Beginnings of Operating Systems

Goal

To look at the history of operating systems and see why they developed as they did; to se the basic functions and designs of operating systems.

history of operating systems

First Generation

Initially, hardware only; programmer wrote, ran program & operated machine - so could halt it, modify it, etc.

- programming done in binary, or rewiring plug boards
- introduction of punched cards made life easier

Problem: "open shop" approach - if you sign up for 1 hour, you get it even if you don't need it.

Example: assume signup time is for 15m blocks, and

input time	0.3m
output time	0.5m
execution time	1.0m

$$\text{£ processor utilization} = \frac{\text{execution time}}{\text{total time}} = \frac{1}{15} \approx 7\%$$

$$\text{£ throughput} = \frac{\text{number of jobs run}}{\text{total time}} = \frac{1 \text{ job}}{15 \text{ min}} = 4 \frac{\text{jobs}}{\text{hr}}$$

Second Generation

This was heralded by transistors making computers more reliable, and by the separation of staff functions:

- operators ran jobs
- programmers wrote jobs on punch cards using assemblers, FORTRAN compilers

How did this affect throughput and processor utilization?

Example: as before, assume signup time is for 15m blocks, and

input time	0.3m
output time	0.5m
execution time	1.0m

so each job is in the system for 1.8m. Then

$$\text{£ processor utilization} = \frac{\text{execution time}}{\text{total time}} = \frac{1\text{m}}{1.8\text{m}} \approx 55\%$$

$$\text{£ throughput} = \frac{\text{number of jobs run}}{\text{total time}} = \frac{1 \text{ job}}{1.8 \text{ min}} = 33 \frac{\text{jobs}}{\text{hr}}$$

Batching began: copy jobs from card to tape, main computer runs tape, output collected on tape which is then taken to another *satellite* computer and printed

Continued example: Now assume the same I/O and execution characteristics as before:

input time	0.3m
output time	0.5m
execution time	1.0m

and let jobs be batched in sets of 50. For each batch we have:

delivery time of 50 jobs	30m
translate cards to magtape	15m
mount tape	5m
execution (1m per job)	50m
print the output tape	25m
manual separation of printer output	15m

so each job is in the system for 1.8m. Then

£ processor utilization
$$= \frac{\text{batch execution time}}{\text{mounting} + \text{batch execution time}} = \frac{50\text{m}}{5\text{m}+50\text{m}} \approx 91\%$$

£ throughput
$$= \frac{\text{number of jobs run}}{\text{mounting} + \text{batch execution time}} = \frac{50 \text{ jobs}}{5\text{m} + 50\text{m}} = 55 \frac{\text{jobs}}{\text{hr}}$$

As computers are so fast at doing many things people do, why not have them schedule themselves? First serious, informal discussion of writing a supervisory program to address idle time and work required to control I/O devices held in 1953 in Herb Grosch's hotel room during the Eastern Joint Computer Conference.

First operating system: the Input/Output System for the IBM 701, written at General Motors: a small program which provided a common set of routines for accessing I/O devices; if programs branched back to it at the end, it would accept and load next job. Made offline operation easier, since to change I/O routines (if for example a new and different printer were used), the ones in the I/O system routines had to change and not the I/O routines in every single program

Next advance: *buffering*: I/O, CPU would overlap. To do this, I/O must work independently of CPU; idea is to keep both the CPU and I/O devices busy simultaneously. First done in SHARE operating system, written by the IBM user's group for the IBM 709. Improved speed and automated much of operator's job, but operators still had to load, unload cards and tapes; also, little error recovery.

Rise of disk operating systems, which stored data on disks, not tape.

- resident loader: loads system and user programs into memory, prepares them for execution, passes control to them. Proper programs return control to another operating system routine which repeats process
- users inform operating system of job needs (such as memory, printers to be used, etc.) using a *job control language*
- device support for many different devices led to true I/O *device independence*

First computer system designed to support an operating system was the *Atlas* system designed by Manchester University and Ferranti Ltd. Several hardware innovations:

- *Extracodes* are special machine instructions causing traps to invoke special software routines; forerunner of system call or trap
- one-level store using large disk or drum as backup memory for main store: first notion of *virtual memory*
- interrupts - used to determine when external event occurs
  Example: an *alarm clock* generates an interrupt at a certain time, and the following software routine is invoked:
  ```
  alarm clock interrupt: disable alarm clock interrupt
                         save program status
                         invoke routine requested
                         reset alarm clock interrupt
                         reset program status
                         resume normal processing
  ```
  Example: a *device* generates an interrupt when a certain event (such as input arriving) occurs, and the following software routine is invoked:
  ```
  device interrupt:      disable device interrupt
                         save program status
                         invoke appropriate routine
                         reset device interrupt
                         reset program status
                         resume normal processing
  ```

Third Generation: Integrated Circuits

The coming of *multiprogramming* (where many jobs run interleaved on one machine)

- spooling: just as I/O can be buffered, so can jobs. Instead of tape, though, put jobs on disks (*backing store*) since going between input from disk, moving data and instructions between the disk and CPU, and doing output to disk is easy, whereas doing the same on tape is very cumbersome

- note monitor can schedule jobs as disks can be accessed in random order; as tapes had to be accessed in sequential order, previously job execution was always FIFO
- computation of one job, I/O of another, can overlap

These were first implemented in EXEC II, an operating system for the UNIVAC 1107; it ran jobs faster than the users could load the cards! Measuring performance in time between submission and resubmission of a job, 33% of all jobs "circulated" in under 5m, with the processor utilization being 90%.

