Notes

The following markdown contains some of the important tasks and terms that I came across while doing the course:

Week 1 : Overview

Lecture Notes

- Process is represented by a data structure commonly known as process control block (PCB)
  - Linux → task_struct
  - gemOS → exec_context
- Program is persistent while process is volatile
  - Program is identified by an executable, process by a PID
- How does OS get the control of the CPU?
  - In short, the OS configures the hardware to get the control. (will revisit)
- How the OS knows which process is “ready”?
  - Why the process may not be ready?
  - A process may be “sleeping” or waiting for I/O. Every process is associated with a state i.e., ready, running, waiting (will revisit).
- What is the memory state of a process?
- How memory state is saved and restored?
  - Memory itself virtualized. PCB + CPU registers maintain state (will revisit)
- ```
  struct user_regs{
          u64 rip; // PC
          u64 r15 - r8;
          u64 rax, rbx, rcx, rdx, rsi, rdi;
          u64 rsp; // stack pointer
          u64 rbp; // base pointer
  };
  ```

OSTEP

Week 2: Process API

Lecture Notes

- When we execute “a.out” on a shell a **process control block (PCB)** is created
- getpid()
- fork()
  - fork( ) creates a new process; a duplicate of calling process
  - On success, fork
    - Returns PID of child process to the caller (parent)
  ```

        - Returns 0 to the child
    - PC is next instruction after fork( ) syscall, for both parent and child

- exec()

    - Replace the calling process by a new executable
        - Code, data etc. are replaced by the new process
        - **Usually, open files remain open**
    - Calling processes is replaced by a new executable
    - PID remains the same
    - On return, new executable starts execution
    - PC is loaded with the starting address of the newly loaded binary

- **wait()**: The wait system call makes the parent wait for child process to exit.

    - In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state

    - The wait() system call suspends execution of the calling thread until one of its children terminates.

    - on success, returns the process ID of the terminated child; on error, -1 is returned.

    - A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal.

    - **Note**: It waits for only 1 child

```
pid_t child_pid, wpid;
int status = 0;

//Father code (before child processes start)

for (int id=0; id<n; id++) {
    if ((child_pid = fork()) == 0) {
        //child code
        exit(0);
    }
}

while ((wpid = wait(&status)) > 0);
```

- **exit()**: On child exit( ), the wait() system call returns in parent

- What is the first user process?

    - In Unix systems, the first user process is called *init*

OSTEP Chapter 4 and 5

- Mechanisms are low-level methods or protocols that implement a needed piece of functionality. For example, we'll learn later how to implement a context switch
- Policies are algorithms for making some kind of decision within the OS. For example, given a number of possible programs to run on a CPU, which program should the OS run?
- a process is simply a running program; at any instant in time, we can summarize a process by taking an inventory of the different pieces of the system it accesses or affects during the course of its execution
- a PCB may include:
    - current memory state of the process
    - some important registers information
    - files and I/O information

> - In early (or simple) operating systems, the loading process is done eagerly, i.e., all at once before running the program; modern OSes perform the process lazily, i.e., by loading pieces of code or data only as they are needed during program execution.

- The OS will also likely initialize the stack with arguments; specifically, it will fill in the parameters to the main() function, i.e., argc and the argv array

- By loading the code and static data into memory, by creating and initializing a stack, and by doing other work as related to I/O setup, the OS has now (finally) set the stage for program execution. It thus has one last task: to start the program running at the entry point, namely main().

- Process States:

    - Running: In the running state, a process is running on a processor. This means it is executing instructions.
    - Ready: In the ready state, a process is ready to run but for some reason the OS has chosen not to run it at this given moment.
    - Blocked: In the blocked state, a process has performed some kind of operation that makes it not ready to run until some other event takes place. A common example: when a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.

- OS likely will keep some kind of process list for all processes that are ready and some additional information to track which process is currently running

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
        int eip;
        int esp;
        int ebx;
        int ecx;
        int edx;
        int esi;
        int edi;
        int ebp;
};
```

```
  // the different states a process can be in
  enum proc_state { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING,
ZOMBIE };

  // the information xv6 tracks about each process
  // including its register context and state
  struct proc {
          char *mem;     // Start of process memory
          uint sz;       // Size of process memory
          char *kstack;  // Bottom of kernel stack
                         // for this process
          enum proc_state state; // Process state
          int pid;       // Process ID
          struct proc *parent; // Parent process
          void *chan;    // If !zero, sleeping on chan
          int killed;    // If !zero, has been killed
          struct file *ofile[NOFILE]; // Open files
          struct inode *cwd; // Current directory
          struct context context; // Switch here to run process
          struct trapframe *tf; // Trap frame for the
          // current interrupt
  };
```

