```
void print_endianness() {
    unsigned int mask = 0x12ab34cd;
    unsigned char * p = (unsigned char*) &mask;
    if (( unsigned int) *p == 0xcd) {
        puts("little");
    } else if (( unsigned int) *p == 0x12) {
        puts("big");
    }
}
```

Calculate a ^ b
- Naive:
  - a * a (b times)
  - O(b)
- Better:
  - (a^2)^(b/2)
  - O(log(b))

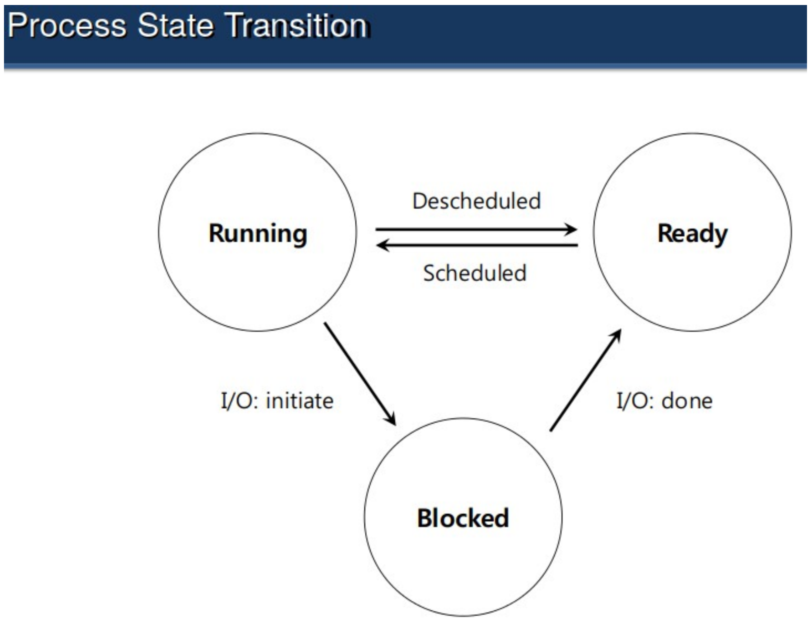
```
    // Psuedo code
    exp(a, b) {
        if (b == 1) {
            return a;
        }
        x = exp(a, b/2);

        return x * x;
    }
```

Default File Descriptors: Stdout, Stdin, Stderr

Von Neuman: Fetch instruction, decode it, execute it, store

Process States
- Running
  - Running on a processor
- Ready
  - A process is ready to run but for some reason the OS has chosen not to run it at this given moment
- Blocked
  - A process has performed some kind of operation
  - When a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor



Trap table: Remember address of syscall handler, Tells which part of opperating system to wake up

Kernel Stack: Saves state, Registers and PC are saved here

Active Stack: Pointer to a stack,when kernel returns if multiple processes, tells which process to run

Interactive Jobs: (Will be on exam) (1) Turn arround time = T_turnaround = T_completing – T_arrival (2) Fairness (3) Response time (T_FR is time that it first appears) = T_FR – T_arrival

First come, first served (FCFS)
- Convoy effect: A big job blocks smaller jobs
- Very simple and easy to implement

example:
- A arrived just before B which arrived just before C
- Each job runs for 10 seconds

Average turnaround time = (10 + 20 + 30) / 3 = 20 sec

Cons:
- Convoy effect: e.g., first job takes a very long time to run, holds up other jobs

Shortest job first:
- Non-preemptive scheduler:
  Once a job starts, don't interrupt it. Let it run to completion

Shortest Time-To-Completion First
- May lead to "starvatin", i.e., a long running job will never be run if small jobs keep appearing

Round Robin Scheduling
- Preemptive: Jobs can be interrupted
- Fair
- Run a job for a time slice (scheduling quantum) then switch to the next job in the run queue until the jobs are finished

- t_slice % t_interrupt = 0 (will be on midterm)
  because if not, then utilization would not be very good
- Turn around time is poor, because jobs will be stringed
  along for a long time
- Shorter response time: Better response time, but the cost of
  context switching will dominate overall performance
- Longer time slice: Amortize the cost of switching, worse response
  time

In case of I/O:
- Run another job while waiting for I/O

Virtual Address:
Physical Address = v + base
0 <= Virtual Address < Bound
Wastefull use of memory!

Direct Execution:
1. Create entry for process list
2. Allocate memory for program
3. Load program into memory
4. Set up stack with argc/ argv
5. Clear registers
6. Execute call main()
7. Run main()
8. Execute returnfrom main()
9. Free memory of process 10. Remove from process list

Restricted Operations (Use syscalls)
Accessing the file system
Creating and destroying processes
Communicating with other processes
Allocating more memory

Context Switch
- A low-level piece of assembly code
- Save a few register values for the current process onto its kernel stack
  - General purpose registers
  - PC
- kernel stack pointer
- Restore a few for the soon-to-be-executing process from its kernel stack Switch to the kernel stack for the soon-to-be-executing process

OS	Hardware	User Mode
Initialize trap table	remember addr of syscall handler timer handler	
	timer interrupt, save regs to k-stack, move to kernel mode, jump to trap handler	
Handle the trap, save regs to proc-struct, restore regs from proc-struct, return from trap		
	restore regs from k-stack, move to user mode, jump to B’s PC	

Three Fundemental Concepts:
1) Virtualization
-> CPU:
Performance + Control
Limited direct execution: OS intercepts
and takes control if need be, but program
runs on the hardware
-> Memory:
Program wants whole memory
Program is given Virtual Address Space
(And never knows the physical addresses)
MMU translates physical address and virtual address
2) Concurrency 3) Persistance: -> Non volitile storage (e.g. Disk, Filesystem, how to align reads and writes)

The refined set of MLFQ rules:
 Rule 1: If Priority(A) > Priority(B), A runs (B doesn’t). Rule 2: If Priority(A) = Priority(B), A & B run in RR. Rule 3: When a job enters the system, it is placed at the highest priority. Rule 4: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced(i.e., it moves down on queue).
 Rule 5: After some time period S, move all the jobs in the system to the topmost queue.