As another example, the Burroughs 5000 Master Control Program interleaved jobs; while one waited for I/O, others ran. It also assigned priorities which influenced choice of program to run.

- other notable characteristic: all user programs were written in ALGOL or COBOL and translated by compilers; there were no assemblers available to users!

Ideas of *customer service* and *compatibility* introduced by IBM with its System/360 family

- extensive customer service and support
- very powerful operating system, the OS/360 (see Brooks' book *The Mythical Man-Month*)
- upward compatibility for all systems in family
- powerful job control language

Several problems introduced in this generation relating to job scheduling, memory management, and protection.

- *Protection* required to prevent one job from wiping out others; for example, one job reads its input and the next job (as the jobs are all in the input stream), or one job issues an illegal instruction and crashes the machine, thereby preventing other jobs from running.

  hardware solutions:
  - for *illegal instructions*, cause a *trap* (interrupt) to prevent system crash
  - to detect *bad memory references* that try to access the operating system, define a special *fence* register to separate monitor and each job. Compare the address of each memory reference to the address in the fence register, and abort if the reference crosses the fence. To detect those that try to access memory of another job, use two registers containing the high and low addresses of the current job..

  software solutions:
  - *infinite use of the CPU*: use a timer to interrupt jobs which hog CPU too long

- to prevent jobs from interfering with I/O of another job (for example by using the device simultaneously), define at least 2 modes of execution
  - kernel (system, supervisor, monitor) mode
  - user mode

  Mark some instructions as "privileged;" these can only be done in kernel mode. If they are tried in user mode, the job will abort (illegal instruction)As user jobs need to do privileged things like I/O instructions, they need a mechanism to request monitor to do it. The mechanism used is the *system call* (originated on ATLAS, and there called extracode); these are, for various computers:

computer	*system call opcode*
IBM 370	SVC
DECSystem-10	UUO
DECSystem-20	JSYS
PDP-11	TRAP
VAX-11	CHMK, CHMS, CHME

  These traps cause control to go to the operating system (monitor), which checks the legality of the request and acts accordingly. Note that the monitor can do things not related to the currently-executing job as well (such as spooling)

Time Sharing (the interactive use of a computer by many users simultaneously) was proposed by Strachy in 1959, and in 1962, a paper by Licklider and Clark emphasized its creative advantages.

- The CTSS (Cambridge Time Sharing System) from MIT and the Q-32 from SDC were the earliest operational time sharing systems; they reduced time between submission of job and obtaining of results. Both also guaranteed response to short requests and let many users share computer.
- Batch merged with time sharing on systems such as DEC's TOPS-20 (for DECSystem-20s) and VMS (for VAXen). Some batch had time-share added (IBM OS/360 with TSO)
- An early time-sharing system, MULTICS, developed at MIT, combined time sharing with many features of ATLAS such as virtual memory, protection, device independence, etc.

Virtual Machines

Archetype is the THE system, and its design is one of process hierarchy. As you go up the hierarchy, each layer defines a more developed system (this technique is called "layers of abstraction") and processes at that level ignore issues of availability of resources managed at lower level; for instance, if memory allocation done at level i, then processes at level j ($j>i$) ignore all issues associated with memory management and just invoke the process at level i when they need memory managed. It's worth looking at in detail:

Level 0: hardware
- real time clock interrupt to prevent CPU hogging
- actual number of processors
- processor management

Level 1: segment controller process
- hides details of storage management
- higher levels see only "segments" (pages), the actual locations of the segments being hidden

Level 2: operator console (message interpreter)
- handles traffic from, to operator at system console
- higher levels see their own console
  why a separate process? because first part of (input) message from system operator must be processed to figure out which process the message is to be sent to

Level 3: I/O handlers
- buffer input, unbuffer output
- higher levels see "logical device units" and not registers
  why is this above message interpreter? if a device malfunctions, the system must be able to inform operators

Level 4: user processes
- each has complete virtual machine with separate I/O devices, operator console, segmented storage and CPU
- other than communication via primitives (*semaphores*), processes completely isolated

Each layer forms "abstract", or VIRTUAL, machine

The operating system TENEX designed by BBN for the DECSystem-10 was similar (and DEC picked up so much, that TOPS-20 was nicknamed "TWENEX").

First true virtual machine was CP/CMS, later to become VM/370 it gave users (apparent) access to *all* machine features including an illusion of private address space, CPU, and I/O devices (such as "mini-disks")

How do they work? There are three modes: "virtual user," "virtual monitor," and "real monitor" modes. All traps, interrupts turn control over to real monitor, which either services the request, then modifies the virtual monitor to make it appear the virtual monitor had serviced the request, and then restarts the virtual monitor (this is how privileged requests like I/O are done), or it returns control at once to the virtual monitor, which executes the appropriate function.

Speed depends on how much you have to do in software; for example, the IBM VM/370 only simulates priviledged instructions, so its speed is acceptable

Advantages:

- protection: you have complete user isolation
- good for operating system development, because if you crash your version, others can continue working and you need not restart the machine.