- Embryo: the process is currently being created
- **Zombie state**: Also, a process could be placed in a final state where it has exited but has not yet been cleaned up (in UNIX-based systems. This final state can be useful as it allows other processes (usually the parent that created the process) to examine the return code of the process and see if the just-finished process executed successfully. Issue a wait() call to indicate OS that it can clean up the child process
- fork():
    - the child isn't an exact copy. Specifically, although it now has its own copy of the address space (i.e., its own private memory), its own registers, its own PC, and so forth, the value it returns to the caller of fork() is different

> ***Read*** more about this diffence between the child and parent
> only the child process pid and parent_pid of the child process are diffent, also the return value from fork is diffent

- exec():
    - given the name of an executable, and some arguments, it loads code (and static data) from that executable and overwrites its current code segment (and current static data) with it; the heap and stack and other parts of the memory space of the program are re-initialized.
    - **Note: this is the case where the OS can write in the processes memory**

Remaining tasks:

- read the man pages of syscalls

- read about waitpid() - it specifies the pid of the child for which we have to wait till change in its state
- signals: interrupt and sleep and others

Week 3: File System API

Lecture Notes

- Processes identify files through a file handle a.k.a. file descriptors
- **open()**

```
int open (char *path, int flags, mode_t mode)
```

  - Access mode specified in flags : O_RDONLY, O_RDWR, O_WRONLY
  - Access permissions check performed by the OS
  - On success, a file descriptor (integer) is returned
  - If flags contain O_CREAT, mode specifies the file creation mode
  - **Per-process file descriptor table with pointer to a “file” object**
  - **file object → inode is many-to-one**

- **read() & write()**

```
ssize_t read (int fd, void *buf, size_t count);
```

  - fd → file handle
  - buf → user buffer as read destination
  - count → #of bytes to read
  - read ( ) returns #of bytes actually read, can be smaller than c
  - Note: Write is similar to read

- fd 0, 1, 2
  - 0 → STDIN
  - 1 → STDOUT
  - 2 → STDERR

- **lseek() & seek()**

```
off_t lseek(int fd, off_t offset, int whence);
```

  - fd → file handle
  - offset → target offset
  - whence → SEEK_SET, SEEK_CUR, SEEK_END
  - On success, returns offset from the starting of the file
  - lseek(fd, 100, SEEK_CUR) → forwards the file position by 100 bytes

- lseek(fd, 0, SEEK_END) → file pos at EOF, returns the file size

- **stat() & fstat()**

```
int stat(const char *path, struct stat *sbuf);
```

  - Returns the information about file/dir in the argument path
  - The information is filled up in structure called stat

```
struct stat sbuf;
stat(“/home/user/tmp.txt”, &sbuf);
printf(“inode = %d size = %ld\n”, sbuf.st_ino, sbuf.st_size);
```

- **dup() & dup2()**

```
int dup(int oldfd);
```

  - The dup() system call creates a “copy” of the file descriptor oldfd
  - Returns the lowest-numbered unused descriptor as the new descriptor
  - The old and new file descriptors represent the same file

```
int dup2(int oldfd, int newfd);
```

  - Close newfd before duping the file descriptor oldfd
  - dup2 (fd, 1) equivalent to

```
close(1);
dup(fd);
```

  - Lowest numbered unused fd (i.e., 1) is used (Assume STDOUT is closed before)
  - **Duplicate descriptors share the same file state**
  - **Closing one file descriptor does not close the file**

- What happens to the FD table and the file objects across fork() and exec()?
  - The FD table is copied across fork( ) ⇒ File objects are shared
  - On exec, open files remain shared by default

- **pipe()**
    - pipe( ) takes array of two FDs as input
    - fd[0] is the read end of the pipe
    - fd[1] is the write end of the pipe
- Shell piping : ls | wc -l
    - pipe( ) followed by fork( )
    - Parent: exec(“ls”) after making STDOUT → out fd of the pipe (using dup)
    - Child: exec(“wc”) after closing STDIN and duping in fd of pipe
    - Result: input of “wc” is connected to output of “ls”

> Q. Shouldn't ls be the child process here and wc the parent process, coz how can a child wait for a parent process to finish, although a parent can do so by calling wait(). For this command the execution of ls should finish before that of wc, thus ls should be the child process

OSTEP Chapter 39

> - Each file is associated with a low-level name called the inode number

- creates a new file in cwd

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);
```

- This array tracks which files are open on a per-process basis

```
struct proc {
        ...
        struct file *ofile[NOFILE]; // Open files
        ...
};
```

- Each process maintains an array of file descriptors, each of which refers to an entry in the system-wide **open file table**. Each entry in this table tracks which underlying file the descriptor refers to, the current offset, and other relevant details such as whether the file is readable or writable.
-

