can be in the same machine or on different machines. While communicating on the same machine the buffer[46] is present in OS memory. For communicating across machines messages are sent on the network. *Unix Domain sockets* are used to communicate between processes on the same machine. *Internet sockets* are used to communicate between processes on different machines.

The client needs some information about the process whose socket it wants to connect to (server). Thus every socket needs a unique number to identify itself. This `ID` for a Internet socket is the IP address and port number. Local sockets are identified a unique *pathname* identifying it in the local machine. Note that the local socket isn't exactly a file but it has a pathname that only serves the purpose of identying it uniquely. The server needs to publicise its identifier (IP or pathname) for the client process to know where to connect to.

Connection-based sockets are those when a client and server are explicitly connected to each other. After this connection they can only talk to each other. So there is no need to explicitly mention which socket we want to communicate to. *Connection-less sockets* are those who can communicate with multiple processes at the same time. Thus we need to mention the address of the socket we want to communicate with.

5.1.1 `socket()` syscall

The `socket()` syscall is used to create a socket. It takes the type of socket as an argument and returns the file descriptor of the socket to the process. This file descriptor is used for all operations on the socket by this process. A socket can optionally bind to the address (pathname or IP address/port number) using `bind()` system call. Server socket binding means that the socket is given a global address which can be used by other processes to communicate to this socket. Note that binding is compulsory for server sockets who need to publicise this binding address to enable clients to talk to them. The `close()` system call closes a socket.

How do processes use **sockets** to communicate?

- **sendto():** is used to send a message from one socket to another connection-less socket. It takes its own socket's file descriptor, the message to be sent, the other socket's address as arguments
- **recvfrom():** is used to recieve a message from a socket. It takes the its socket's file descriptor, a message buffer to copy the message into and a sockets address structure into which the recieving socket's global address is filled. Now this process

[46] intermediate storage which stores message to be sent

can use the address it just obtained to communicate back to the process which sent it a message.

Lecture 11

Connecting sockets

Connection-oriented sockets need to be explicitly connected for transfer of messages. After a server binds to a well-known address[47], it uses the `listen()` syscall to listen for "requests to connect" from clients who want to connect to server. Clients use the `connect()` to send a request to connect to the server. The server uses `accept()` syscall to accept a request from the client to establish a connection. Once an `accept()` syscall is sucessfull then a connection is established between the client and server.

The `accept()` syscall takes the file descriptor of a socket as an argument. This socket's address is what clients use to send a "request to connect". Once the `accept()` syscall is succesful it returns a **new file descriptor** to a socket which is connected to the client.

Hence it is important to note that there is a *listening socket* corresponding to whose address requests are send to connect and there are (possibly multiple) *connected sockets* explicitly connected to a client socket to enable communication between two unique processes. The `accept()` syscall will block the server till is recieves a request to connect from a client. Similarly, the `connect()` syscall blocks the client till the server accepts the requests.

After a client connects to a server, the *pair of sockets* are used to exchange data. Again let's reassert that the *connected socket* is the one which is used to transfer messages not the *listen socket*. Syscall `send/write` to send a message which only goes to the connected socket. Similarly syscall `recv()/read()` is used to recieve a message on connected socket. The arguments given to send/recv are the socket fd, message buffer, buffer length and (optional) flags. The return value is number of bytes read/written or error number. We need not specify socket address on every message, as they connected already and only communicate with each other using that particular socket.

In any kind of real system multiple connected and connection-less sockets connect multiple processes in several ways forming a complex network of processes all working with one another to achieve some task.

[47] gets assigned a proper global address

5.2 Message queues

Message queues are another method used for exchanging messages between processes. We open a connection to a message queue identified by a "key", and get a file descriptor to it. The sender can open a connection to the message queue and send a message. The reciever opens a connection to the message queue and recieves the message later on. The message queue acts as a buffer till the message is retrieved by a reciever.

There are three main syscalls associated with a message queue:

- `msgget()`: syscall to get the id of a message queue. The parameters it takes are the key of the message queue . It returns the msg_queue identifier on success and -1 on failure.
- `msgsnd()`: The parameters it takes are the msg_queue identifier and the message to be sent and its size. It returns 0 on success and -1 on failure.
- `msgrcv()`: The parameters it takes are the msg_queue identifier, size of the message and the buffer to write the message into. It returns the number of bytes recieved on success and -1 on failure.

