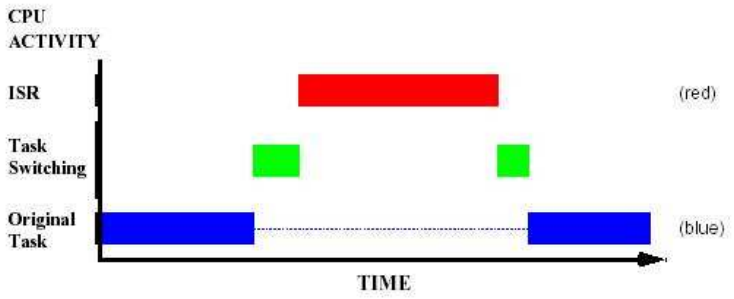

Figure 1: Interrupt

0.1 Interrupts and Traps

- Consider the case when data is to be input from the outside world. One approach is to execute a small code fragment to manage the transfer of data in from the outside world e.g. a peripheral. But when should this code fragment be run?
  - In a polling system, the computer periodically executes (or polls) the peripheral device of interest and inputs data when it is available. This means data can only be input if the peripheral device is polled.
  - In an interrupt driven system, the peripheral triggers the execution of the previously mentioned code fragment when it has data ready. We say interrupt because the handling of this data transfer interrupts normal program execution. The original task (blue) has execution interrupted while a task switching occurs (green) and then the interrupt service routine runs (red) to do I/O. Later, the original task resumes.
  - As the system has more than one peripheral, it will need more than one interrupt service routine available. Generally, a separate interrupt service routine (ISR) is provided for each peripheral that needs to trigger some CPU activity so an array of function pointers holds the start address of each of the ISRs provided i.e. **void** $(*isr\_vectors[])()$.
  - This trigger mechanism allows a computer response to an external event – what about internal events? Most computers are also able to trigger special event handling in software by executing a special instruction called software interrupt or trap.

- The operating system gets the control of the CPU (which may be busy waiting for an event or be in a busy loop) when either an *external* or an *internal* event (or an exception) occurs.
  - external events
    * Character typed at console
    * Completion of an I/O operation (controller is ready to do more work)
    * Timer: to make sure operating system eventually gets control.
    * Hardware failure
    * An *interrupt* is the notification of an (external) event that occurs in a way that is *asynchronous* with the current activity of the processor. Exact occurrence time of an interrupt is not known and it is not predictable
  - internal events
    * System call
    * Error item (e.g., arithmetic overflow, division by zero, illegal instruction, addressing violation)
    * Page fault, reference outside user's memory space
    * A *trap* is the notification of an (internal) event that occurs while a program is executing, therefore is *synchronous* with the current activity of the processor. Traps are immediate and are usually predictable since they occur while executing (or as a result of) a machine instruction.
- Interrupt Cycle
  - Fetch next instruction
  - Execute instruction
  - Check for interrupt
  - If no interrupts, fetch the next instruction
  - If an interrupt is pending, divert to the interrupt handler
- Systems that generate interrupts have different priorities for various interrupts; i.e., when two interrupts occur simultaneously, one is serviced "before" the other.

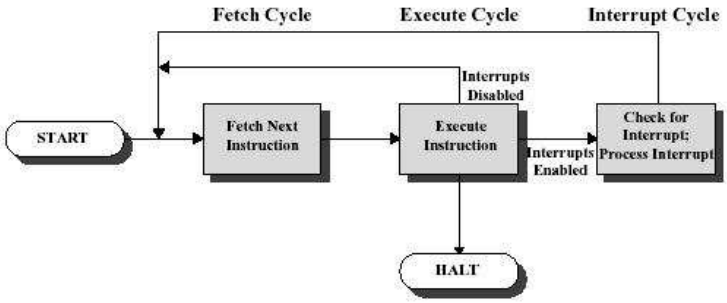

Figure 2: Interrupt Cycle

  - When a new "higher priority" interrupt occurs while lesser interrupt is being serviced, the current handler is "suspended" until the new interrupt is processed. This is called the "*nesting of interrupts*."
  - When interruption of an interrupt handler is undesirable, other interrupts can be "*masked*" (inhibited) temporarily