```
struct file {
        int ref;
        char readable;
        char writable;
        struct inode *ip;
        uint off;
};

struct {
        struct spinlock lock;
```

```
        struct file file[NFILE];
} ftable;
```

- However, each process has a separate entry in the Open File table, the processes created via fork() point to the same entry in the OFT as the parent

```
int main(int argc, char *argv[]) {
        int fd = open("file.txt", O_RDONLY);
        assert(fd >= 0);
        int rc = fork();
        if (rc == 0) {
                rc = lseek(fd, 10, SEEK_SET);
                printf("child: offset %d\n", rc);
        } else if (rc > 0) {
                (void) wait(NULL);
                printf("parent: offset %d\n",
                (int) lseek(fd, 0, SEEK_CUR));
        }
        return 0;
}

prompt> ./fork-seek
child: offset 10
parent: offset 10
prompt>
```

- **NOTE:** You also need to fsync() the directory that contains the file foo. Adding this step ensures not only that the file itself is on disk, but that the file, if newly created, also is durably a part of the directory
- rename() syscall renames a file, in a manner called **atomic** update. This is because if a crash occurs while renaming the file, the file name is either the old name or the new name and nothing in between.
- This property is used by the text editor, which creates a temp file while the user is editing and then renames it to the curr file name. Thus the update is **atomic**
- Each file system usually keeps the information about a file or directory in a structure called an inode
- After creating a hard link to a file, to the file system, there is no difference between the original file name (file) and the newly created filename (file2); indeed, they are both just links to the underlying metadata about the file, which is found in inode number 67158084
- only when the reference count reaches zero does the file system also free the inode and related data blocks, and thus truly “delete” the file.
- A hard link points to the same file, that is referred to via the inode number, whereas a soft link or symlink points to a file using its path as the parameter and thus when you delete the original file/directory, it leads to dangling pointer

- hard links can't be created to a directory and to a file in other disk partition
- The permission bits define what a user of the system can do with the file, are written as 'abc'. The permission bits are in the order:
    - a: Owner: rwx, or a combination of these
    - b: Group: what the group can do with the file
    - c: Others: what other users on the system can do with file

Summary

- A directory is a collection of tuples, each of which contains a human-readable name and low-level name to which it maps. Each entry refers either to another directory or to a file. Each directory also has a low-level name (i-number) itself. A directory always has two special entries: the . entry, which refers to itself, and the .. entry, which refers to its parent
- To access a file, a process must use a system call (usually, open()) to request permission from the operating system. If permission is granted, the OS returns a file descriptor, which can then be used for read or write access, as permissions and intent allow

Topics to be covered

- Inodes
- **strace** to figure out what the program is actually doing
- open file table
- For more on what is shared by processes when fork() is called, please see the man pages.
- can a file be opened in multiple modes in the same program, what happens if a write happens to a file and then the read proceeds

Week 4: Virtual Memory, Address Space

Lecture Notes

Imp Resource

- A typical executable file contains code and statically allocated data
- **Statically allocated: global and static variables**
- **Code segment size and initialized data segment size is fixed (at exe load)**
- intialised data segment : external, global, static, and constant variables whose values are initialized at the time of variable declaration in the program
- **unitialised data segment:** An uninitialized data segment is also known as bss (block started by symbol). The program loaded allocates memory for this segment when it loads. Every data in bss is initialized to arithmetic 0 and pointers to null pointer by the kernel before the C program executes. BSS also contains all the static and global variables, initialized with arithmetic 0. Because values of variables stored in bss can be changed, this data segment has read-write permissions.
- End of uninitialized data segment (a.k.a. BSS) can be adjusted dynamically

- Heap allocation can be discontinuous, special system calls like mmap( ) provide the facility
- Stack grows automatically based on the run-time requirements, no explicit system calls
- **brk()**

```
int brk(void *address);
```

  - If possible, set the end of uninitialized data segment at address
  - Can be used by C library to allocate/free memory dynamically
  - return 0 on success and -1 on failure

- **sbrk()**

```
void * sbrk (long size);
```

  - Increments the program's data space by size bytes and returns **the old value of the end of bss**
  - sbrk(0) returns the current location of BSS

- Finding segments (At program load)
  - ***etext***: end of text segment
  - ***edata***: end of initialized data segment
  - sbrk(0) returns the current location of BSS
  - ***end***: BSS

> - Linux provides the information in /proc/pid/maps

- **mmap(): discontiguous allocation**
  - Allows to allocate address space
    - **with different protections (READ/WRITE/EXECUTE)**
    - at a particular address provided by the user
  - Example: Allocate 4096 bytes with READ+WRITE permission