There can be several different message queues all with their own unique "keys". Also note that multiple processes can use the same message queue if they all have access to its key. Thus we need to be careful about which process's message is being recieved when `msgrcv()` is used.

5.3 Pipes

A pipe is a unidirectional FIFO channel into which writing is done on one end and reading from another end. The syscall `pipe()` is used to create a unidirectional pipe. It returns two file descriptors for the end points (one for write, other for read). A pointer to a `struct` of a pair of numbers are passed as a parameters of the pipe. Once a pipe is succesful this `struct` has both file descriptors. Why do we have access to both descriptors? This is because Pipes are frequently used for communication between a parent and a child. The data written into a pipe is stored in the buffer of the pipe channel until they are read. Obviously, bi-directional communication between two processes needs two pipes.

Anonymous pipes are only available to communicate between process and its children. The pipe file descriptor is available to both the parent and the child.

Named pipes are used by two unrelated processes to talk to each other. The pipe is opened with a pathname which is accessible across processes. One of the processes has access to the read end of a pipe, while the other other process opens the write end.

Buffer Mechanism: When we want to send a message the general mechanism stays the same across all these sycalls. We write the message into a temporary space called the buffer. This buffer is usually some memory location inside OS not directly accessible by the user. In all of these system calls the buffer maintains the messages in a FIFO order. When the buffer is read the message which was in it and was read is cleared. Sender usually can't be blocked by `send()` syscall unless the temporary buffer is full. Similarly reciever can be blocked if temporary buffer is empty. We can also customize the sycalls to not block the process but return some error code for us to handle to error manually.

5.4 Shared memory

Processes in a system do not share any memory unless mentioned otherwise. Child process *copies* the memory of the parent but doesn't share the memory image. Shared memory means that the same memory image appears in multiple processes. Each memory *segment* shared is identified by a unique key. A process can request to map/attach this shared memory segment into its own memory image using this identifier key.

Sharing of memory is a risky and volatile task. The processes thus need extra mechanisms to know for example, if the memory has been modified. This enables them to coordinate with one another

Lecture 12

Part III

Memory Management

Abstraction of Address space

Every process assumes it has access to a large space of memory from address 0 to a max value. The maximum value depends on the available number of bits for the memory space. The virtual address sapce contains all the code/data that a process can access. Addresses in CPU registers point to these virtual addresses.

However this is not how memory is actually stored. The OS maintains this illusion of virtualized address spaces by having a map from every virtual address to a physical address. When the CPU accesses memory using an address (virtual address) the Memory Management Unit (MMU) converts it to a real address and gives back the data at the physical memory corresponding to the virtual address. OS allocates physical memory to a process, it has the translation information provides it to MMU.

Why do we need the MMU? Why not the OS? This is primarily since we don't want to trap into the kernel for every memory access. The OS does the background work of allocating memory for processes and maintaing the virtual address → physical address mapping and simply passes on the information to the MMU. The MMU deals with translation of virtual memory to real memory. During Bootup the OS starts on the MMU and starts giving it translation information.

Why virtual address?

The real view of memory is messy and it is not easy to manage manually. In earlier systems there was only one process and its own code. Nowadays we have multiple parallely running processes which all do timesharing on the same CPU. Memory is also allocated in a non-contiguos manner, but we still want to maintain the illusion of memory being continuosly allocated. Thus virtualization has become a neccesity in modern systems. The CPU **only** uses the virtual addresses to perform any action.[48]

[48]For example addresses of a pointer obtained from `malloc()` is the virtual address

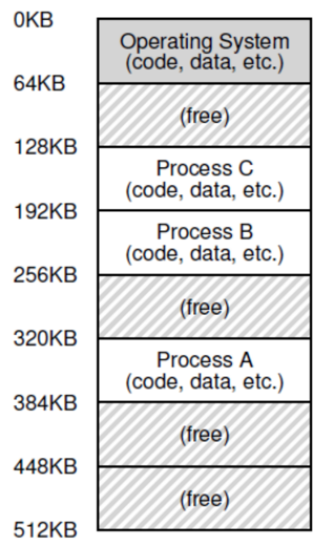


Figure 3: Memory allocation for three processes

1 Base and bound

This is the simplest form of memory management. It is not used in modern systems but we will understand it to get some insights. The memory image from [0,N] is placed at memory address base B. That is every virtual address X is stored at physical address B + X. There is also a bound associated with this base which tells us the maximal address for the memory allocated in this memory image.