- Interrupt handling by "words". When the CPU receives an interrupt, it is *forced* to a different context (kernel's) and the following occur:
  - The current state of the CPU (PSW) is saved in some specific location
  - The interrupt information is stored in another specified location
  - The CPU resumes execution at some other specific location–the interrupt service routine
  - After servicing the interrupt, the execution resumes at the saved point of the interrupted program
  - Although the details of the above differ from one machine to another, the basic idea remains the same: *the CPU suspends its (current) execution and services the interrupt.*

- Modern languages such as C++ and JAVA allow the programmer to write their own exception handlers. You are writing a special function that can return no result (since it is not so much "called" as "triggered"); and the computing environment is allowing you to store the start address of your exception handler in the array $isr\_vectors[]$.

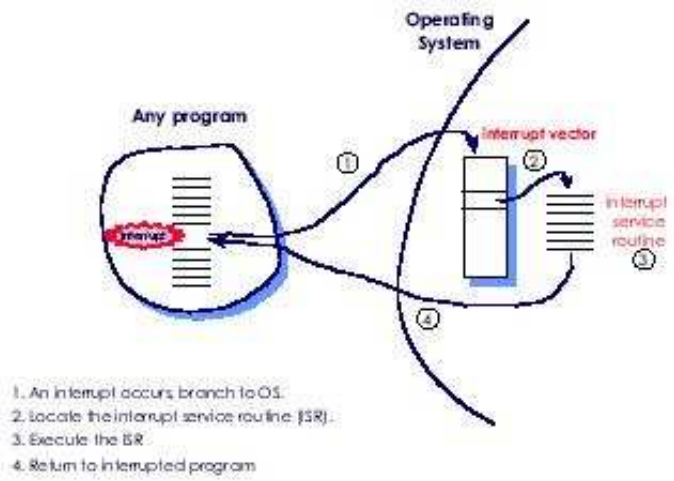

Figure 3: Interrupt Picture

- As the computer designers have total control over which event handlers can be accessed by users and which ones are reserved for their use, the exception handling mechanism is also a good way to allow a user program to make a request for a resource from the operating system. We will come across special operations such as a system call or monitor call which are implemented by the exception handling mechanism and provide controlled access to system resources.

0.1.1 Accessing OS Services

- The mechanism used to provide access to OS services (i.e., enter the operating system and perform a "privileged operation") is commonly known as a *system call*. The (only) difference between a "procedure call" and a "system call" is that a system call changes the execution mode of the CPU (to *supervisor mode*) whereas a procedure call does not.

- *System call interface*: A set of functions that are called by (user) programs to perform specific tasks. System call groups:

  - Process control, fork(), exec(), wait(), abort()
  - File manipulation, chmod(), link(), stat(), creat()
  - Device manipulation, open(), close(), ioctl(), select()
  - Information maintenance, time(), acct(), gettimeofday()