```
ptr = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_ANONYMOUS|MAP_PRIVATE, -1, 0);
```

> - What is the structure of PCB memory state, i.e. how is the memory state maintained in the PCB to know about the processes adrress space allocation pattern?
> Answer in slides: Through a circularly linked list

- fork() copies the parent memory state to the child
- exec() reinitializes the memory state of the process as per the new executable

OSTEP

Chapter 13. THE ABSTRACTION: ADDRESS SPACES

- Three major goals of OS in achieving virtualization in memory:
    - Transparency: The OS should implement virtual memory in a way that is invisible to the running program.
    - Efficiency: The OS should strive to make the virtualization as efficient as possible, both in terms of time (i.e., not making programs run much more slowly) and space (i.e., not using too much memory for structures needed to support virtualization)
    - Protection: The OS should make sure to protect processes from one another as well as the OS itself from processes.
- **TLBs**: A translation lookaside buffer (TLB) is a memory cache that stores the recent translations of virtual memory to physical memory. It is used to reduce the time taken to access a user memory location. It can be called an address-translation cache.

Chapter 14: INTERLUDE: MEMORY API

- In the program below, both the allocation of memory on the stack as well as heap is done, the variable a is declared on the stack, whereas memory is allocated on the heap, the pointer to which is stored as x

```
void func() {
        int *x = (int *) malloc(sizeof(int));
        ...
}
```

- argument of malloc defines how many bytes you require
- **sizeof()**: The ***sizeof()*** is considered as an ***operator*** rather than a function because it assigns a value at the **compile-time** whereas a **function** would allocate the value at **run-time**
-

```
int *x = malloc(10 * sizeof(int));
printf("%d\n", sizeof(x)); // returns 8 on 64-bit machines

int x[10];
printf("%d\n", sizeof(x)); // returns 40
```

- If while allocating dynamic memory for a string, you forget to add 1 for the end of string character, your program may still work fine, because the malloc may have allocated one extra byte, however this is not the case always, and thus each program that runs correctly may not be correct always
- Where is the memory of src allocated, in heap or stack?

```
char *src = "hello";
char *dst; // oops! unallocated
strcpy(dst, src); // segfault and die
```

- In general, when you are done with a chunk of memory, you should make sure to free it. Note that using a garbage-collected language doesn't help here: if you still have a reference to some chunk of memory, no garbage collector will ever free it, and thus memory leaks remain a problem even in more modern languages.

- **mmap()** can create an anonymous memory region within your program — a region which is not associated with any particular file but rather with **swap** space, something we'll discuss in detail later on in virtual memory

Topics to be covered:

- Imp Resource
- Where are the variables declared in the main function stored? stack or unitialised/initialised data
- How does the compiler returns the value while returning from a function call?
  Ans: Stores it in a register and returns. Same with return from trap
- Read Ch 15 if you've got time

Week 5: Limited Direct Access

Lecture Notes

- Can the OS enforce limits to an executing process by itself?
    - No, the OS can not enforce limits by itself and still achieve efficiency
    - OS requires support from hardware!
- What kind of support?
    - **Privilage Levels**
        - CPU can execute in two modes: user-mode and kernel-mode
        - Some operations are allowed only from kernel-mode (privileged OPs)
        - If executed from user mode, hardware will notify the OS by raising a fault/trap
        - From user-mode, privilege level of CPU can not be changed directly
        - The hardware provides entry instructions from the user-mode which causes a mode switch
        - The OS can define the handler for different entry gates
    - The OS can register the handlers for faults and exceptions
    - The OS can also register handlers for device interrupts
    - ***Registration of handlers is privileged!***
- After the boot, the OS needs to configure the handlers for system calls, exceptions/faults and interrupts
- **X86: rings of protection**

    - 4 privilege levels: 0 → highest, 3 → lowest
    - Some operations are allowed only in privilege level 0
    - Most OSes use 0 (for kernel) and 3 (for user)
    - Different kinds of privilege enforcement
        - Instruction is privileged
        - Operand is privileged

- Privileged instruction: HLT (on Linux x86_64)

```
int main( )
{
        asm(“hlt;”);
}
```

    - HLT: Halt the CPU core till next external interrupt
    - Executed from user space results in protection fault
    - Action: Linux kernel kills the application

- Privileged operation: Read CR3 (Linux x86_64)