Simple example: Let's take a small porgram as given hello

```
void func(){
    int x = 300;
    x = x + 3;
}
```

This code compiles to give instructions.

Suppose the OS places the entire memory image in one chunk starting at physical address 32kb, then the OS indicates base and bound to MMU. MMU performs this translation from Virtual adress to physical address. But what if some garbage address/ out of bound address is given? Then the MMU raises a trap to call the OS to handle this situation. This is another type of hardware level trap (program fault). In this case int n is inserted into the stream of instructions run by the CPU.

2 OS vs MMU

The OS is the one who allocates the memory and builds the translation information of a process. However the MMU is the one who translates virtual address to a physical address. Note that once the user starts running the OS is out of the picture unless there is a trap.

When a context switch happens the OS changes the translation information in the MMU to enable it to perform translating addresses corresponding to this new process [49]. As told before when the CPU has an instruction which wants to access memory the MMU translates it to a physical address which can be used to access the data.

3 Segmenation

It is an upgradation of base and bound. The program is split into several segments (Code, data, stack) and each of them is placed at a different memory location (base). Each of these bases also have a bound associated with them. When the address exceeds the bound of a segment we obtain a *segmenation fault*[50] . Thus when we context switch into a process the OS updates the MMU with all the necessary bases and bounds. Each virtual address is hence a segment identifier bit sequence concatenated with an offset for that particular byte.

But this is still inefficient due to the variable sizes of these segments. This also results in *fragmentation* where there is some unusable space between two segments.

4 Paging

Paging is the mostly widely used system of memory Management today. We divide our virtual address space into fixed size pages. Each one of these pages is assigned a free **physical frame**[51]. The memory allocation for each page is done at a fixed size (eg. 4kb). Since the pages are tightly packed we have no wasteage of size between pages (ie) no *external fragmentation*. However inside the pages we could have some wasteage of space (ie) *internal fragmentation*.

What is the optimal page size? Why 4kb?

- *Large page size* → larger amount of wastage inside individual pages.
- *Small page size* → OS has to keep track of huge number of pages which creates a

[49] same virtual address means different physical address in different processes
[50] *sigh*
[51] Just the physical memory associated with a single page

large overhead.[52]

[52] Large processing power is needed to maintain page table

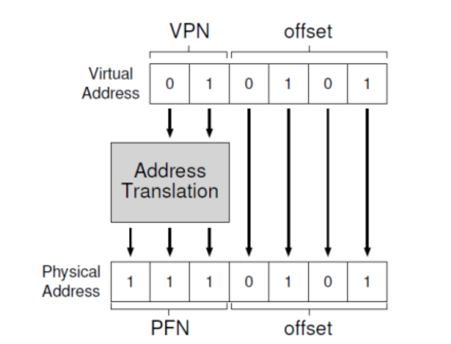


Figure 4: Visual representation of address translation

Lecture 13

5 Page table

What is a page table?

The page table is a per process structure (ie) each process has its own page table. This table helps to translate virtual addresses to real ones. It stores frame numbers for all the pages used by the process in an array format. Table[i] gives the physical frame number of the ith virtual page of the process. Thus the ith entry of the page table corresponds to the physical address of the starting byte of the ith page. The page table is stored in a part of a OS memory (stored in PCB). The MMU has access to the page tables of current processes and it uses it for translation.

Given page table of a process, how is physical address obtained from virtual address?

When you have the virtual address you can find which page your data is in. This can be done by (Virtual Address / Page size) and the offset is (Virtual Address % Page size). That is the most significant digits give the page number and the least significant ones give the offset.

Hence the most significant digits give the Virtual Page Number (VPN) and least significant bytes give the offset. The VPN and the page table can be used to get the Physical Frame Number. Note that Page_Table_array[VPN] = PFN[53]. Obviously the offset to get the byte we want is same in both the Physical frame and the virtual page.

