4/1/19

* Intel CPUs : x86
- Computer - CPU, RAM, Buses (I/O)
  - System bus to connect these.
- No user application directly accesses hardware, ~~bb through OS~~
  It does through OS.
  - Convenience for programmers ☺

OS provides a set of functions to programmers - 'System Call'

- Isolation ☺
  - Sharing between programs, users, etc
- Efficiency ☺ ... of memory allocation, etc.

→ CPUs
- follows some ('assembly') language instructions
- has registers to LOAD to or STORE from.
- Program being executed is in main memory (RAM)

* Compilation :- .c → executable file
  (Assembly)
- Executable files contain instructions that CPU can understand.
  - Some of these could be OS-specific instructions.
- Executable files are stored in ROM, have to be shifted to RAM for execution.
- They contain instructions and data (variables)

- Execution :-
- CPU loads stuff from RAM to registers.
- PC points to instruction being executed.
- Fetch + Decode + Execute instruction

↳ System calls

- There are OS-specific instructions in executable file that CPU does not understand (accessing hardware, etc)
  - PC is paused. OS code is accessed. OS helps run that instruction, then returns to executable instructions.
  - 'Privileged' mode of execution for system calls.
    - Function calls run in normal mode.

* Hierarchy of caches for efficiency

→ Process ≠ Running program
- Job of OS is 'process management'
  - Life cycle of a process - Creation, support, killing
- OS should also multiplex processes for efficiency.
  - Many processes are created and each takes turns to be executed on the CPU, whether partially or entirely.
  - ~~Every~~ No process perceives this multiplexing.

'Virtualizing of CPU'

- Memory for a single process may not be contiguously stored.
- Each individual process believes it has the entire RAM to itself.
- OS manages this RAM allocation.

'Concurrency management'

→ Parts of course:- ① Process management
② Memory management
③ Concurrency
④ I/O management - Device drivers

→ System calls v/s Function calls
OS ↓ Library
eg- printf() in C library.

- Library code is at the same privilege level as other user codes.
- It is simply a convenient packaging for common codes.
- System calls work in privileged mode.

9/1/19

Process Management

1. 'Memory image' of program is loaded from ROM to RAM.
2. CPU registers (like PC) contain information about the program being run through its memory image
   ∴ 'CPU Context'

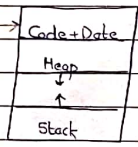
- ~~CPU Context ≡ Memory image + CPU information~~

- Memory image is destroyed when process is completed, but may not be destroyed asap

- If program uses dynamic memory allocation, the executable does not know how much memory to allocate to that variable
  - Memory image contains some vacant memory ('heap') for such variables.
    - If necessary, heap size can be increased for a process by the OS.
- Local variables to be initialized by a function block are not allocated memory in executable file, because the function might be called ~~for~~ many times or not called at all.
  - These are managed using a 'Stack' in the memory image, along with arguments to the function.

Memory Image = Code + Data + Heap + Stack

Comes from executable →



If heap and stack collide, OS must allocate more memory

- 'Stack frame' :- Portion of stack for variables/arguments of 'one' function
  - Hierarchy of stack frames

- Memory image has addresses from zero to max RAM value
  - Every process believes it can use the entire RAM.

→ System Calls
- PC in CPU context jumps from memory image code to some external specified code for OS work

I Multiple Processes

- Some system calls are 'blocking' by nature
  - That process is paused, some other process is executed in the meanwhile

eg scanf() :- Reading from keyboard.
This contains a blocking system call

- Blocking :-
  Store CPU context of current process somewhere. Retrieve CPU context of some other process.

- Process Control Block (PCB)
  - Data structure that contains info about a process.
  - One PCB for each process.
  - CPU context is stored in its PCB
  - 'State' of a process (Running or Blocked) is stored in PCB.

∴ • 'Polling' for external hardware
  - Wait for external hardware to do its thing.

Page No.
Date

Better • 'Interrupts' from external hardware
- Current process is stopped even though it did not make a blocking system call and its info is saved in PCB.
- Execution is passed on to 'Interrupt Handler' which is a part of OS code.

eg. Process P1 wants input from keyboard. P1 is blocked, P2 is running. Keyboard interrupt occurs and is handled by interrupt handler.
- 3rd 'state' = 'Ready' (for execution)
  - P2 becomes 'ready' when interrupt is being executed
  - On receiving input from keyboard, P1 becomes 'ready'

• Total 5 states of process :- {Running, Blocked, Ready, Embryo, Terminated}
(Embryo: being created; Terminated: pending cleanup)

• Scheduler will decide when to run the 'ready' processes.