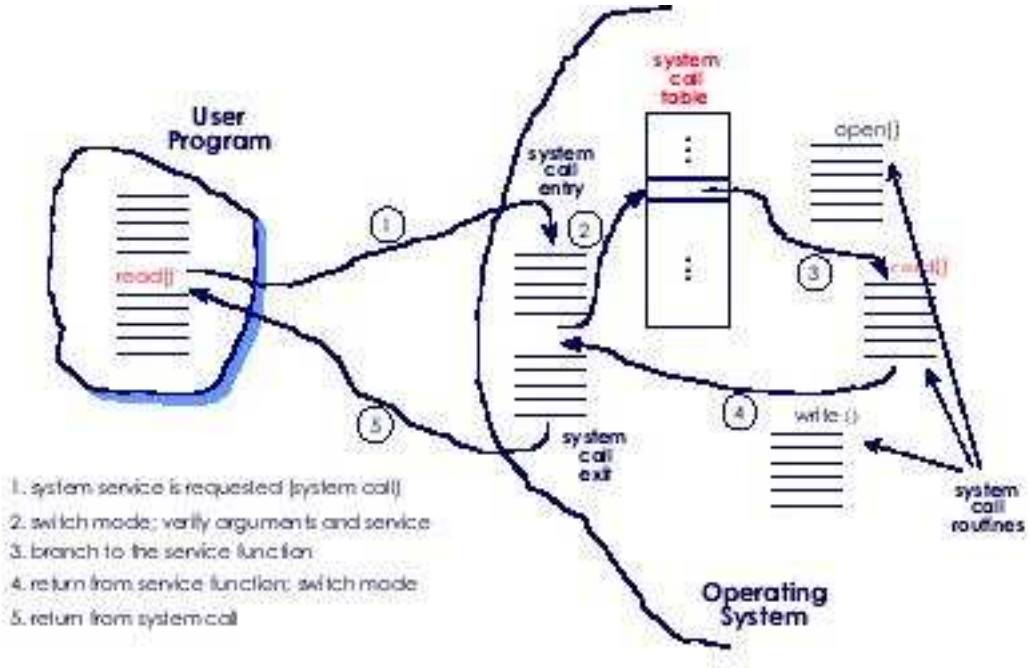

Figure 4: System Call

– Communications, socket(), accept(), send(), recv()

0.2 Operating System Components

An operating system generally consists of the following components:

- Process management
- (Disk) storage management
- Memory management
- I/O (device) management
- File systems
- Networking
- Protection
- User Interface

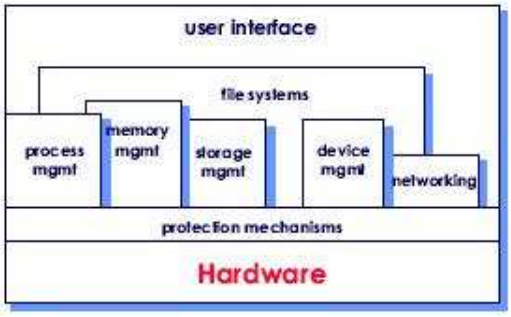

Figure 5: OS Architecture

0.3 Bootstrapping

- The process of initializing the computer and loading the operating system is known as *bootstrapping*. This usually occurs when the computer is powered-up or reset.
- The initial loading is done by a small program that usually resides in non-volatile memory (e.g., EPROM). This in turn loads the OS from an external device.
- Once loaded, how does the operating system know what to do next? It waits for some event to occur: e.g., the user typing a *command on the keyboard*.
- During "normal" operations of a computer system, some portions of the operating system remain in main memory to provide services for critical operations, such as dispatching, interrupt handling, or managing (critical) resources.
- These portions of the OS are collectively called the *kernel*.

Kernel = OS - transient components
remains *comes and goes*

System Startup

- On power up
  - everything in system is in random, unpredictable state
  - special hardware circuit raises RESET pin of CPU
    - sets the program counter to 0xffffff0
      - this address is mapped to ROM (Read-Only Memory)
- BIOS (Basic Input/Output Stream)
  - set of programs stored in ROM
  - some OS"s use only these programs
    - MS DOS
  - many modern systems use these programs to load other system programs
    - Windows, Unix, Linux

BIOS

- General operations performed by BIOS
  1) find and test hardware devices
     - POST (Power-On Self-Test)
  2) initialize hardware devices
     - creates a table of installed devices
  3) find *boot sector*
     - may be on floppy, hard drive, or CD-ROM
  4) load boot sector into memory location 0x00007c00
  5) sets the program counter to 0x00007c00
     - starts executing code at that address

Boot Loader

- Small program stored in boot sector
- Loaded by BIOS at location 0x00007c0
- Configure a basic file system to allow system to read from disk
- Loads kernel into memory
- Also loads another program that will begin kernel initialization

Initial Kernel Program

- Determines amount of RAM in system
  - uses a BIOS function to do this
- Configures hardware devices
  - video card, mouse, disks, etc.
  - BIOS may have done this but usually redo it
    - portability
- Switches the CPU from *real* to *protected* mode
  - real mode: fixed segment sizes, 1 MB memory addressing, and no segment protection
  - protected mode: variable segment sizes, 4 GB memory addressing, and provides segment protection
- Initializes paging (virtual memory)

Final Kernel Initialization

- Sets up page tables and segment descriptor tables
  - these are used by virtual memory and segmentation hardware (more on this later)
- Sets up interrupt vector and enables interrupts
- Initializes all other kernel data structures
- Creates initial process and starts it running
  - *init* in Linux
  - *smss* (Session Manager SubSystem) in NT