Disadvantages:

- sharing hardware painful; how do you share 3 disks among 7 users? (One way: VM/360 gives each user a smaller, "virtual" disk)

Minicomputers

Here, “mini” refers to price in the early days. Gradually, price and size became (somewhat) related, so the lower the price the smaller the size.

1950s
- Burroughs E-101, Bendix G-15, etc.
  Price: under $50,000
  Characteristics: large; vacuum tubes; slow

1960s
- CDC-160, IBM-1620 both had limited instruction size (12 bits), so relative and indirect addressing modes introduced
- PDP-1, PDP-8
  Price: under $18,000
  Characteristics: PDP-8 introduced real-time clock, DMA, etc. so it began real-time control for minicomputers; however, it suffered from limited system software and used punched paper tape
- DDP-116, DATA-620, IBM-1130, IBM-1800
  Characteristics: 16-bit architectures, vectored interrupts, multiple accumulators, etc; DDP-116 had “Input/Output Selector,” which was the beginning of operating systems for minicomputers; also, IBM introduced a disk operating system for the 1800 These accepted commands from a terminal, loaded and ran programs, and monitored real-time devices

1970:
- PDP-II
  Characteristics: planned as family of compatible computers. Three operating systems available:

single-user (RT-11, had notion of foreground and background jobs); timesharing system (RSTS), real-time executive (RSX-11, supported memory management, multiprogramming, etc.)

- UNIX: Thompson and Ritchie unhappy with operating system at Bell Labs, combined ideas from DTSS, CTSS, MULTICS, to get UNIX, but made it cleaner and simpler (in part because of the space constraints!). Originally done in assembly, but rewritten in C later for maintainability and portability; no longer tied to any particular machine.

Microcomputers, Personal Computers, Workstations

As chips became smaller and cheaper, microcomputers became more common:

1970s • Intel 4004: 4 bit CPU on a few chips; followed by … Intel 8008: 8 bit CPU; then the 8080, an improved 8008.

Many hobbyists built micros; some companies made it, others didn't. Apple did, using 6502 chips to get Apple II. System software was slow to develop as micros seen as toys; Digital Research created CP/M (Control Program for Microprocessors).

Workstations started at Xerox; The Palo Alto Research Center (Xerox PARC) was chartered to carry on pure research at frontiers of technology.

Alto the first computer to use a mouse; also one of the earliest workstations

Star intended to be the heart of "office of the future" which Xerox was developing, this workstation was not particularly successful but it inspired others

One of them was the workstation Sun Microsystems started marketing, which quickly became widely accepted in the technical community.

The Star also inspired many of the *personal computers* being developed, most notably the Apple Lisa computer (which was not very successful) and the Lisa's successor, the Macintosh (which was introduced in 1984 and is quite successful).

These led to the view of "open operating systems," a design philosophy which views the operating system as a collection of subprograms rather than something which gets in the way of applications.

Networks and Distributed Systems

As local and wide area networks became available, they were used in two ways:

network operating systems: treat the net as a unified set of resources accessible by any computer on the net

distributed operating systems: similar to network operating systems, but applications cannot tell which host they run on

1969 ARPAnet made functional: nodes at SDC, UCSB, UCLA, and SRI International

1972 ARPAnet publicly demonstrated at First International Conference on COmputers and Communication; proves feasibility of long-distance networking technology.

Operating Systems Functions

The goal is to look at common operating system functions, see what a good operating system needs to do, desireable features.

I/O Functions

Two ways to read in data:

polling: CPU starts I/O, busy-waits until done; no overlap of CPU computation, I/O

interrupts: CPU loads registers and goes on; setting the registers starts device controller and I/O; on completion, device driver generates an interrupt, CPU jumps to routine, handles completion

Example: PDP-II, using console driver to output a character

polling: data is stored in memory location 0777568, and bit 7 in status register at location 0777564 is set; this causes data to prints data at console

interrupts: interrupt vector at location 064. To output, program starts the output procedure, and while the character is being transmitted, the program continues with (other) processing. When output complete, program *interrupted*: it saves state, so it can continue where it left off, and control is transferred to address in interrupt vector; that does any cleanupm, and returns, restoring the saved state and thereby allowing the program to continue. *Problem:* This is 1 interrupt per input or output char/word, which is acceptable for slow devices, but for disks, tape.., or other high-speed devices, is not. Solution? DMA

Direct Memory Access (bettter known as DMA), also called a "channel" on many mainframes. Data not copied to registers; instead, an address and length is put in a register, the I/O operation started by commanding a small, special-purpose computer called a *controller* to begin it, and then while the program returns to whatever it was doing, the controller transfers large amounts of data between the device and memory (not registers, hence the name). When done, the controller raises an interrupt.

