OS CS219 Notes

January 30, 2024

Lecture 1

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

[1] process is defined later

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

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

[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

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

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]

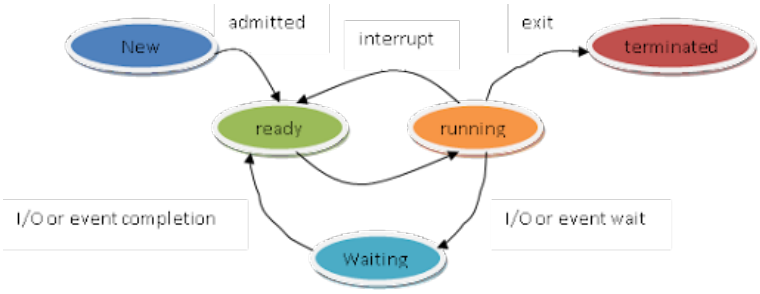


Figure 1: Figure to show process state switching

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]

[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

- **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
- **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

[10] refer Section 3

[11] Stores the details about one process

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

Lecture 3

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[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:

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

[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

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]

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[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

[16]Done using INT in assembly
[17]Lottt more detail on this furthers
[18]the heap,stack,instructions,data is the memory image of a process

Lecture 4

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 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]

6.3 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 page table 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

7 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

8 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

8.1 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

8.2 Trap Frame in xv6 and trap handling

- A data structure called the Trap frame is pushed onto the kernal 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 harware 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 "n" argument in 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

8.3 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

9 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]

9.1 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 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]

[30]Refer 8.3

[31]Refer 8

[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

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

9.2 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.

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

could have typos, will be corrected after quiz

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)

[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

stores the current top. Thus we can access the full of the stack this way. We can also very easily access arguments since they are on the base of the stack

9.3 `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.

9.3.1 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.

9.3.2 Mechanism of `swtch()`

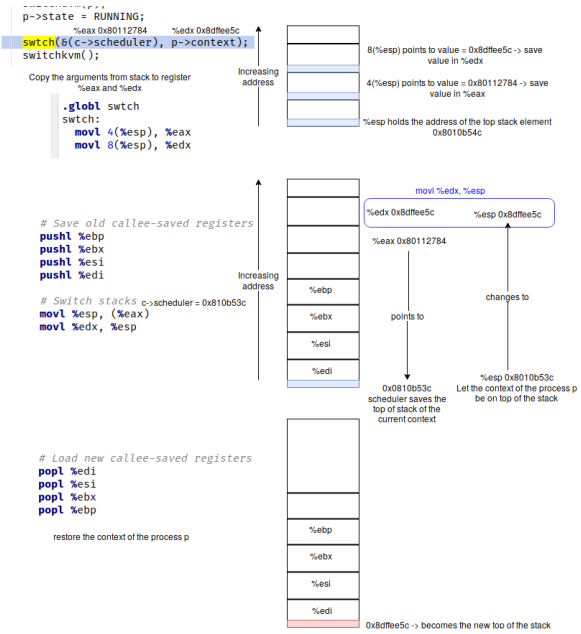
Note that only the caller saved registers and the `EIP` is present on the kernel stack when `swtch` is called. So what must switch do?

- 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)

9.3.3 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()`

9.3.4 `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