0.4 System Structure

- An operating system is usually large and complex. Therefore, it should be engineered carefully. Possible ways to structure an operating system:
  - Simple, single-user, *MS-DOS,MacOS, Windows*
  - Monolithic, multi-user, *UNIX,Multics, OS/360*

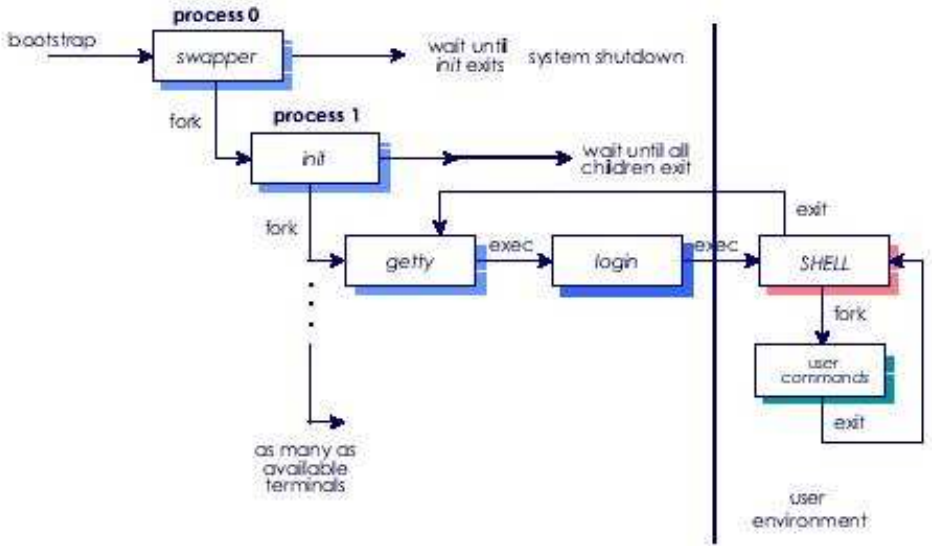

Figure 6: UNIN System initilization

- Layered, *T.H.E. operating system*
- Virtual machine, *IBM VM/370*
- Client/Server (microkernel), *Chorus/MiX*

0.5 Why Study Operating Systems?

- Build or modify real operating system.
- Tune application performance. Understanding the services offered by an operating system will influence how you design applications.
- Administer and use system well. You will develop a better understanding of the structure of modern computing systems, from the hardware level through the operating system level and onto the applications level.
- Can apply techniques used in an OS to other areas;
  - *interesting, complex data structures*
  - *conflict resolution*
  - *concurrency*
  - *resource management*