Example:: PDP-II, TC-II device (DEC tape reader):

memory location	*what is there*
077350	memory location for data
077346	address of TC-11

077344	number of bytes of data
077342	command (read, write, ...)
077340	status register
000214	associated interrupt vector

May have to wait for I/O completion in interrupt - driven scheme, for example if the program needs the data to continue; systems generally have a **wait** system call to block the program until an interrupt occurs, or they allow the I/O operation to be called in a blocking fashion (meaning that when the operation ends, the data is in memory)

£ operating systems must provide for:

- I/O handling: the interrupts, etc. should all be invisible to the program
- error handling: given the above, just want the user to get an error code of some kind
- interrupt handling: don't want the user to have to know where the interrupt vectors are, or even (necessarily) they exist; at most, be able to specify routine to execute on interrupt
- resource control: one user at a time should be able to access an I/O device
- protection: don't want one user to be able to read another user's terminal screen just after the second typed a password
- I/O scheduling: to determine in what order processes get access to the I/O devices

Process Functions

Process is program in execution. Typical actions:

create to start a new process
delete to terminate an existing one
schedule to specify when one process should begin
synchronize to have two processes co-operate with one another
communicate to have a process send or receive information to or from another

£ operating systems must provide for:

- job sequencing and control: for scheduling, creation, deletion
- resource control: to control who gets access to what resource

- JCL interpretation: to translate the user's desires, resources into something that can be used by the computer
- protection: for example, don't want one user to kill another user's jobs
- communication: for synchronization, transmission and reception of information
- accounting: to know who to charge for computer time (common in mainframe shops, supercomputing centers, etc.)

Memory Functions

Need to share memory among the operating system and many processes. How? Essentially, the operating system transforms the addresses generated by the process into physical addresses, thereby managing where the process data goes. So the operating system must allocate (and deallocate) memory as well as track who is using what portion of memory. Typically, memory looks like:

interrupt vectors		interrupt vectors
operating system	or	operating system
user processes		user processes

£ operating systems must provide for:

- resource control: to control who gets access to what part of memory (the resource)
- protection: for example, don't want one process to access memory being used by another process
- error handling: given the above, just want the user to get an error code of some kind
- accounting: to know who to charge for memory use (common in mainframe shops, supercomputing centers, etc.)

Secondary Storage Functions

Typically, too much data and instructions to keep in memory, so some saved on secondary storage and brought in when needed. So, the operating system must determine when to move data and instructions to and from secondary storage; manage the space on those devices; and map file names to addresses on the storage medium.

⇒ operating systems must provide for:

- disk scheduling: determine how best to allocate locations on disk to files, and how to arrange for data to be transferred there most efficiently
- I/O scheduling: to determine in what order processes get access to the I/O devices
- long-term storage management: how to arrange files so that there is as little wasted space as possible (storage devices usually require data to be written in blocks of fixed size, say 1024 bytes)
- accounting: to know who to charge for memory use (common in mainframe shops, supercomputing centers, etc.)

User Interface Functions

To provide command interpreter and language to enable users to submit jobs and have them run.

⇒ operating systems must provide for:

- acceptable user interface: enable users to enter commands using the job control language easily

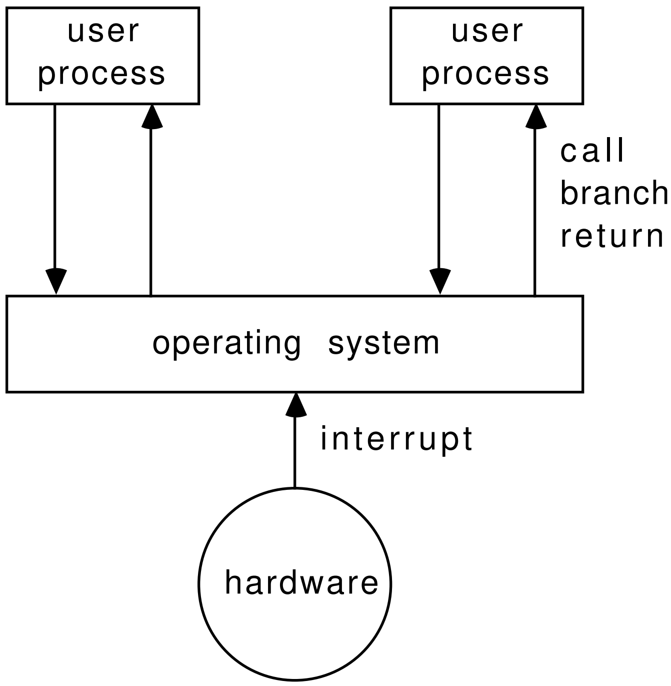
Desireable Features

- *efficiency* so system works as quickly as possible. But what metric do you use? It depends on what your system is used for. Some more common ones are:
  - mean time between jobs
  - turnaround time (batch system)
  - response time (interactive systems)
  - resource utilization (for example, processor utilization)
  - idle CPU time
  - throughput
- *reliability* so system error-free, robust (able to handle error conditions without crashing)
- *maintainable* so system can be modified or fixed easily; requires
  - modularity (layering)
  - written in a high-level language as much as possible
- *small* which usually, but not always, implies simplicity. Benefits:
  - easier to maintain, debug, check
  - more room for user applications

Organization of operating systems

monolithic

Operating system is set of programs executing on hardware; modules do different things but basically form a single “process”. user programs seen as "subroutines", invoked when operating system not performing system functions.