```
#include<stdio.h>
int main( ){
        unsigned long cr3_val;
        asm volatile("mov %%cr3, %0;"
                : "=r" (cr3_val)
                :: );
        printf("%lx\n", cr3_val);
}
```

    - CR3 register points to the address space translation information
    - When executed from user space results in protection fault
    - “mov” instruction is not privileged per se, but the operand is privileged

- **Interrupt Descriptor Table (IDT): gateway to handlers**

    - Interrupt descriptor table provides a way to define handlers for different events like external interrupts, faults and system calls by defining the descriptors
    - Descriptors 0-31 are for predefined events e.g., 0 → Div-by-zero exception etc.
    - Events 32-255 are user defined, can be used for h/w and s/w interrupt handling
    - Each descriptor contains information about handling the event
        - Privilege switch information
        - Handler address
    - The OS defines the descriptors and loads the IDTR register with the address of the descriptor table (using LIDT instruction)

- **System call INT instruction (gemOS)**

    - INT #N: Raise a software interrupt. CPU invokes the handler defined in the IDT descriptor #N (if registered by the OS)
        - Conventionally, IDT descriptor 128 (0x80) is used to define system call entry gates int means interrupt, and the number 0x80 is the interrupt number. An interrupt transfers the program flow to whomever is handling that interrupt, which is interrupt 0x80 in this case. In Linux, 0x80 interrupt handler is the kernel, and is used to make system calls to the kernel by other programs. The kernel is notified about which system call the program wants to make, by examining the value in the register %eax (AT&T syntax, and EAX in Intel syntax). Each system call have different requirements about the use of the other registers. For example, a value of 1 in %eax means a system call of exit(), and the value in %ebx holds the value of the status code for exit().
    - gemOS defines system call handler for descriptor 0x80
    - System call number is passed in RDI register and the return value is stored in RAX register
    - Parameters are passed using the registers in the following order
    - RDI (syscall #), RSI (param #1), RDX (param #2), RCX(param #3), R8 (param #4), R9 (param #5)
    - Let us write a new system call!

- **Post-boot OS execution**
    - OS execution is triggered because of interrupts, exceptions or system calls
    - Exceptions and interrupts are abrupt, the user process may not be prepared for this event to happen. What can go wrong and how to handle it?
    - *The interrupted program may become corrupted after resume! The OS need to save the user execution state and restore it on return*

- **The OS stack**
    - OS execution requires a stack for obvious reasons (function call & return)
    - Can the OS use the user stacks?
    - No. Because of security and efficiency reasons,
        - The user may have an invalid SP at the time of entry
        - OS need to erase the used area before returning
    - ***On X86 systems, the hardware switches the stack pointer to the stack address configured by the OS***

- **Management of OS stacks**
    - A per-process OS stack is required to allow multiple processes to be in OS mode of execution simultaneously
    - Working
        - ***The OS configures the kernel stack address of the currently executing process in the hardware meaning that the OS saves the return address to the SP of OS stack in case of system calls***
        - ***The hardware switches the stack pointer on system call or exception***
    - What about external interrupts?
        - Separate interrupt stacks are used by OS for handling interrupts

- **User-kernel context switch**

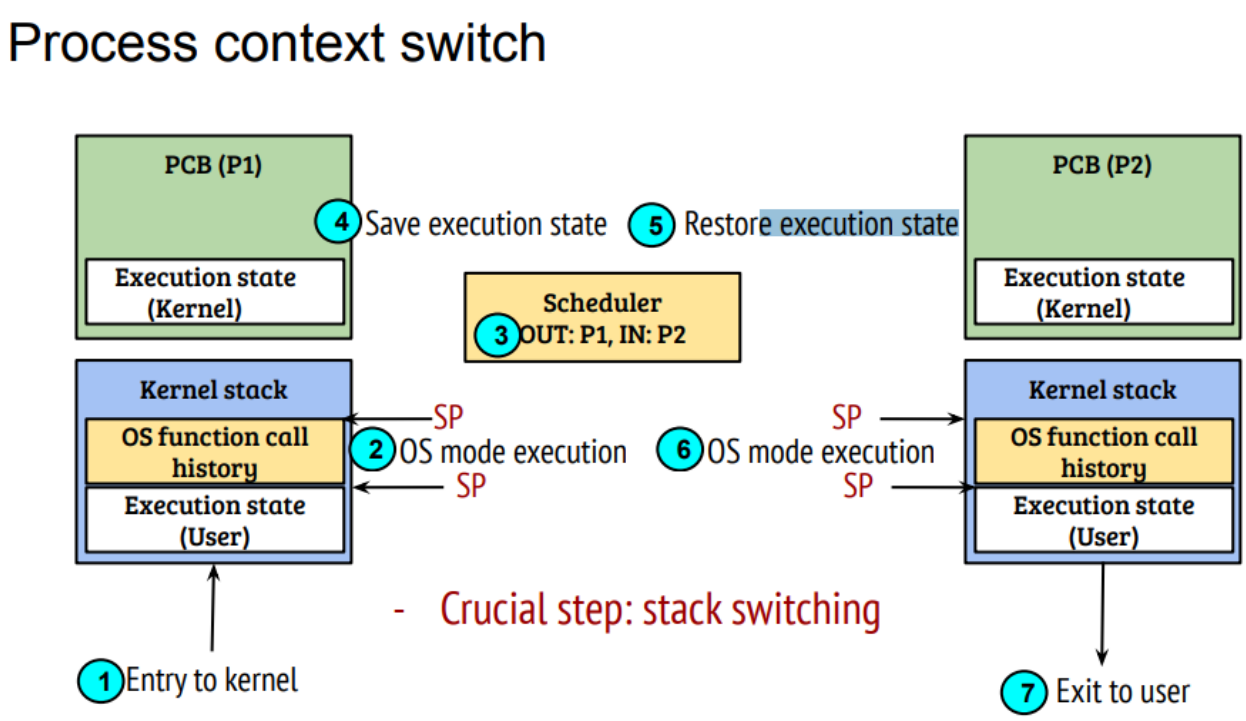
    - The OS configures the kernel stack of the process before scheduling the process on the CPU
    - The CPU saves the execution state onto the kernel stack
    - The OS handler finds the SP switched with user state saved (fully or partially depending on architectures)
    - The OS executes the event (syscall/interrupt) handler
        - Makes uses of the kernel stack
        - Execution state on CPU is of OS at this point
    - The kernel stack pointer should point to the position at the time of entry
    - ***CPU loads the user execution state and resumes user execution***

- The Kernel Stack consumes a part of the address space from all processes and **protect** the OS addresses using H/W assistance (most commonly used), because if we have a separate OS address space, we will always have to change the translation table entry

> Where is the kernel stack saved in the process address space?

Context Switches

- Triggers for process context switch
    - The user process can invoke the scheduler through explicit system calls like *sched_yield* (see man page)
    - The user process can invoke sleep( ) to suspend itself