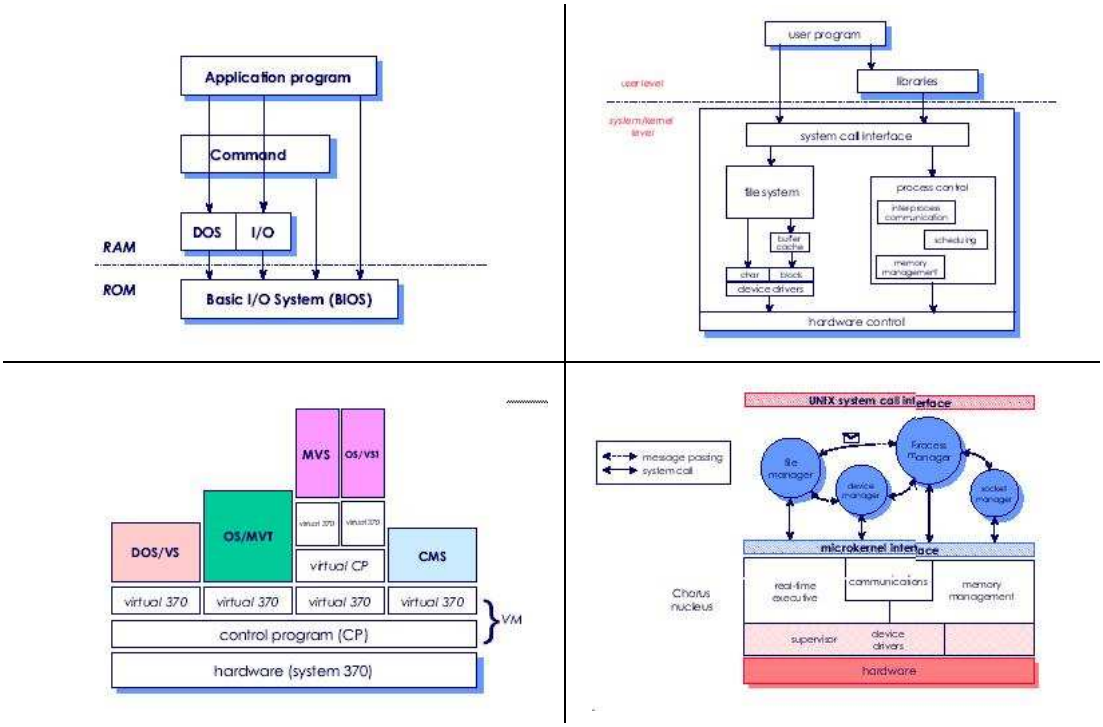

Figure 7: OS Structures, MS-DOS, Unix, IBM VM/370, Chorus

- Challenge of designing large and complex systems
- Future decisions regarding operating systems will be based on more secure knowledge.
- Curiosity: How the system works.
- *For your Course Requirement!!*

0.5.1 Problems in building OS

- *Large Systems:* 100k's to millions of lines of code involving 100 to 1000 man-years of work
- *Complex:* Performance is important while there is conflicting needs of different users, Cannot remove all bugs from such complex and large software
- Behavior is hard to predict; tuning is done by guessing

1 Processes and Threads

Simple C example
Include *text* of *header* file in <> for system, user header name in " "
main program called "main", with these argument types

```
#include <stdio.h>
int main(int argc, char *argv[ ])
{
int i;
for (i=0; i < argc; i++)
printf(‘‘command line argument [%d] = %s \n’’,
i, argv[i]);
}
```

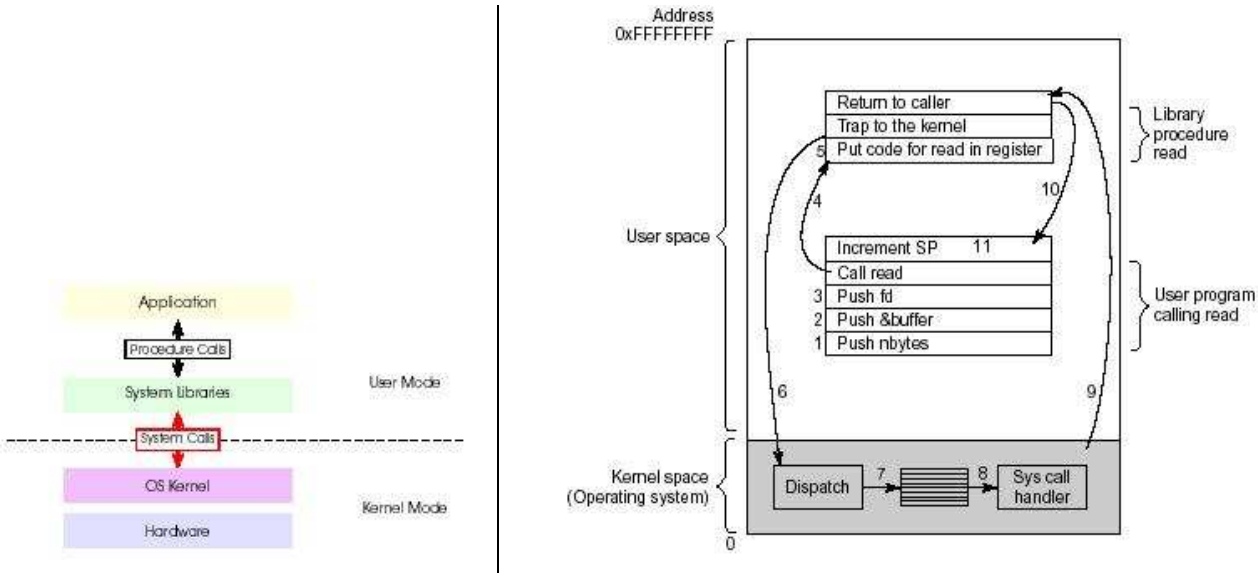

Figure 8: System Calls

1.1 Processes

What is a Process

- talking about programs executing but what it is meant?
- At the very least, we are recognizing that some program code is resident in memory and the CPU is fetching the instructions in this code and executing them