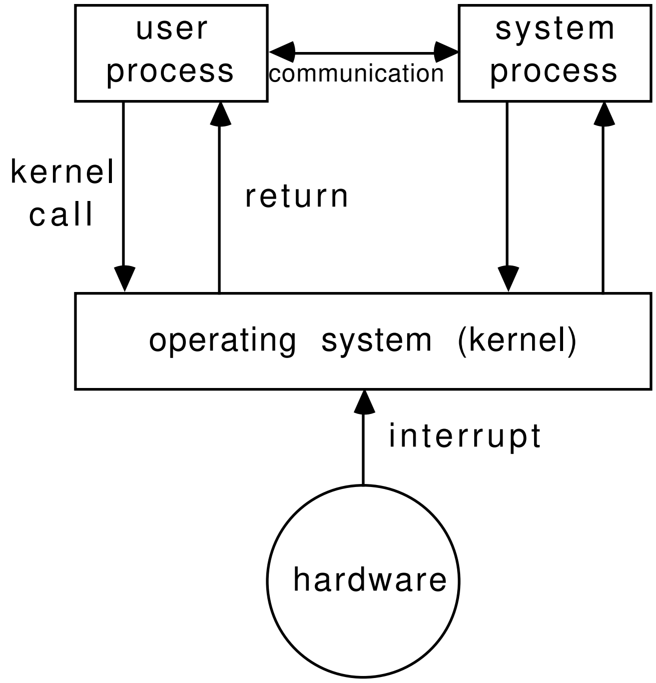


User program runs until:

- terminates (like procedure return)
- times out (*i.e.*, too much CPU time)
- issues service request (*i.e.*, I/0 operation); then the operating system executes the request
- interrupt occurs; operating system must attend to something

kernel

Operating system performs only most vital lower level functions; like monolithic, here but far fewer functions.



Example: one of the earliest is Brinch Hansen's RC 4000. Four process control primitives (create, start, stop, remove); note there is a natural hierarchy of processes ordered by process creation. Processes can also communicate.

process hierarchy

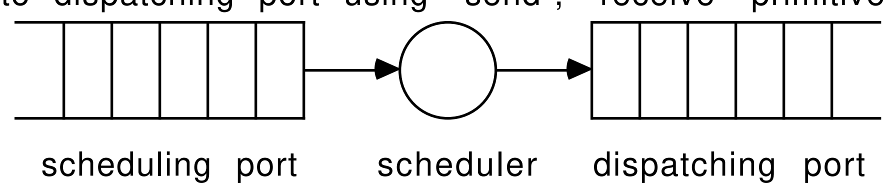
Virtualize lower level resources

Example: THE operating system

object oriented

View system as collection of objects (processes, procedures, pages, devices, messages,etc.,) and capabilities (pointers to objects, also containing rights). Kernel maintains type definitions of objects and enforces access restrictions. To access another object, issues kernel call naming capability of target and operation.

Example of use: scheduler takes processes from scheduling port, put them into dispatching port using “send”, “receive” primitives



Example: Intel's iAPX-432

client-server model

Kernel just passes messages; for example, to read, client process sends message to file server, and file server sends back data. <u>But</u> I/O is privileged, so ...

1. file serve runs in kernel mode with complete access to hardware, but passes messages in usual way
2. kernel gets message from file server, sees it is for a specific, special address. Messages sent to that address just get copied, *e.g.*, to I/0 device registers to begin a read. Kernel does not inspect contents

Use? Excellent for distributed systems... client need not know whether its local host, or a remote host, handles a request.

User Interface

Before looking at OS implementation, consider what the user sees. The *user interfaces* control how the user interacts with the system. In general, the user sees three types of software:

- *kernel* needs hardware privileges (*i.e.*, ability to execute in privileged mode); *example*: login
- *essential utilities* need no privilege but users need them and they determine user's view of OS; *example*: shells, command interpreters
- *optional utilities* are useful things users may wish to use on occasion; *example*: text formatters.

Command Interpreter

Translates user commands into sequences of actions. There are two types of languages used:

- *Job Control Language* (*JCL*) used for batch; must be complex, powerful so users can describe what is to be done for system actions
- *Command Language* used interactively; may be less complete than JCL as user can intervene when appropriate

Both are built up of commands; what these commands do is influenced by the programs and the environment.

program is a set of instructions packaged so a process can be started to execute those instructions

environment (of a process) is what distinguishes it from other invocations of the same program

Example: to compile a PASCAL program, may need to specify:

- which compiler to use
- where the source is
- where to store the resulting image
- whether to warn about non-standard usage

- whether to a generate listing
- whether to run resulting program

The first describes the program to be run; the rest describe the environment.

Invoking programs

Typically, you just name the file containing the program to be run. The command interpreter then searches a series of directories for that file (in UNIX™ terminology, this is your *search path*). If found, it is executed; if not found, give an error condition. If the file can't be executed, either continue to read commands or give error.

- should allow sensible abbreviations (command completion)
- should be as mnemonic as possible

The execution environment is composed of two parts:

global which persists until changed
Example: set CPUlimit 15m
sets the CPU limit for all processes

local which applies to one process only
Example: (set CPUlimit 15m ; who)
sets the CPU limit for the one process *who*

If I process starts another, the subprocess might inherit its parent's environment, or the parent might specify a different (local) environment. This can be done using:

- options to programs (called *parameters*)
  Example: cp -i x y
  -i is part of local environment.