[53]Physical Frame Number

Translation Lookaside Buffer(TLB)

An important point to note with paging is that one memory access has become two accesses now. One access to the page table and the one to the actual byte we want. This results in a significant slow down since memory accesses are very slow. This is the overhead associated with paging

To reduce this overhead the MMU caches the most recent translations in translation lookaside buffer(TLB)[54]. TLB caches only the **page table entries** [55]. If we have a hit, we fetch the memory contents of the given address in one access since we already have the page table entry in the cache. If there is a miss we get the memory from RAM and place it in TLB for future accesses after which we use the translated address and then again use this page table entry to translate the virtual address and get the memory we wanted.

Interestingly, during a context switch the TLB is flushed. This is due to the fact that the MMU needs to translate the Page table for every processes in a different way (different page table entries).

Thus the TLB is very important and *needs* to have a high hit rate. It works parallely with the CPU cache to further optimize memory accesses.

[54] It is a fully associated cache

[55] the physical data in 32/64 bit addresses are cached by the actual cpu cache

Lecture 14

Page table entries

As we saw before the page table is an array of page table entries. There are a large number of virutal pages available but each process only uses a few. Thus to indicate that a virtual page is not used each page table entry of a page table has a `valid` bit. If this bit is 0 that entry is invalid and if it is 1 then it is used. The invalid page table entries in a process can be used by some other process at the same time. So not being able to access these entries is extremely important for isolation purposes.

We also have other similar **permission bits** which we will look at later.

The number of bits in the virtual address corresponding to the offset of a particular page is dependent on the size of the page. If the size of the page is say x bytes then the number of bits required for offset is $\log_2 x$[56]. The remaining bits of the address are used to indicate the Virtual Page Number. If the Virtual Page number does not correspond to a valid entry of the page table then the MMU raises a trap (it is an out of bounds access). Thus the most significant bits are used as an index to get the Physical frame address from the page table and the remaining bits are the offset to get the actual memory byte we want to access.

In a 32 bit (4GB) system usually, a page is 4kb large. There are (1024 * 1024 pages (ie) around a million pages) in a page table.

Inner and outer page table

The page table is of size 1MB (All 32-bit addresses are stored). But we know that the size of the physical frame which is the size of an available contiguos memory chunk is 4kb. So how is the page table stored?. Interestingly the page table is divided into 1024 inner page tables which have their start addresses in another ”outer” page table corresponding to each inner page table. To reiterate, each of those 1024 byte sized chunks of the page table is an inner page table. This outer page table has a size of 1kb (Since each page table entry covers 1024 bytes corresponding to one inner page table). This solves our limitation in the size of our physical frame. The outer page table is as powerful as the normal page table which comes into use.

Say we have a process which has 2k pages needed for code and data and after that it has 6k unallocated pages and then 1k unallocated pages (for the stack) followed by

[56] 12 bits in 4kb case

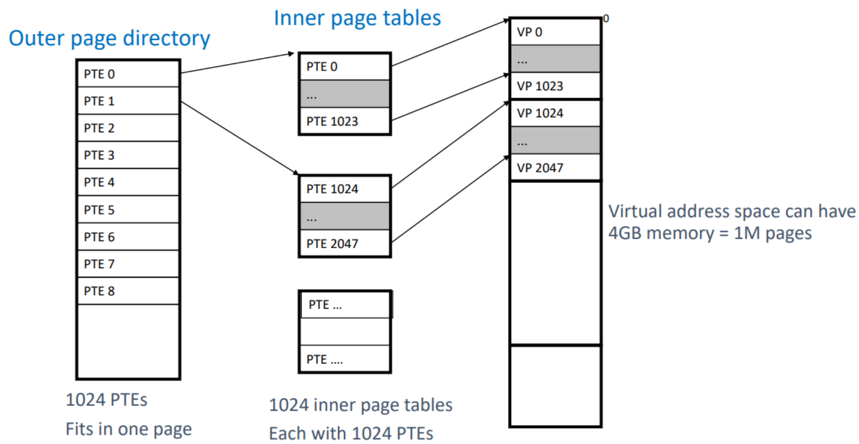


Figure 5: Mapping from outer page table to inner page table

the stack itself[57]. Here if we had entries for all pages in their corresponding inner page table with valid/invalid bits it is easy to see that it is a lot of work since we need to bother about 1024 * 1024 entries. A smart solution is, rather than worrying about the inner pages tables at all just set an entry of the outer page table to be invalid iff all the entries in the inner page table corresponding to it are not used. Thus instead of setting 1024 entries of the inner page table to invalid we directly make the outer page table entry corresponding to all of them invalid. Hence we need not even bother creating/maintaing inner page tables with no valid entries.