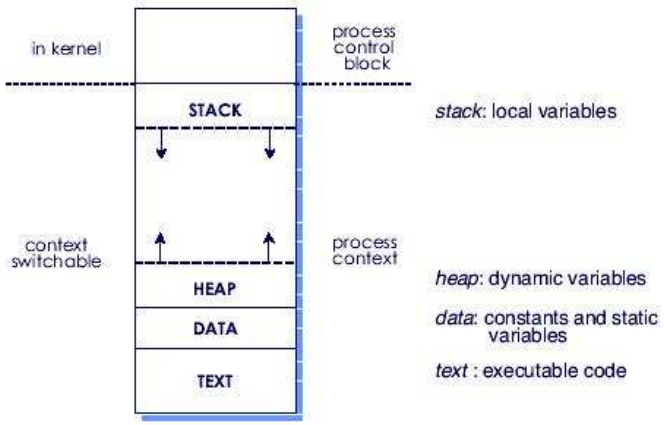

Figure 9: A UNIX Process Context

- Of course, a running program contains data to manipulate in addition to the instructions describing the manipulation. Therefore, there must also be some memory holding data.
- We are starting to talk of processes or tasks or even jobs when referring to the program code and data associated with any particular program
  - a program in execution,
  - an instance of a program running on a computer,
  - a unit of execution characterised by a single, sequential thread of execution,
  - a current state an associated set of system resources (memory, devices, files),
  - process execution must progress in sequential fashion
- An operating system executes a variety of programs:
  - Batch system, jobs
  - Time-shared systems, user programs or tasks
- Keep track of the states of every process currently executed. make sure; no process monopolises the CPU, no process starves

1.1.1 The Process Model

The operating system must know specific information about processes in order to manage and control them. Such information is usually grouped into two categories:

- process state information
  - CPU registers (general purpose and special purpose); used by the process will include:
    - memory access registers such as a stack pointer and a heap pointer, a stack frame pointer (points at a data block on the stack holding data exchanged between caller and callee functions),
    - a processor status register, possibly a register to hold return addresses,
  - program counter; this is a pointer to the program memory (text) location where the next instruction for this process resides
- process control information
  - scheduling priority, this describes the rules enforced when determining access to a processor by this process, and can include the identity of the "process ready to run" queue that this process is placed in when it is ready to take CPU time
  - resource use information, this information records the use of CPU time, elapsed time, process identity number, user or account identity number, etc.
  - I/O status information, this can include a list of I/O devices used by the process, a list of open files and any buffers associated with them
  - memory allocated, this can describe the region of memory in use (a base address and a size), the page tables (a description of which pieces of memory are "mapped" into the single region used by the process)
- This collection of process information is represented in the operating system by a data structure element called *process control block (PCB) or a task control block*. Consists of:
  - An executable program (code), which is usually referred to as the text section

- Associated data needed by the program (global data, stack)
  - the global data variables and constants, which are usually referred to as the data section
  - the dynamic storage memory used to hold temporary variables and pass function call arguments and results, usually referred to as the stack
  - the dynamic storage memory used by C++ new/delete operators and C calls to malloc()/free(), usually referred to as the heap
- Execution context (or state) of the program;
  - contents of data registers,
  - program counter,
  - stack pointer state (waiting on an event?),
  - memory allocation,
  - status of open files,

