- **4 Computer Sys Parts**: *Hardware, OS, Applications, Users*
- **OS:** resource allocator (decides b/t conflicting requests for efficient use) & control program (control exec. of programs to prevent errors/misuses).
- **Bootstrap program:** load at power-up (stored in ROM; **firmware**) **loader** locates kernel, load into mem.
- Device controller informs CPU it's finished an operation by issuing **interrupt.**
- Interrupt transfers control to the interrupt service routine thru the **interrupt vector**.
- OS (when interrupted) preserves state of CPU by storing registers and program counter.
- **System call –** request to OS to allow user to wait for I/O completion. Accessed via API rather than direct system call use (e.g. Win32, POSIX, Java).
- 3 ways to pass params: (1) in registers, (2) store params in block/table in mem & pass as block param, (3) process pushes params on stack and OS pops off stack. No param lims on length/size for 3rd way.
- **Device-status table** – contians entry for each IO device, indicating type, address, and state.
- Main memory – large storage media that CPU accesses directly (**RAM**, usually **volatile**).
- Secondary storage – extension of main memory that provides large **nonvolatile** capacity.
- HDDs divided into **tracks** and **sectors**. **Disk controller** determines logical interaction b/t device and PC.
- Storage systems organized in hierarchy (1) speed, (2) cost, (3) volatility.
- **Caching** – copying info into faster storage system; main mem viewed as *cache* for 2ndary storage.
- Registers ↔ cache ↔ main mem ↔ electronic disk ↔ magnetic disk ↔ optical disk ↔ magnetic tapes.
- **Multiprocessors** (i.e. **parallel systems**): PROS: (1) increased thru-put, (2) economy of scale, (3) increased reliability.
- 2 types of multiprocessors: **Asymmetric**(1 CPU controls others)/**Symmetric**(all split job).
- Clustered systems share storage via **storage-area network**. High avail. when survives fails. **Asymmetric** has 1 machine in hot-standby mode. **Symmetric** has many nodes, monitor each other.
- **Multiprogramming** organizes jobs/apps. 1 job selected to run via **job scheduling**.
- **Timesharing** (**multitasking**) – CPU switches jobs so fast, creating **interactive**.
- **Response time** < 1sec. Each user has at least 1 job in memory (process). **Virtual memory** allows execution of processes not completely in memory.
- Software error or request creates **exception** or **trap**. **Dual-mode** op allows OS to protect self and PC.
- **User mode** and **kernel mode**. **Mode bit** provided by hardware. Allows OS to distinguish when system is running user or kernel code. Some instructions are **privileged**, only exec. in kernel mode.
- Timer to prevent infinite loop. Set interrupt after specific period. At 0, INTERRUPT. Back to kernel mode.
- 1 **program counter** per thread (process) specifying loc of next instruction to execute.
- Registers(1kb) ↔ cache(16MB) ↔ main memory(16GB) ↔ disk storage(100GB).
- **Protection** – mechanism for controllling access of processes/users. **Security** – defense against malicious attacks. All OSes have **UI** (e.g. CLI, GUI, Batch).
- 5 purposes of OS: **********
- **Policy** (what will be done) vs. **Mechanism** (how to do something). Allows max flex if policy changes.
- Layered OS (modularity, good debugng) vs. Microkernel sys struct (comms b/t user modules done by msg passing; PRO easy to extend, easy port to new archs, reliable, secure, CON overhead for comms)
- **Virtual Machine –** new OS that treats base OS as part of hardware.
- App failure can generate a **core dump** w/ mem of fail'd process. **Crash dump** contains kernel mem.
- **Probes** fire when code exec. Captures state data & sends it to user.
- **Batch** system executes *jobs*. Timeshare systems exec *user programs / tasks*.
- Process is program in execution. Includes (1) program counter, (2) stack, (3) data section.
- **Stack** contains temp data: function params, return addrs, local vars.
- **Process states**: new, running, waiting (for event), ready (to assign to CPU), terminated.
- Long-term scheduler (sec-min)(**job scheduler**) – selects processes for ready queue, Short-term scheduler (millisec)(**CPU scheduler**) selects which process to execute on which CPU.
- CPU switches processes via **context switch**. Kernel saves state of old process in **PCB** and loads saved state of another proceess. **Context** represented in PCB. Switch time=overhead, not useful.
- Parent/child processes have **pid**s. **Fork()** - sys call creates new proc, **exec()** sys call used after fork() to replace mem space w/ new prog.
- The call to fork() will return 0 to the child process, and the pid of the child process to the parent process.
- **Cascading termination** – parent done, kills all child execs.
- Processes share data via buffer (e.g. producer-consumer). **Unbounded** and **bounded** buffers.
- **Direct communication** – processes must name each other explicitly (send(), receive()). **Indrect communcation** – process msgs directed and received to/from **ports** (create new port, send/receive msgs thru port, destroy port).
- **Synchronous (i.e. blocking**) has sender block until msg is received & has receiver block until msg is available. Requires persistent connection (i.e. chat). **Asynchronous** (i.e. **non-blocking**) has sender send msg & continue, and has receiver receive a valid msg or NULL (e.g. email).
- **Zero** (sender must wait for recever **(rendezvous**) / **Bounded** (wait till full to send) / **Unbounded** buffering (no wait to send).
- **Automatic buffering** ((un)bounded) – indef long queue; doesn't need blocking. **Explicit** – procs might be blocked waiting for available queue space.
- **Send-by-copy** – dupe data to prevent corruption. **Send-by-ref** – offer flexibility but risk data corruption.
- **Fixed-size msgs** – sys-level implementation easy but programming more difficult. **Variable-size** – harder to implement, easier to program.
- Comms in Client-Server Systems (sockets, remote procedure calls, pipes, remote method invocation).
- **Remote procedure call** (RPC) abstracts procedure clals b/t processes on networked systems. **Stubs** – client-side proxy for actual procedure on server. Stub locates server and marshalls params. Server-side stub receives msg, upacks params & performs the procedure on server.
- **Pipes -** conduits allowing two processes to communicate (output-|->input). Unidirectional.
- **Named pipes > ordinary pipes**. Comms are bidirectional. No parents-children unlike ordinary pipes.
- For each Java class, the compiler produces an architecture neutral **bytecode** output (.class) file that will run on any implementation of the JVM. If the JVM is implemented in software, the intepreter might interpret the bytecodes for each class instance one-at-a-time. **JIT compiler** conserves time by saving the machine code for bytecode files the 1st time each class is invoked. Saves a fraction of time for each invocation of those same classes.