The command interface can present parameterss as strings, to be interpreted by the program, in one of three ways:

- the command interface can send them as messages in a message-passing system
- the program can request parameters via system calls (cf. the TOPS-20 system call CMND)
- the command interface can arrange for new program to start with parameters stored in its virtual space (UNIX™)

One principle of all command languages is that *options performed frequently should be easy to invoke* (the *user principle*), giving rise to:

- default settings
- built-in shorthand
- init file (cshre)
- automatic chaining (invoking sequences of programs with a single command) [macros]
- personalized shorthand (aliases)

A command script crosses automatic chaining with personalized shorthand (control structure), and uses a *little language*. Carrying this to extremes, the same language could be used both to program and to issue commands. Several problems:

- interactive, noninteractive are different enough so some constructs are not likely to be used for both
- non-interactive are read more often than written, so large percentage of the characters present to make the program readable would be unnecessary when used interactively.

Example: ITS: command interpreter and program debugger the same

- debugger can invoke, interrupt, examine, modify programs
- so can command interpreter

Some other features that are good for user interfaces, in no particular order:

interrupts to cancel a command (^\ or ^C in UNIX), to get system load (^ T in TOPS-20); these are often outside control of the command interpreter; basically, the terminal driver notices the cancel character and stops the process, and the command interpreter realizes the process has stopped.

suspend stops executing process, but allows later resumption; a *process tree* scheme may stop/suspend all processes in the tree, or just one.

background vs. foreground: allows multiple processes to run simultaneously from a single command interpreter (parallelism)

pattern matching: match multiple file names at once

- done in the file server
- done in the command interpreter (UNIX)
- done in utilities (TOPS - 20, by giving the CMND system call)

history remember last few commands

command completion: finish partially-typed commands (TOPS - 20, TENEX, TCSH under UNIX)

- no completion: ring bell
- ambiguous: list possibilities

subordinate command interpreter: *called* "shell escapes" command interpreter → mailer → command interpreter → editor

Main problem is you must go back in same order

redirection: “bind” output destination, input source to program.

To bind for program “simulate”:

- bind in environment; set up the redirection before executing the program, as in:
  set associate simulate.in input.one
  set associate simulate.out output.one
- bind as parameters; for example,
  simulate input.one output.one
- have the command interpreter do the binding (UNIX); here, < binds to input and > to output:
  simulate <input.one >output.one
- bind output of 1 program to input of another (UNIX); here, | binds output of first to input of second:
  simulate <input.one | sort >output.one.

For displays *bitmapped displays* and *graphical user interfaces* allow commands to be given not just as words but using a variety of input devices (mice, tracking balls, menus) and output to be given in a variety of ways (icons, windows, graphics, etc.)

the kernel

Question

How do processes interact with the hardware of the computer? What does the program that mediates this interaction (kernel) look like?

The System Kernel

Consider now the basic, fundamental structure of operating systems.

Overview

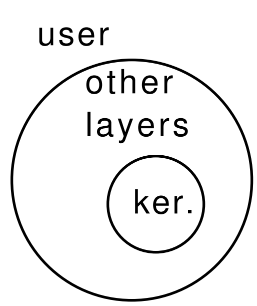
The *kernel* or *nucleus* is the interface between machine hardware and the operating system, and is to provide an environment in which processes can exist.

£ 4 classes of processes:

- primitives for process creation, destricution, IPC
- primitives for allocating, deallocating units of resources such as main memory or secondary storage
- I/O primitives (read, write, control transfer of data to/from main memory from/to secondary storage)
- operations to handle interrupts.

All functions require some basic hardware:

- interrupt mechanism; it must save PC for interrupted process, go to fixed location
- memory protection
- privileged instruction set, and hence at least user, supervisor mode (may be more, *eg*. the VAX's executive and kernel modes)
- real-time clock, which interrupts at fixed intervals according to time as measured in the outside world



There are three basic parts to the kernel:

- the *first level interrupt handler* performs initial handling of all interrupts
- the *dispatcher* switches the central processor between processes
- procedures to implement P and V (or some IPC primitives), which enable processes to communicate

All this used to be done in assembler; now the I/O and interrupt initialization routines are usually done in assembler, but the rest in some higher-level language like BLISS, C, Concurrent Pascal, Modula, or Ada™. This improves maintainability, intelligibility, and reduces the probability of error.

The Process

All pieces involve process management, so let's look at what a process looks like. We assume a highest-level process creates another process, for each user as he/she logs in (the UNIX *init* process, for example). This process initiates, monitors, and controls user's process

First-Level Interrupt Handler

This responds to 2 types of signals:

- *interrupts* from outside (devices)
- *traps* from inside (errors, invalid opcodes, etc.)

The FLIH must:

1. determine source of interrupt
2. initiate service of interrupt

In many cases, the hardware transfers control to different locations, making figuring out the source easy; but this requires extra interrupt locations.