• Running → Ready
1) Interrupt occurs
2) Current process has been running for too long. Discretion of scheduler.

1/1 • Programs written in any language can be run on any OS.
- 'POSIX' API - Programming OS Interface
  - Common interface that is shared by many OSes like Linux and Windows
  - Names of system calls for different POSIX compliant OS are same, but implementation may differ.
  - Source code can be run on different OSes.

- Programming languages (C library) any way repackage system calls under different functions. So, programmer does not worry about POSIX compliance.

II SYSTEM CALLS

1 fork()
- Parent process forks to produce a child process.
- Forking is the only way to create processes.
  - The first process is called 'init' process
- init → Terminal based OS → 'shell' process (Cmd prompt).
- init → GUI based OS → GUI process

eg-
```
main()
    int ret = fork()   → C library function which simply calls
    if (ret == 0)        fork system call
        { //child program }
    else { //parent program }
```
- Child continues executing the same program as parent, but value of ret() is different.
  - To the parent, the return value 'ret' is the process ID (PID) of the child
- ~~The exec() system call~~

2 exec()
- Used in child program (ret = 0) block to
- Takes argument as an executable file or binary file.
- Then the child process ~~can~~ CPU context is entirely renewed to run the new code instead.

* Many shell commands are just pre-compiled (C) programs

3 getpid()
- returns that process' PID.
- init has PID = 1
- PCB of any process contains self-PID and parent's PID, to help construct hierarchy of processes
- It does not have child's PID.

exec() (Contd.)
- Entire memory image is rewritten.
  - No useful code should follow exec() statement in child code
  - Further code is run only if exec() fails (to return an error)

4 exit()
- Terminates process
- This is added by compiler at the end of code.
- Termination is not instant. Initially, the state of process is changed to 'Terminated'
- A 'Terminated' process is destroyed (cleaned up) only when its parent calls wait()

5 wait() - Reaping process
- Destroys any one of your 'terminated' child processes.
- Blocks current process if no child is terminated
- Every process must run wait() once for each child
  (Unblocked when a child is terminated.)

6 waitpid()
- Destroys a specific child process
- Can not block if child is not terminated

- If a child's parent is terminated before it is, the child's parent PID is changed to 'init'
- init periodically runs wait() to destroy its adopted orphans

If you know that C2 process will take a long time to complete and you don't want its parent to wait around to kill it, the parent forks twice.

P → C1 → C2

Then C1 is exited, P waits to destroy C1. Then C2 is adopted by init and P can die in peace.

* Inter-process Communication (IPC)
- Signals

eg - When you press Ctrl+C, a sigint signal is sent to current process.
- kill() (C function & system call)
  - can be used to send a signal from one process to another
- Signal Handler - Handles received signals of different types
  - Programmer can overwrite signal handler code to modify how receiver process reacts to a signal
- OS allows processes to send signals only to processes created by the same user, for security reasons.

- Signal number 9 is 'kill', which terminates process.
  - Signal handler for kill cannot be overwritten, so that processes don't contrive to run forever.

- By default, a signal is sent to an entire 'process group' (containing children and grandchildren along with parent)

eg - On pressing Ctrl+C, parent and all its children should die. But by modified signal handling, a bash shell, on receiving Ctrl+C, kills its children but not itself.

16/1

* Function Call

Program	Executable
main() {	push ....
f( )	jmp ....
}	
f( ){	
}	

- When function is called,
  1) Store old PC in stack, along with arguments, local variables, regs (allocate memory for 'New stack frame')
  2) Update PC
  3) Restore stack stuff.
  4) Restore PC.

- The PC of function might as well be in a different file (eg - C library)

→ System Call

- Does not use user's stack for security reasons.
- Uses 'Kernel Stack'
  - Present in a part of memory that users are not allowed to access.
- CPU has two modes - User, Kernel
  - CPU can access protected memory in kernel mode.

- OS has an 'Interrupt Descriptor Table' (IDT)
  - Mapping : Type of interrupt/system call → Address in kernel memory
  - Used to update PC
  - Loaded during boot-up. Before boot-up, we are always in kernel mode at the end of which init process is created

* Kernel = Core part of OS
  = OS - Utilities like desktop, wallpaper, etc

- 'Trap' instruction — Before system call, interrupt, program faults (0, 2, 3)

① - Called just before system call.
  0) Go to kernel mode of CPU
  1) Update SP to kernel stack
  2) Push old PC, etc
  3) Update PC to within kernel memory by checking IDT

- Before calling trap, the required arguments (relating to which system call is to be called) must be stored in pre-defined registers (specified in CPU manual)

② - CPU automatically calls trap for external events, interrupts.
  - Every type of interrupt has an IRQ (Interrupt Request) number. That corresponding entry in IDT is jumped to.