SYNOPSIS	DESCRIPTION	RETURN VALUE
#include <unistd.h> pid_t getppid(void)	*getppid*() function returns the parent process ID of the calling process.	*getppid*() function shall always be successful and no return value is reserved to indicate an error.
#include <unistd.h> pid_t getpid(void);	*getpid*() function returns the process ID of the calling process.	*getpid*() function shall always be successful and no return value is reserved to indicate an error.
#include <unistd.h> pid_t fork(void);	*fork*() function creates a new process. The new process (child process) shall be an exact copy of the calling process (parent process) except as detailed below:	Upon successful completion, *fork*() returns 0 to child process and returns process ID of child process to parent process. Both processes continue to execute from *fork*() function. Otherwise, -1 is returned to parent process, no child process is created, and *errno* set to indicate error.
`int main(int argc, char ** argv)`	1st param, argc (argument count) is an integer that indicates how many arguments were entered on the command line when the program was started. 2nd param, argv (argument vector), is an array of pointers to arrays of char objects. The array objects are null-terminated strings, representing the args that were entered on the CL when the program was started.	MORE DETAILS 1st elem of the array, argv[0], is a pointer to the char array that contains the program name or invocation name of the program that is being run from the CL. Argv[1] indicates the 1st arg passed to the program, argv[2] the 2nd, etc.

