OS services (help user use computer)
>User interface >Program execution (load/run/end, errorhandling)
>I/O operations >Communications of processes via shared mem
>File System manipulation (create/del/rename/ r/w /search, manage permission)
>Security&error detectn - handles debugging facilities (isolate/recover from error)
>Resource allocation (how to allocate among multiple jobs concurrently?)
>Accounting (keep track of processes)
(*These 3 must be done in kernel done (cannot be interrupted, most important services of OS kernel), cannot be taken out of kernel mode unlike others.)
System calls (programming interface provided by OS kernel for user to access services) (kernel has access to hardwares, user is restricted for security)
OS acts as an **intermediary** b/w Hardware & User Programs
<u>To make system calls through OS interface:</u>
1. program calls kernel want to request a certain service
2. a system call is associated with a number
3. each number refers to a specific service (I/O, time, date, file op. etc)
<u>Types of system calls:</u> (1 set corr to each OS subsys)
> Process control, file managemt, device managem, info maintenance, comm, protection
System call is like fn calls, need to pass parameters to it
<u>General methods of Parameter Passing:</u>
> Pass in registers (simplest, but may hav more param than reg)
> Param pushed onto stack by program and popped off stack by OS kernel
> Params stored in a block/table, in mem, & addr of block passed as a param (eg C pointers) in a reg
(Block & stack mtd do not limit #params being passed)
printf("hello world") is PP via stack cause "hello world" is passed as an argument to print directly. **printf("%f is the value", var)** is PP via table cause you pass var to print (address of it) to the print function. **printf(5)** would use registers for most efficiency.
API (Application Programming Interface) adds layer of abstraction than direct system calls (user -> API --(syscall)-> Kernel). Good API makes it easier for developers to develop a computer program
System Programs: help users use OS services, convenient tools for user applications & users to perform sys calls & use computer hardware. They still have to make sys calls to kernel this means that they are NOT RUN IN KERNEL mode but they may have root access (such as absolute addresses of RAM)
- Can be simply UI to system calls or even perform complex series of system calls to provide services to user apps
<u>System Program Categories:</u>
1. File Manipulation 2. Status Info 3. File modificatn 4. Programming Language Support 5. Program Loading n Execution 6. Communication 7. Application Programs (1-6) useful for program dev & execution also known as system utilities, these require root access (can access hardwares esp RAM addresses directly) Note that these are NOT ENTIRELY run in uninterruptible kernel mode, may be in user mode but just that they are carefully written to have root access.
7 - sometimes OS comes w programs that do not need direct access to hardwares eg notepad
<u>UI of OS services:</u>
GUI (eg. desktop), Terminal (use shell, UNIX shell:BASH, interprets commands & executes as indiv processes), Batch file (series of commands telling comp exactly what to do)
<u>Layered Approach: UNIX</u>
- Each layer uses services provided by layer beneath for modularity & ease of debugging {User > User Prog -(highlvl)-> Shells > API --syscall-> Kernel > HW)
- Layered cause better security and higher efficiency
Modularity can be achieved in 2 ways:
1. Layers (each layer can only interact w layers above & below (O(N) connections) Bottom layer (layer 0) is the hardware; highest layer (layer N) is the user programs.
2. Modules (each module can interact w one another (O(n^2) connections)
<u>General Idea of OS Design:</u>
- Varies widely. 1. Policy (what will be done, impt for all resource allocation, eg scheduling policy, memory alloc policy) 2. Mechanism (how to do it, technical details to implement policy. Rg how to implement FIFO scheduling policy (queue))
- Depends on hardware & purpose (user goals, system goals)

User goals - OS should be convenient to use, easy to learn, reliable safe fast | **System goals** - OS should be easy to design, implement, maintain, make users happy, while balancing b/w needs of many users (flexible, fair)
<u>OS Examples (Most OS written in C + assembly)</u>
1. Simple & Layered structure (monolithic): MS-DOC, UNIX
2. Microkernels: the original Mach
Microkernel provides (1) <u>minimal process and management</u>, and (2)<u>communication facility</u> + (3) <u>some native</u> (basic, like r/w to disk) <u>I/O operations</u>. Everything is moved to system program and user program.
Benefits of Microkernel: 1. kernel is small & lightweight (code) 2. hence easy to debug 3. doesn't need to be updated very much since its operations are very basic & fundamental
3. Object oriented: JX - single address space system, no MMU JX OS is mostly in Java except domainZero which is also in C + assembly. Each domain is an independent JVM. An instance of JVM (Java domains (A,B,C,etc)) is made whenever a Java applet is run. (Each Java app forms a new domain running on the JX OS) JVM: An abstract machine that can **run on any host OS**. Takes care of its own memory management (allocation and garbage collection). The JVM provides a **portable execution environment** for Java-based applications.
Benefits: 1. Easier to extend a microkernel | 2. Easier to port OS to new architectures | 3. Better reliability & modularity (processes are protected from each other) | 4. User-space code easier to debug
Disadvantages: Performance overhead of communication between user space and kernel space
A program is **executable**. A set of instructions that can tell the kernel what to do to create a **process image** (which is the start state when you open an app + last saved state (if any) Process image: everything about a process at a point of time (stack, heap, reg values, instr, PCB data structure)
Program on disk is only copied over to RAM to be executed. It is unchanged and stays in the disk
A program doesn't change over time, passive in nature.
Making Process: A process image from the executable is loaded into the private VM space of the process (can be on RAM or disk). This gives the initial state of a process. An entry in process table is created to indicate that a new process is going to be run in system. Then as PC moves & execute the instruction, **a process is happening**. A process changes over time, it is **active** in nature.
Process table is in RAM (in kernel space). (has all the saved states of current processes) Process image is the instructions (eg. code)
exit() in linus (deallocate resources immediately from RAM, the process table entry remains)