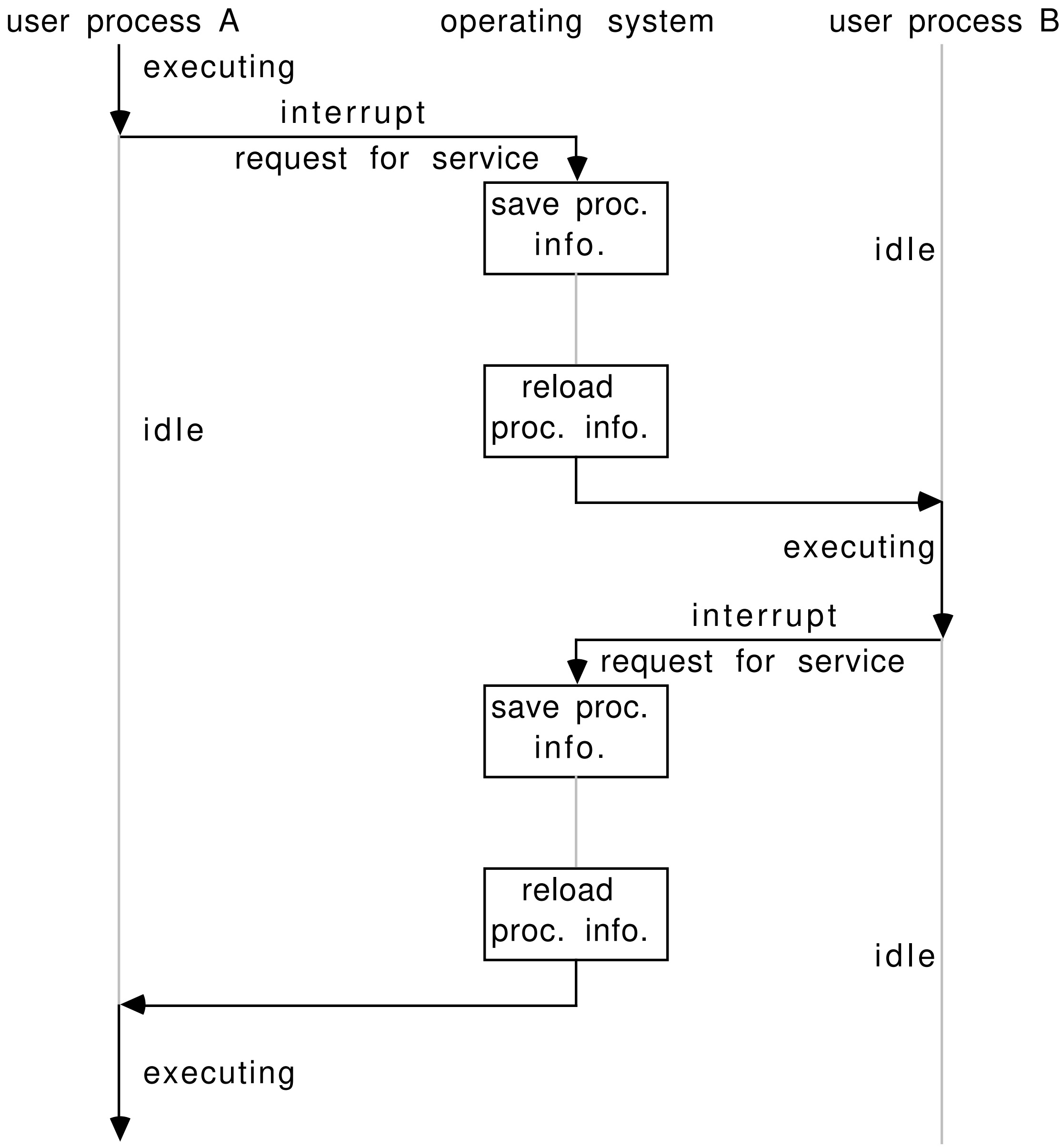
examples: the PDP and VAX both have locations in memory used for interrupt vectors, with every location assigned to a specific device.

In the most primitive hardware, all interrupts transfer control to same location, and the system must test all possible sources. In other cases, as a compromise, provide a small number of interrupt locations shared by a group of interrupt causes; then test which of the members of the group caused it

Interrupts and traps have many causes:

- I/O completion - device handler needs servicing, due to completion of I/O requests, errors, etc.
- Alert - unexpected interrupt from conditions outside the system, *ie*, operator pressing an *interrupt key*
- Timer - a time interval expired, or a real-time clock ticked; must update software clocks accordingly

- System Requests - deliberating genrated trap (IOT, TRAP) plus a code indicating what service program wants
- Program Fault Interrupt - program messes up, *ie*, attempt to divide by 0, bad memory reference, etc.
- Machine Fault - hardware error, ex. memory fetch error, bad board, etc. The system may either try to isolate faulty component and bypass it, or just terminate.



Suppose two interrupts occur at the same time. Which (if either) do you do?

Interrupts/traps have a *priority* associated with them. So you service the one with the higher priority first, then the one with lower priority.

example: on a PDP-11, 8 priority levels of interrupt; the clock has highest priority, then I/O devices, ...

May need to prevent certain types of interrupts from occuring; for example, can't have anything interrupt the updating of a software clock. So you *mask interrupts*.

- *Interrupt mask register*: this contains bits corresponding to specific interrupts or classes of interrupts; set bit means block interrupt.
- Set a priority value in a special register; only interrupts at a higher priority are accepted.