③ - Trap is called for any fault in programs - Segmentation fault, divide-by-0, index out of bounds, etc.
  - Process is eventually killed

- 'Return from Trap' instruction
  0) Restore stack stuff
  1) Update SP to user stack
  2) Go to user mode of CPU
  3) Restore PC

• If you don't want to return from trap call back to that process (eg - process calls exit(), program fault, process calls blocking system call),
Kernel scheduler decides which process to return to.
'Context Switch'

- Type of kernel schedulers :-
  1) Non-pre-emptive - Switch process only when current process is completed or gives an error.
  2) Pre-emptive - Switch processes regularly any way
     ✓ More popular

- Context switch :-
  Jump from kernel mode of one process to that of another process which had moved to kernel mode some time in the past
  'Kernel mode → Kernel Mode'
  eg - - P2 was in user mode (U).
  - It switched to kernel mode, after storing user context, based on where it left user code from.
  - Then it was context switched. Before that, it stores kernel context, based on where it left kernel code from
  - Execution moved to kernel mode of P1 based on the kernel context that P1 had previously stored.
  - If P1 were a new process, its kernel context had been initialized to some dummy stuff. This is done by fork() itself.

- This is why parent has to clean up its terminated child.
  - When child calls exit(), it goes into its kernel mode. You cannot erase kernel stack while you are working in it.
  ∴ Terminated process can be cleaned only by some other process, after jumping to its kernel mode.

III FILE DESCRIPTOR ARRAY

13/1 ✱ Along with self and parent's process ID, PCB also stores a 'file descriptor array'
↳ Stores the file descriptors of all files that the process is allowed to access
  - File descriptor = An integer allotted to each file used in code.
    int fd = open ( );

• When a child is born (forked), it inherits the parent's FDA, (with same files and indices)
  - The init process already has 3 fd's, which all processes inherit.
    a) STDIN :- Standard Input (usually keyboard) read(0,...
    1) STDOUT :- Output screen write(1,...)
    ✱ 2) STDERR :- Error

• open() and close() are system calls for files.
• 'I/O Redirection'
  eg printf ("Hello"); // Prints on screen
  close(1);
  printf ("Hello"); // Error
  fd = open (foo.txt) // fd = 1 (first empty array index)
  printf ("Hello") // writes to foo.txt.

eg Shell command :- $ ./a.out > foo.txt
Shell prints nothing. All printed output goes to foo.txt.
Implementation from shell :- ret = fork();
if (ret == 0) { close(1);
open(foo.txt);
exec(a.out);
}

• exec() does not clear file descriptor array.

IV. → Inter process Communication (IPC) (Cntd)

A] * Shared Memory

- Can be shared between processes by calling a few system calls

Memory image

- Both processes can read or write, edit variables, etc.
- Other process does not understand activities of other processes in same memory (not 'notified' of edits)

B] * Signals (done before)

C] * Pipes. = Location in memory

- Pipe() returns two file descriptors

int fd[2] pipe(fd);

- One fd is for read end. Other is for write end

- IPC :-

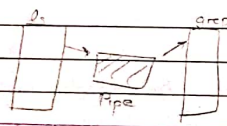
Parent creates a pipe. Forks. Parent closes one of fd's. Child closes the other fd. Voila.

- In general, more than one process writing to same file can cause errors.
- Only one guy writes to pipe. Only one guy reads from pipe

eg $ ls | grep "foo.txt"

Redirect output of program 1 to input of program 2

Shell makes two children → ls
→ grep



Both processes can run simultaneously while exchanging data, as read for ls & write for grep available

Named / Anonymous pipes

- If pipe buffer is full, write() will block
  empty, read()
- Reading data also clears up the buffer
- Can be made non-blocking and simply return back.

D] Sockets

fd = socket(); // adds to file descriptor array

Each process can create one socket, then connect those two. ✓

Socket connection = 'Bidirectional Pipe'

- Both can read and write.
- Socket connections also work between processes on different computers.

• Connection :-

Socket 1 binds to an 'address'
↳ any string (not for processes on same PC)

Socket 2 connects to that address.