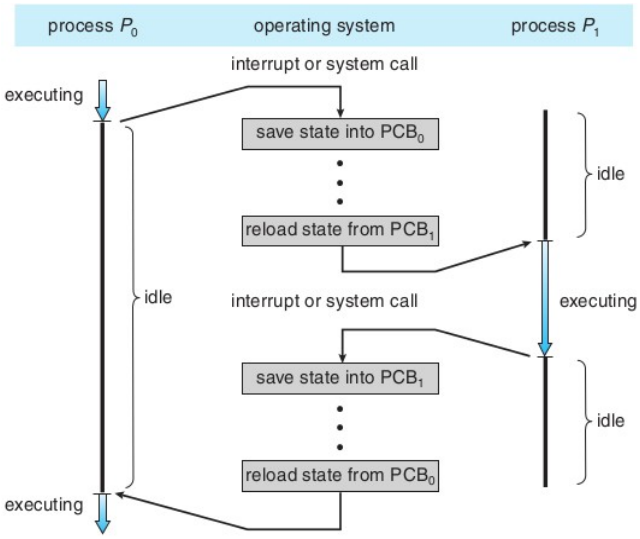

Figure 3.4 Diagram showing CPU switch from process to process.

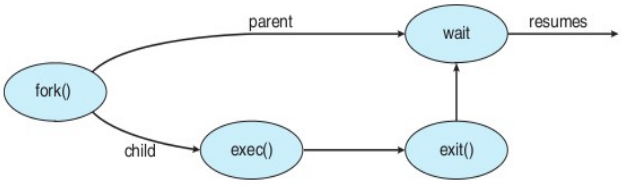

Figure 3.11 Process creation using `fork()` system call.

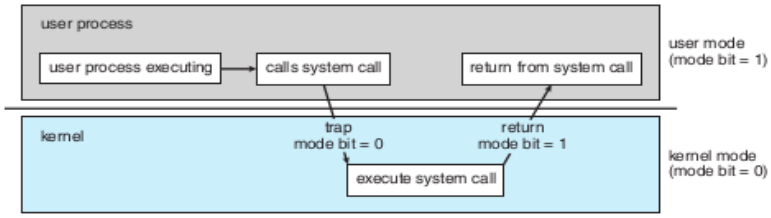

Figure 1.10 Transition from user to kernel mode.

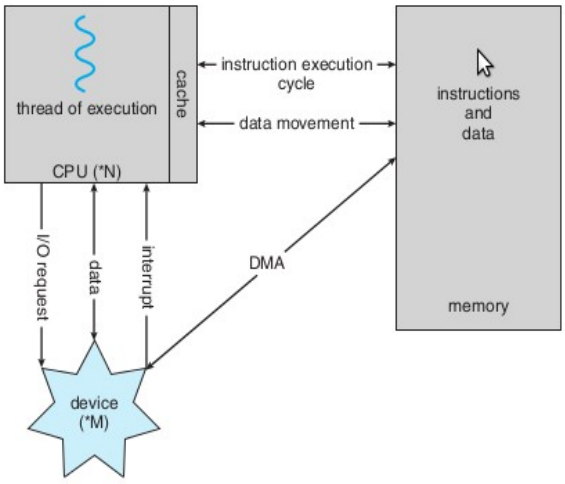

Figure 1.5 How a modern computer system works.

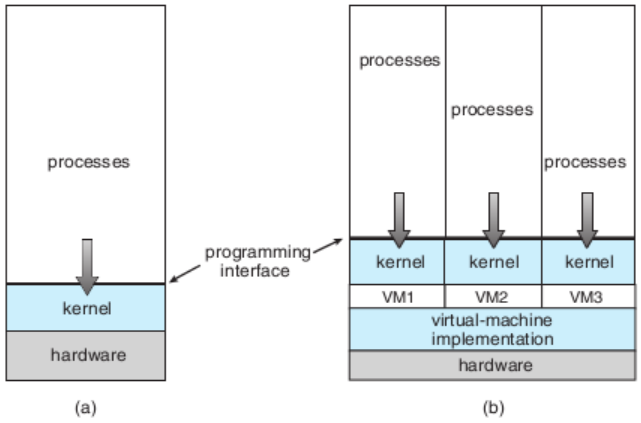

Figure 2.17 System models. (a) Nonvirtual machine. (b) Virtual machine.