In either case, when interrupts re-enabled, pending interrupts are serviced.

processes in the kernel

Data Structures

Processes, resources represented by structures called *control blocks* or *descriptors*

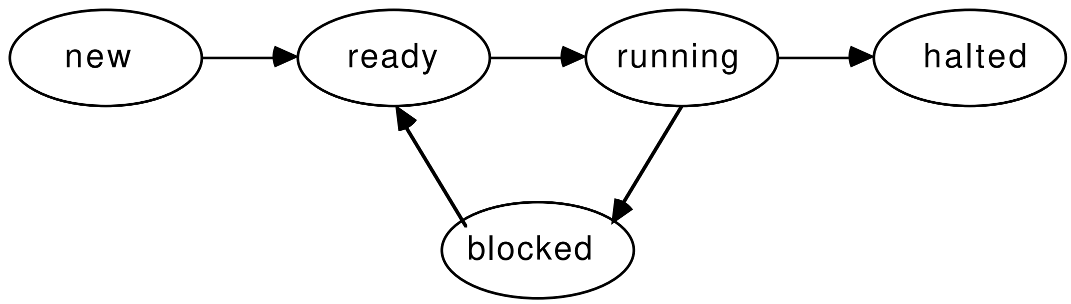
- PCB (*process control block*) represents *process* to OS; it is made at process creation time, and represents process during its existence. For example:

PCB = (id, CPU-state (registers, etc.), processor #, parts of main memory used, status, parent, children (linked list), priority,...}

- *id*: unique name or number
- *state vector*: execution of process is sequence of state vectors $s_0,\ldots,s_L,\ldots$ where each s_L, contains pointer to next instruction and values of all values of all local, global variables; also includes state of processor, address space allocated, associated resources; in short, it contains that amount of information requred by a processor to run process.
  example: In handout: state vector is CPU_State (record). Note that when the process is running, this field is undefined.
- *processor*: number of processor running process when it is running
- *Main_Store*: storage map describing address space of process, containg bounds registers or list of memory blocks or pointers to memory management structures
- *Resources*: list of all files, peripherals, etc., allocated to process
- *Created_Resources*: list of resources created by process (*eg*. ports)
- *Status*: status of process

running: process running on processor
ready: process ready to run
blocked: process cannot logically proceed until it receives a particular resource or message
halted: process has terminated
new: process just created

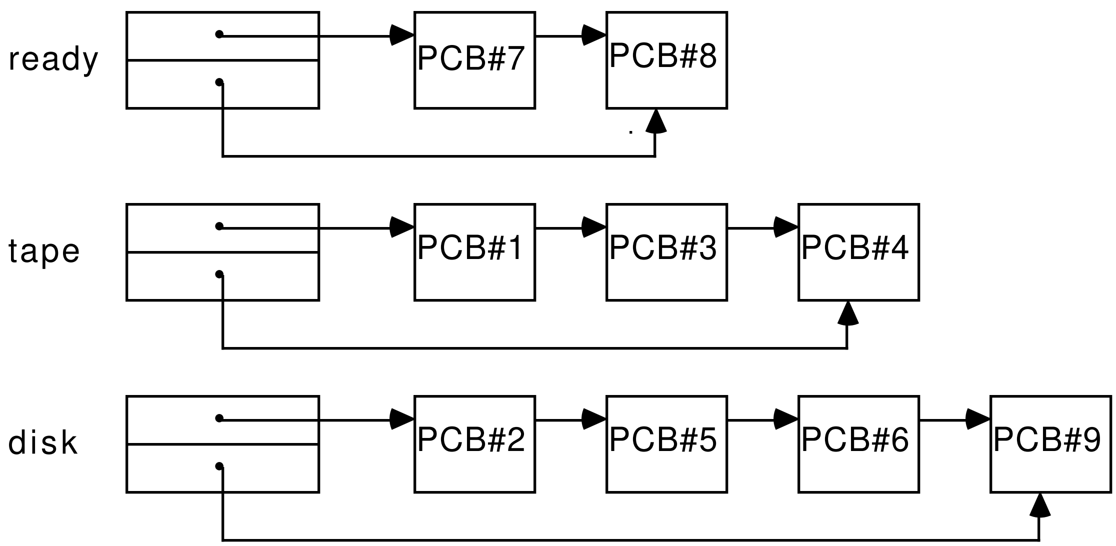
The following state diagram describes when the process can assume various states:



Queues

We want to have some process running at all times, so maintain queues of them.

- *ready queue* contains processes ready to run (really, PCBs of those processes). Note: these queues may NOT be first-in-first-out
- other queues in system: for instance, when a process asks to do I/O to a disk, it gets put in the *device queue* for that device:



Process enters system, goes on ready queue, gets CPU (a CPU burst). Then does I/O (an I/O burst); back to CPU (a CPU burst) ... until ends with a CPU burst. This is called the *CPU-I/O Burst Cycle*.

Dispatcher

This actually transfers control of the CPU to the process to be run (chosen by another routine called the *scheduler*, which assigns priorities). If there is nothing in the ready queue, 2 options:

- dispatcher can loop
- dispatcher can start a *null process*, which has the lowest priority but is always runnable; it may do nothing, or may compute decimal values of π, or may play out chess endgames, or whatever

Operation:

1. Is the current process (the one last run) to continue? If so, return control to it by returning from the interrupt
2. If not, save current process' state vector
3. Obtain state vector of next process to run
4. Give it control.

General rule: you want it small and fast.