22/1

V SCHEDULING POLICIES

When a process switches to kernel mode, the OS calls the scheduler to possibly context-switch.

Most OS's are pre-emptive.

• 'Ready Queue' has all ready processes

→ Performance Metrics :- ↑ fairness, efficiency, ↓ overhead

1. Turnaround time = (Exit() time) - (Creation time)
2. CPU burst = Run-time of process
3. Response time = (Time when process is first given CPU) - (Creation time)

→ Naive Policies – Knows burst times for all processes beforehand.

1. First Come First Serve
   - Non-pre-emptive by definition.

2. Shortest Job First
   - ↓ average turnaround time
   - Could be implemented by a min heap
   - Lowest average turnaround time among all non-pre-emptive policies.

3. Pre-emptive Shortest Job first / Shortest Time to completion First.
   - Process that would complete first is context-switched in.

→ Practical Policies – Burst times unknown.

1. Round Robin
   - Processes in ready queue are each given a 'quantum' of time in queue order
     - Very small quantum ⇒ ↑ Proportion of overhead :(
       ↓ Response Time :)
     - Typical quantum ~ ms
   - If a process blocks or terminates, move to the next.
   - Compared to FCFS :- ↓ response time :)
     ↑ average turnaround time :(
     ↑ Fairness

2. Priority Scheduler
   - Always run process with highest priority till completion
   - :(Starvation of low-priority processes.

→ Linux – Multilevel Feedback Queue (MLFQ)

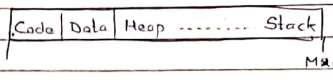
- Priority-wise arrays of processes
  1 o → o → o →
  2 o → o →
  3
  ⋮
- The nice() command can be used to change default priority of processes
- Always execute highest priority level till completion
- Within a priority level, do a RR.
- Priority decays
  - Every process keeps getting demoted
  - Processes that do not utilize their entire quantum are not demoted
    - Processes that only do a little I/O should not be penalized.
- To prevent starvation of demoted processes, all processes are periodically brought to priority 1

24/1

Memory Management

- Memory allocated to a process is usually not contiguous, for efficiency

- Every process believes it has memory $\in [0, \text{some max value}]$
  - 'Virtual Address Space'



- Virtual address - What programmer perceives
  eg - What &x will p assume
  - CPU also deals with virtual addresses only

- OS must maintain a map: Virtual addresses → Real addresses
  - Map is stored on PCB.

→ Memory Management Unit (MMU)
- Performs address translation. Present between CPU and RAM.
- Input: Virtual address from CPU
  Output: Real address to RAM
- OS manages, modifies MMU

CPU - Cache - MMU - RAM

→ Size of virtual address space
- Dictated by number of bits in PC

eg 32-bit architecture has 32-bit PC. ⇒ 2^{32} bytes of virtual memory

→ Granularity of chunks that virtual memory is divided into, each of which gets a contiguous physical address space

1. Segmentation.
- Divides virtual memory into segments (code, data, etc.), each of which is mapped to contiguously to physical memory

- Maintains a table, which is given to MMU

	Size	Virtual base (start)	Physical base (start)
Code segment	50	100	3000
Data segment	50	150	6000

Size is fixed

- ∴ Sizes of code segment, data segment, etc are different for different processes and may not be equal to allocated size.

Better 2. Pages
- Divides virtual memory into 'logical pages' of equal size, without caring if it is code, data, etc. and maps them to 'physical frames' of same size in physical memory.
- Granularity of allocation is 'page'
  - Some wastage of space, as each process might not use the entire page
    'Internal fragmentation' IDGAF
  * Wastage caused by variable segment sizes in segmentation was the OS's headache
    'External fragmentation' ∴
- 'Page Table': Logical Page No. → Physical Frame No.
  - Stored in PCB, given to MMU.
- Typical page size = 4kB

→ Expansion of a process's allocated memory
- During fork, child is allotted same memory as parent.

- Initially, there is empty memory between heap and stack.
  - Stack is automatically built up by OS.
  - System call must be made to increase heap size.
    'brk() or sbrk()'

PTO.

Page No.
Date

Last address value

* 'Program break' $\triangleq$ Code + Data + Heap = in heap.
  - brk() or sbrk() expands program break (expands heap)
- Expansion of heap is in granularity of one page.
- There actually is large separation between heap and stack.

* mmap() :- System call which returns address of any one empty page from anywhere in between (between heap and stack).