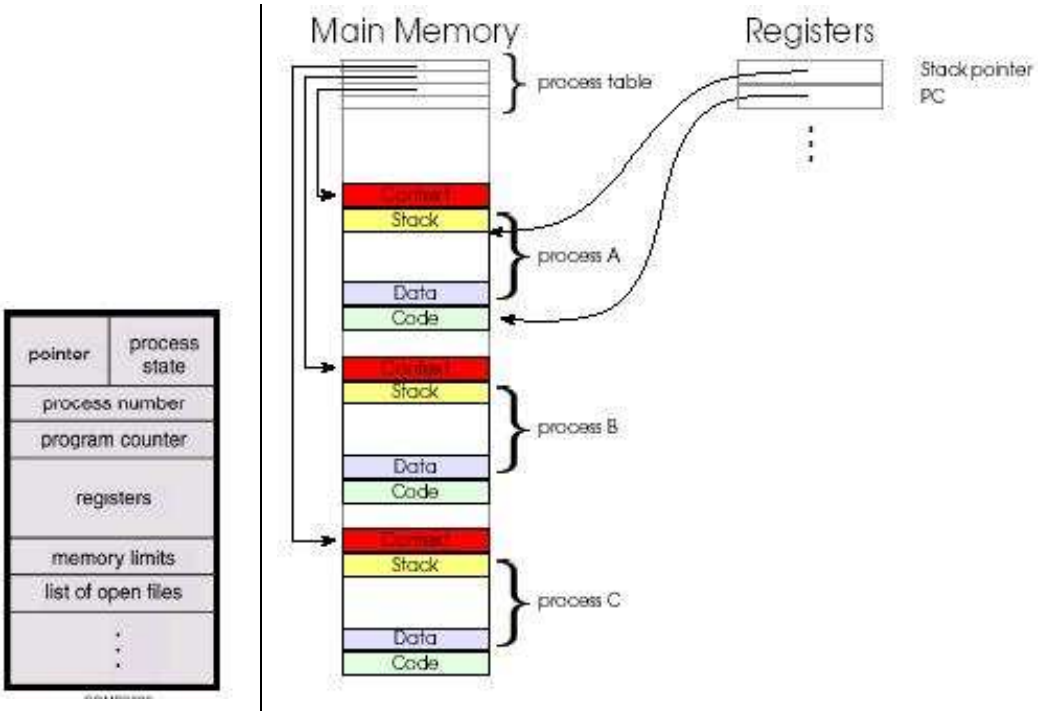

Figure 10: Process Control Block (PCB), Processes from main memory to registers

1.1.2 Context Switch

- Switching between processes is termed a context switch. When the CPU switches to another process, the system must save the state of the old process and load the saved state for the new process;
  - process table keeps track of processes,
  - context information stored in PCB,
  - process suspended: register contents etc stored in PCB,
  - process resumed: PCB contents loaded into registers
- Context-switch time is overhead; the system does no useful work while switching.
- Context switching can be critical to performance,
- Dealing with multiple processes is difficult;
  - Synchronization ensure a process waiting for an I/O device receives the signal, signals may be lost or duplicated.
  - Failed mutual exclusion attempt to use a shared resource at the same time.
  - Non-deterministic program operation; program should only depend on input to it, not relying on common memory areas.
  - Deadlocks.
- OS requirements for multiprogramming;
  - Policy to determine which process to schedule (Scheduler).
  - Mechanism to switch between processes (Low-level code that implements the decision Dispatcher).
  - Methods to protect processes from one another (memory system).

1.1.3 Dispatcher

- Dispatch Mechanism OS keeps system-wide list of processes. Each process in one of three modes; Running: On the CPU (only one in uniprocessor system), Ready: Waiting for the CPU, Blocked: Waiting for I/O or synchronization with another thread. Dispatch loop:

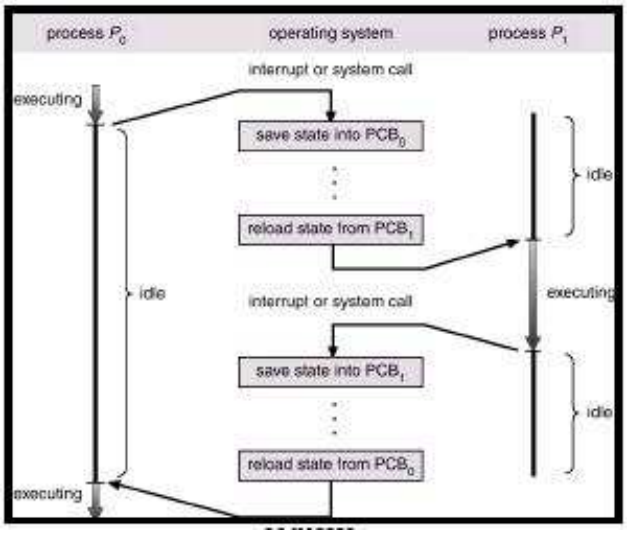

Figure 11: CPU Switch From Process to Process

```
while (1) {
run a process for a while
stop process and save its state     context-switch
load state of another process       context-switch
}
```