        - *sleep( ) is not a system call in Linux, it uses nanosleep( ) system call*
    - This condition arises mostly during I/O related system calls
        - Example: read( ) from a file on disk
    - The OS gets the control back on every system call and exception, Before returning from syscall, the schedule can deschedule
    - Timer interrupts can be configured to generate interrupts periodically or after some configured time
    - The OS can invoke the scheduler after handling any interrupt



Chapter 6: MECHANISM: LIMITED DIRECT EXECUTION

- When the OS wishes to start a program running, it creates a process entry for it in a **process list**, allocates some memory for it, loads the program code into memory (from disk), locates its entry point (i.e., the main() routine or something similar), jumps to it, and starts running the user's code
- when running in user mode, a process can't issue I/O requests.
- On x86, for example, the processor will push the program counter, flags, and a few other registers onto a per-process **kernel stack**
- The user code is thus responsible for placing the desired system-call number in a register or at a specified location on the stack; the OS, when handling the system call inside the trap handler, examines this number, ensures it is valid, and, if it is, executes the corresponding code. This level of indirection serves as a form of protection; user code cannot specify an exact address to jump to, but rather must request a particular service via number.

> How does the hardware remember the address of this trap table?
> Ans from Wikipedia: The IDT is an array of descriptors stored consecutively in memory and indexed by the vector number. It is not necessary to use all of the possible entries: it is sufficient to populate the table up to the highest interrupt vector used, and set the IDT length portion of the IDTR accordingly.
> The IDTR register is used to store both the linear base address and the limit (length in bytes minus 1) of the IDT. When an interrupt occurs, the processor multiplies the interrupt vector by the entry size (8 for protected mode, 16 for long mode) and adds the result to the IDT base address

> Q. How does the OS recognise which trap handler to go to
> Answer from Stack Overflow:
> In all cases:
>
> - The processor enters privileged mode.
> - The user-mode registers are saved somewhere.

> - The processor finds the base address of the interrupt vector table, and uses the interrupt/trap/fault number as an offset into the table. This gives a pointer to the service routine for that interrupt/trap/fault.
> - The processor jumps to the service routine. Now we are in protected mode, the user level state is all saved somewhere we can get at it, and we're in the correct code inside the operating system.
> - When the service routine is finished it calls an interrupt-return instruction (iret on x86.) (This is the subtle distinction between a fault and a trap on x86: faults return to the instruction that caused the fault, traps return to the instruction after the trap.)

> Q. In what order the PC, stack and privilage level changes while making a syscall?
> Ans. They can be any order but the important thing is that all of this happens only using one instruction i.e. **trap**. After this instruction executes, the user process registers are saved onto the kernel stack, the privilage level of system has changed, and the kernel stack has come to picture

- The process then completes its work, and returns from main(); this usually will return into some stub code which will properly exit the program (say, by calling the exit() system call, which traps into the OS). At this point, the OS cleans up and we are done.

> *know more about this* what is the stub code being talked about This stub code performs early low-level initialization, such as:
>
> - Configuring processor registers
>   - Initializing external memory
>   - Enabling caches
>   - Configuring the MMU
> - Stack initialization, making sure that the stack is properly aligned per the ABI requirements
> - Frame pointer initialization
> - Initialization of the C/C++ runtime
> - Initialization of other scaffolding required by the system
> - Jumping to main
> - Exiting the program with the return code from main

- To save the context of the currently-running process, the OS will execute some low-level assembly code to save the general purpose registers, PC, and the kernel stack pointer of the currently-running process, and then restore said registers, PC, and **switch to the kernel stack** for the soon-to-be-executing process.
- By switching stacks, the kernel enters the call to the switch code in the context of one process (the one that was interrupted) and returns in the context of another (the soon-to-be-executing one). When the OS then finally executes a return-from-trap instruction, the soon-to-be-executing process becomes the currently-running process. And thus the context switch is complete.