As a conclusion, in our paging system there are 4GB bytes which are split into 1M pages of size 4kb each. There are 1M entries in the page table corresponding to each virtual page. The 1M entires of the page table are divided into 1024 inner page tables. The inner page table has the physical frame address corresponding to a virtual page. Each of these inner page tables is mapped to an entry in the outer page table thus the outer page table has 1024 entries.

5.1 Multi-level Page Tables

How are addresses translated in a two level Page table? As we saw before the most significant bits denote the page number. Now we can even split these most significant bits into two parts where the more significant bits represent the index of the outer page table whereas the one less significant represent the offset within the inner page table to get the appropriate page table entry. Thus to get the actual physical address the MMU walks along the page table arrays. This is why the MMU accessing the page table is called walking the table.

[57]Note that

MMU stores starting, physical address[58] of the page table in a single level page table or specifically the outer page table incase of multi-level paging. This is stored in the page table base register of the MMU.

What if splitting into two levels isn't enough? This process of making page tables for page tables can be continued up until the final/outermost page table can fit within a single page. However this comes with the obvious overhead of making each page table walk longer and thus more time consuming.

[58]Since we need the actual access to the page table

Lecture 15

As an example consider a 48-bit cpu architecture with pages of 4KB size. Thus the number of pages will be $\frac{\text{Total number of bits}}{\text{Size of a page in bits}}$ (ie) $\frac{2^{48}}{2^{12}}$ so 2^{36} pages. So we need an array to store the corresponding physical frame address for each virtual page that is 2^{36} entries. Now obviously this innermost page table isn't less than the size of one page which is 2^{12}. Hence it can't be stored continuosly in a single page and we need to find futher divide the table into page table sized chunks and map each of them to a physical frame address. So lets do make a page table for the inner most page table. From similar caluculations the size of this new table will be $\frac{2^{36}}{2^{12}}$ which is 2^{24}. This again isn't small enough to fit in one page. Let's divide again. Now we have $\frac{2^{24}}{2^{12}}$ which is 2^{12}. Finally this is exactly one page long. Thus we stop.

We ended up having 4 levels of page tables here. The size of the innermost page table entry is 2^{36}, so 4-byte (32 bit) addresses are not enough to identify each entry uniquely thus we have 8-byte addresss. The most significant 36 bits help us get the physical frame address of each page and the remaining 9 bits give the offset. Now among these 36 bits the first 24 tell us the physical frame address of the beginning of a chunk into which the inner most page table is divided and the 9 bits give us the offset among one such chunk. Reiterating that the innermost page table is divided into 2^{24} chunks and each of this is mapped to a page. We can continue in this way to index each element of the tables at every level when given a 48-bit address.

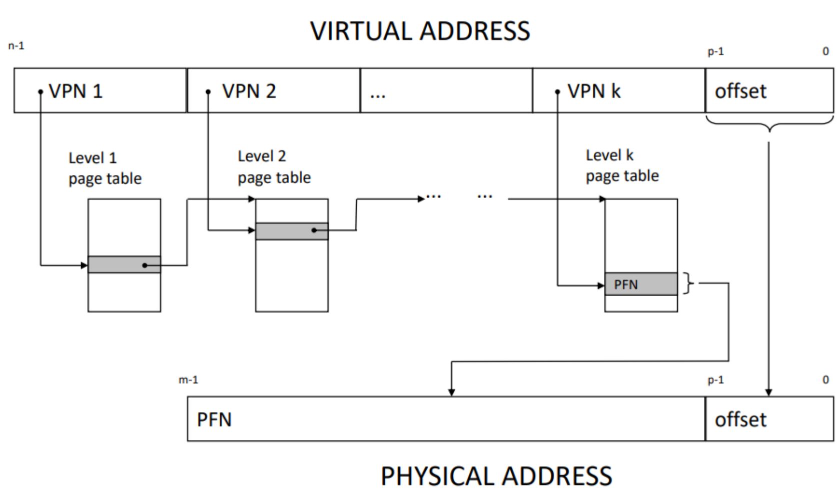


Figure 6: Translation from multi level page table to physical address

6 Virtual Addresses

It is important to note that every element of memory needed by a process has a virtual address associated with it. This is due to the fact that MMU can access elements only via virtual addresses. So every element like the code, data has a virtual page and a mapping to its physical address. Even libraries that we import or the OS is mapped to a virtual page by **each** process.