- How does Dispatcher gain Control?
  - must change from user mode to system mode; the CPU can only do one thing at a time. While a user process is running, dispatcher cannot run, thus the operating system may lose control
  - two ways operating system gains control;
    - Traps: Events internal to user process (System calls, Errors, Page faults)
    - Hardware interrupts: Events external to user process (Character typed at terminal, Completion of disk transfer, Control given to OS interrupt service routine (ISR))
- Dispatcher must track state of process when not running; On every trap or interrupt, save process state in Process Control Block (PCB)

1.1.4 Process Creation

- There are two ways of creating a new process:

  - Build one from scratch:
    * Load *code* and *data* into memory.
    * Create (empty) a *dynamic memory workspace (heap)*.
    * Create and initialize the *PCB* (make look like context-switch).
    * Make process known to dispatcher.
  - Clone an existing one (e.g., Unix fork() syscall):
    * Stop current process and save its state.
    * Make a copy of *code*, *data*, *dynamic memory workspace* and *PCB*.
    * Make process known to dispatcher.

- Who creates the processes and how they are supported? *Every operating system has a mechanism to create processes.*

- in UNIX, the **fork()** system call is used to create processes. **fork()** creates an identical copy of the calling process. After the **fork()**, the *parent* continues running concurrently with its *child* competing equally for the CPU. **exec** system call used after a **fork** to replace the process' memory space with a new program.

```
cmd = readcmd();
pid = fork();
if (pid == 0) {
// Child process -- Setup environment here
// e.g., standard i/o, signals exec(cmd);
// exec doesn t return
} else {
// Parent process -- Wait for child to finish
wait(pid);
}
```

- in MS-DOS, the **LOAD_AND_EXEC** system call creates a child process. This call suspends the parent until the child has finished execution, so the parent and child do not run concurrently

- Parent process create children processes, which, in turn create other processes forming a tree of processes

- Resource sharing
  - Parent and children share all resources.

  - Children share subset of parent's resources.
  - Parent and child share no resources.

- Execution, once a parent creates a child process, a number of execution possibilities exist:
  - Parent and children execute concurrently.
  - the parent may immediately enter a wait state for the child to finish – on UNIX, see the man pages for {wait, waitpid, wait4, wait3}.
  - the parent could immediately terminate.

- If the parent happens to terminate before the child has returned its value, then the child will become a zombie process and may be listed as such in the process status list!

- Address space, once a parent creates a child process, a number of memory possibilities exist:
  - the child can have a duplicate of the parent's address space – as each process continues to execute, their data spaces will presumably diverge.
  - the child can have a completely new program loaded into its address space.

- If either process needs to run a different program, it can perform a call to int execlp(const char *file, const char *arg, ...) where arguments specify the executable file and optional run-time arguments which the caller may wish to provide. See the man pages on fork and also {execl, execlp, execle, exect, execv, execvp}.

- How does each process know whether it is the parent or child after a fork? On BSD UNIX, fork() returns a value of 0 to the child process and returns the process ID of the child process to the parent process.

1.1.5 Process Termination

- A process enters the *exiting* state for one of the following reasons:
  - normal completion: Once a process executes its final instruction, a call to **exit()** is made.
  - abnormal termination: programming errors.

  - run time.
  - I/O.
  - user intervention.

- Even if the user did not program in a call to exit(), the compiler will have appended one to int main()
  - The final result of the process from its int main() is returned to the parent, with a call to wait() if necessary.
  - Process' resources are deallocated by operating system.

- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - Parent is exiting
    * Operating system does not allow child to continue if its parent terminates.
    * Cascading termination.