- A timeline of the entire process is shown in Figure 6.3. In this example, Process A is running and then is interrupted by the timer interrupt. The hardware saves its registers (onto its kernel stack) and enters the kernel (switching to kernel mode). In the timer interrupt handler, the OS decides to switch from running Process A to Process B. At that point, it calls the switch() routine, which carefully saves current register values (into the process structure of A), restores the registers of Process B (from its process structure entry), and then switches contexts, specifically by changing the stack pointer to

use B's kernel stack (and not A's). Finally, the OS returns from-trap, which restores B's registers and starts running it.

OS @ boot (kernel mode)	Hardware	
initialize trap table		
	remember addresses of...	
	syscall handler	
	timer handler	
start interrupt timer		
	start timer	
	interrupt CPU in X ms	

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A
		...
	timer interrupt	
	save regs(A) → k-stack(A)	
	move to kernel mode	
	jump to trap handler	
Handle the trap		
Call `switch()` routine		
save regs(A) → proc_t(A)		
restore regs(B) ← proc_t(B)		
switch to k-stack(B)		
return-from-trap (into B)		
	restore regs(B) ← k-stack(B)	
	move to user mode	
	jump to B's PC	
		Process B
		...

Figure 6.3: **Limited Direct Execution Protocol (Timer Interrupt)**

> It so happens that when a ***trap*** is called the registers of the currntly running process are saved into the kernel stack of the corresponding process by the hardware.
>
> Q. Can the OS not save these registers into the kernel stack or can the process itself save them into the kernel stack?
> Ans.

Might be useful

Week 6-7 Scheduling

Lecture Notes

- **System Idle Process**
  - There can be an instance when there are zero processes in ready queue
  - A special process (system idle process) is always there
  - The system idle process halts the CPU
  - HLT instruction on X86_64: Halts the CPU till next interrupt

> What does halt do? How does the CPU returns from halt

- **Scheduling: preemptive vs. non-preemptive**
  - There are scheduling points which are triggered because of the current process execution behavior (non-preemptive)
    - Process termination
    - Process explicitly yields the CPU
    - Process waits/blocks for an I/O or event
  - The OS may invoke the scheduler in other conditions (preemptive)
    - Return from system call (specifically fork( ))
    - After handling an interrupt (specifically timer interrupt)
- **Scheduling metrics**
  - Turnaround time: Time of completion - Time of arrival
    - Objective: Minimize turnaround time
  - Waiting time: Sum of time spent in ready queue
    - Objective: Minimize waiting time
  - Response time: Waiting time before first execution
    - Objective: Minimize response time
  - Average value of above metrics represent the average efficiency
  - Standard deviation represents fairness across different processes

Some extra knowledge:

*** Q: What exactly is the kernel stack of a process?

The main memory (RAM) stores many things. Some part of the RAM has the memory images of user processes - the code/data/stack/heap and so on of various processes. This memory image of a process is accessible to a process in user mode. Some part of RAM has the kernel code and all its data structures, and this is not accessible to processes in user mode. In this kernel part of memory, the OS stores data structures like the list of PCBs of all active processes, and so on. The PCB of a process is a collection of all information about a process in one place, maintained by the OS. We will see the PCB of xv6 next week, and we will see that one of the important pieces of information stored in the PCB is the kernel stack of a process. That is, the kernel stack is a piece of memory in the OS part of the RAM, not accessible to a process in user mode.

Q: What is the kernel stack of a process used for? Recall that a user program uses the user stack to push arguments/stackframe/return address and so on to the stack during a function call. Similarly, the kernel stack is used to store state by the kernel when a process is running in kernel mode. That is, when you make function calls in kernel mode, you won't use the user stack, but instead, you will use the kernel stack. The kernel stack is also used to save CPU context (state of CPU registers) when a process moves from user mode to kernel mode, when we switch from one process to another during context switch, and so on. That

is, every time the OS wants to save some state of a process in order to restore it later on, it will do so on the kernel stack of the process.