What virtual addresses are given to the OS,libraries which are not technically part of the code? We give such elements *high virtual address*. This is done to prevent accidental collisons in the virtual addresses assigned to these elements and other things like stack, code, data. Note that this does not mean each process has a copy of the OS, libraries. Rather each process has an entry in its page table and thus allocates virtual pages to these elements, however the physical addresses these virtual pages point to *remain the same.*

Also thinking the other way around anything inaccessible by a process is not part of the virtual address space.

7 OS is not a process

From the discussion above one important point to notice is that the OS isn't a seperate process with its own address space. The OS is mapped for each process to be a part of its virtual memory. This denotes a very important point established in the beginning of the course. **Each process thinks it owns the system.** Each process thinks it is the only one existing and that it has a monopoly over the system and that the OS code is a part of it.

When the Bootloader runs, it places the OS memory in the very beginning addreses of the ram. Thus the OS has high virtual address and low physical addresses.'

7.1 Permission bits

How is illegal OS access prevented? As we saw before we have permission `bits` which can make a page table entry invalid, make a page read(or write) only, make it accessible only during user mode etc. Thus the OS page table mappings have permission bits to allow access only in kernel mode. Similarly our code's pages are assigned read only privileges since we do not have to change instructions after compilation.

7.2 Memory management in xv6

xv6 has a two level page table system. It has 1024 inner page tables with 1024 entries corresponding to each of them in the outer page table. The physical address of the base of the outer page table is put in the CPU's cr3 register, which is used by the MMU during address translation.

The size of the virtual address space in xv6 is 4GB. The physical address space ranges from [0,PHYSTOP]. `PHYSTOP` is a macro to denote the maximum size of the usable physical address in xv6.

Lecture 16

8 Demand Paging

Now, think about if it is necessary to have all the pages in the main memory? This is not necessary as a process can't use them at once. This is also not possible with larger address spaces. Modern operating systems provide only virtual memory. That is not all logical pages are assigned physical frames. Even those that are, are not permanently kept in main memory. Some pages are stored in disk and brought to main memory *on demand*. This is called **Demand paging**

Swap space

This is the special area in the disk to hold pages that do not fit in DRAM. Pages are pushed to the swap space when the main memory (DRAM) is full. This is brought into memory when a page is accessed.

Recall valid bits. If a page is invalid we don't store any frame number for them. However for valid pages we either store the physical frame number or a disk address space. Now we also need to denote if an entry contains a disk addreses or a physical frame address. For this purpose we have a permission[59] bit called the present bit. If the present bit is 1 then the memory is in DRAM else it is in disk space.

Note that in mulitlevel paging we can still have some of the pages in DRAM and some of them in disk. From the above discussion we can see that the OS *overcommits* pages (ie) tells the process it has access to a lot more pages than it actually does and it takes care of giving these pages by getting them from disk **on demand**

[59] Refer 7.1

Valid	Present	What it represents
1	1	The page is valid and in DRAM
1	0	The page is valid and is in disk
0	1	Stupid, doesn't happen
0	0	Invalid page

Table 4: Table to show valid, present bit configurations

8.1 Page faults

When the MMU walks the page table to translate an address to a physical address, the various permission bits of the page table are also examined. If the MMU notices some unexpected behaviour it raises a trap and gives control to the OS.

Possible errors raised by MMU:

1. **Segmenation fault:** Trying to dereference some address of an unallocated virtual address. That is accessing memory which doesn't belong to the program. The OS kills the process in this case.
2. **Illegal action:** The code performs some illegal action like attempt to access a page table that is not valid, or an attempt to modify a read-only page which obviously terminates the process.
3. **Page allocation in DRAM:** When a valid bit is set but present bit is not set (page in disk), then OS allocates a page in the DRAM, updates the page table and then gives control back to the process. The process won't terminate in this case since the error was on the OS's side. Rather after handling the allocation process continues to run.

8.2 Reclaiming Pages

The OS keeps track of all the free pages that are not being used so that they can be used during a page fault [60]. If there is no free page available for the OS, then it evicts a `vicitm page` and then pushes it from memory to the swap space (page table is updated). After this the page just freed up in the DRAM is allocated for the page which raised the fault.