*** Q: When a process goes from user mode to kernel mode due to a trap occurring (system call / interrupt / program fault), who saves the context of the process? What exactly happens?

Let me make an attempt to explain this process in detail. Suppose a process P is running in user mode. At a certain instruction in its code (say, PC = x), the program has a trap, say due to a system call. Now, this PC=x must be saved somewhere, and PC must jump to a different address "y" in the kernel code, so that OS code can run to handle this trap. [Note: "x" and "y" are memory addresses at which instructions are stored. "x" is an address in the user's code within the memory image, "y" is some address of OS code]

Q: Where does the CPU get this address "y" from? In a normal function call, the assembly code will have an instruction like "call address of function". That is, if you make a function call in user programs, the CPU knows which instruction to jump to in memory because the address is provided as argument to the instruction. A similar thing cannot happen for system calls. User cannot be trusted to provide an address of kernel code to jump to. Why? User may do bad things and jump to random addresses in the kernel. So, when user makes a system call, how does the CPU know which PC to jump to? This address is obtained from the IDT. During boot up, the kernel will tell the CPU a list of kernel addresses where kernel functions to handle traps reside. So, CPU will look up this address "y" in IDT and update PC to this address "y".

But before updating PC to "y", it must store this PC value of "x" somewhere. Otherwise, how will we know where to go back to in the user program? Therefore, the CPU must store this old PC "x" somewhere, before it can update the PC to "y".

Q: Where is this old PC / CPU context saved? This old PC (and few other registers) are stored on the kernel stack of the currently running process. Normally, any such information that needs to be preserved during a function call (e.g., return address) is stored on the user stack (which is part of the memory image of a process). Once again, OS doesn't trust the user much. So this old PC, and other registers which are part of the CPU's context, are stored on the kernel stack of the currently running process.

Q: Who saves the old PC? Old PC must first be saved before the PC switches from "x" to "y". Now, who saves this old PC? Can the kernel do it? No! Why? Because unless PC reaches "y", kernel code cannot even run. So, someone must save the old PC, switch PC to "y", and only then the kernel code can run. So, who is this someone who saves the old PC? It is the CPU hardware.

Q: So, finally, what happens on a trap? On a trap, the CPU hardware switches stack pointer to kernel stack of process (from user stack), saves old PC "x" on kernel stack, looks up IDT to obtain new PC "y" to jump to, and then jumps to code located at "y". Now, the kernel code is finally running. The kernel code can also go ahead and save more context (more CPU registers that it wishes to save, in order to fully capture user context). Then the kernel code can handle the trap, and do whatever else it wants to do. Eventually, it will return back from the kernel mode into user mode by restoring the user context (i.e., by restoring PC to "x" again).

Q: So, finally, who saves user's CPU context when moving from user mode to kernel mode? I hope the above explanation makes it clear that some part of the context is saved by hardware, and some part by the kernel. But the hardware support is essential to saving context when moving from user mode to kernel mode.

Now, let's continue the story. The kernel starts executing at address "y", runs some code, and decides it wants to context switch away from this process to some other process. By this point, the PC is pointing to

some other instruction "z" in the kernel's code. Now, we need to store this address "z" also somewhere on the kernel stack, and then switch to another process, so that we can resume again at "z" later on. This is a second type of saving context that happens during a context switch, and this "kernel context" is different from the "user context" that was saved during switch from user mode to kernel mode.

Q: So there are two different types of contexts saved on the kernel stack? Yes. When the process went from user mode to kernel mode, the hardware/OS saved the CPU register context of where the execution stopped in user mode. For example, this "user context" has saved the value of PC="x". Later on, after running in kernel mode for some time, the process stops execution in the kernel and CPU jumps to the kernel mode of another process. At this point, a second context is saved on kernel stack. For example, this "kernel context" saves the value PC="z". Both these contexts are saved on different locations on kernel stack.

After the context switch, at a later point, the OS returns back to this process, resumes execution at kernel code at address "z". Later, it may also return to user code, located at address "x". All of this is possible because we saved both these contexts on the kernel stack.