We need to have a policy to determine how to choose a victim page. Note that the victim page can be in the same process or a different process.

[60] The third type of fault

File backed, anonymous, dirty pages

Pages in the memory image of a process are of two types.

- **File-backed pages:** This contains data that comes from a file on the disk. (e.g Page with the executable code). So even if the memory in the DRAM is lost we have the file saved in the disk
- **Anonymous Pages:** These are pages containing data which are not stored anywhere. Some examples are the stack, heap. So if these are removed from DRAM they are gone forever.

File backed pages are further classified as:

- **Dirty Pages:** These are file backed pages whose contents is different from their copy on the disk.
- **Non-Dirty Page:** Pages which are identical to their copy on the disk.

This classification is kept track by the page table, again using permission bits [61]. Why do we need this classification? We need to deal with page faults differently depending on which type of page it is. When we reclaim memory from a victim page, we need to copy its content since it has changed compared to the file from which it came iff it is a dirty page. Else we can simply delete the page.

When we allocate memory during a page fault, the free memory frame must be intitialized with the content from the disk for a file backed and non-empty anonymous pages. For empty anonymous pages we just give an empty page for it.

Note that a process maybe blocked quite a few times for disk I/O during a page fault. The average memory access time shoots up greatly if we have too many page faults.

TLB and address translation

Just a recap of how addresses are dealt with. The CPU only had virtual addresses which it deals with. The processor requests the corresponding physical address from the MMU which looks in the TLB for an entry. If we have a hit/match for that address, the physical address is directly returned. Else the page table is walked and the entry corresponding to this Virtual address. This entry is used to get the address from memory. This address is also populated into the TLB for future use.

[61] eg. the dirty bit

Lecture 17

9 Thrashing

We have several causes for an application to slow down but a page fault is an important reason for a massive decrease in efficiency. So it is essential that we have a large enough working memory to avoid page faults.

Every process has a working set which is the number of frequently used pages in the memory image. The actual set of pages can change from time to time depending on what process is running. This is usually smaller than the total virtual memory of the process. If we allocate memory lesser than the size of the working set, we have frequent page faults.

Thrashing is this scenario where the system spends too much time servicing page faults and swapping back and forth from disk, and too little time doing application based work. A fun example to think of to understand this is to assume that you want to study but you keep switching subjects every minute. Most of your time goes off in taking and putting back books.

The way to deal with Thrashing is to create efficient processes that don't use that many pages at the same time and also for the OS to terminate some processes.

10 Page replacement policies

We need to decide which page is our victim page we want to replace when the DRAM is full. Hence we need a replacement poilicy

Some of them are

- **Optimal policy:** We have 4 pages which the process needs but the process has space only for 3 pages in the DRAM. We assume that we can "look" into the future and then evict the page which is not needed for the longest amount of time in the future. This ensures minimal number of misses. Obviously it is not implementable since we are not oracles who can look into the future.

- **FIFO:** We evict the page which was first put in the DRAM (ie) first page which was put in chronologically and then we evict that page. Bad idea since pages first put in are generally reused often

  NOTE: Beleady's anomalies are very weird examples of some policies doing worse on some inputs when they have more resources.

- **LRU:** The page which was accessed the farthest time in the past is evicted. That is if a page was accessed recently it is kept .The logic behind the policy is intuitive since a page accessed recently is much more likely to be accessed again.

  LRU *seems* to be very easy to implement. Diving into the details makes the problems a lot more apprarent. Something as simple as maintaining a time stamp is difficult since time is not something that is consitently maintained in the system. Also the question of where you will store this information also comes up. The best part is OS doesn't even know which pages were translated/accessed since it's last trap so this information cannot be updated by it.

  Thus implementing a **perfect LRU** is almost impossible. Rather we maintain a very crude version of LRU.

  The way LRU is actually implemented is by maintaining an access bit which is set to 1 when MMU is accessing it. When a trap to the OS occurs for a page replacement, the OS just picks a page which has a 0 value at the access bit meaning it was not accessed recently and replaces it.

  The OS also periodically wipes the access bits to 0 periodically. The OS also avoids choosing dirty pages as victim pages since we need more resources to write into disk for them.