Process Control Block (PCB): a data structure that stores information about each process **for scheduling purposes**.
<u>The OS scheduler manage the scheduling of processes, & keeps track of all the processes' PCB in the process table.</u>

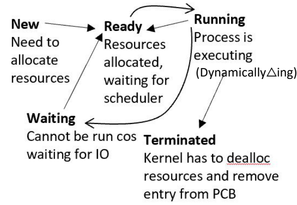
The name of each process in the system

new, ready, running, waiting, terminated

struct task_struct
```
pid_t pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```

<u>Other typical process information:</u>
1. Process state 2. Program counters 3. CPU registers 4. CPU scheduling information (priority, scheduling protocol) 5. Memory management: pagetable, value of base and limit regs 6. I/O: list of open files, connected devices 7. Accounting: time running, resources used
<u>In Process Table</u>, each entry is process control block. A Process Table (Proctable) contains the info of **ALL** processes in system.
<u>Process Scheduling Queues:</u> (3 main types, the PCB contains all process info, these queues serve as a **fast lookup** data structure for the scheduler to select a process {after selecting, will refer & update relevant PCB entries for context switch}
1. Job queue: set of all processes in system
2. Ready queue: resides in RAM, ready to exec
3. Device queue: wait I/O devices
Processes may migrate among different types of queue
<u>Context Switch</u> (save states of Pi, load states of Pj, run Pj) Interleave executions between processes in the system is an overhead because switching does no useful work apart from giving the "concurrency" illusion for single core CPUs. (Context switch time dependent on hardware support)
Interleaved Execution: processes take turns to be executed bit by bit
Concurrency: interleaving process execution gives illusion of concurrency, where all programs seem to "run" at same time
Parallelism: only multi-core systems achieves true parallelism

<u>Scheduling state</u>

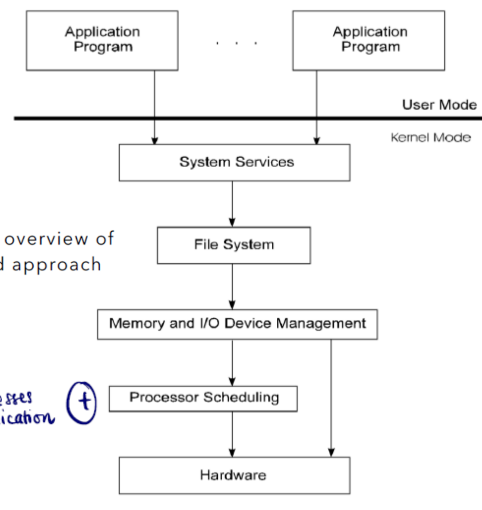


System Call Implementation

- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
  - *Recall*: it's like interrupt, but it's a *software* interrupt (*trap* instruction)
- System call interface (e.g., documented as Unix/Linux manpages) invokes intended system call in OS kernel and returns status of the system call and any return values
- Caller needs know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call (usage is just like a function or library call)
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler), e.g., libc (C) or JCL (Java)

Message Passing Example: Sockets

- A socket is defined as an *endpoint for communication*
  - Usually for network communication (e.g., TCP/IP), but also IPC within single machine (e.g., so called Unix domain sockets)
- Endpoint specified as concatenation of IP address and (TCP or UDP) port
  - E.g., **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication occurs between a pair of sockets – two flavors:
  - Connection oriented (e.g., TCP)
  - Connectionless (e.g., UDP)
- More details when we study networking



Generic overview of layered approach

inter-processes communication

1. _Context switching_ involves saving the execution state of a process and restoring the saved execution state of another process. It is a key mechanism that allows the kernel to switch the use of the CPU among different processes. Among all the ready processes, which process the kernel selects to next use the CPU at a scheduling point is a question of _policy_ (in contrast to mechanism).
2. What is contained in the text section of a process's address space?
Executable code (alt: program, instructions)
3. Use one word to complete the following sentence. A process is usually protected from bugs present in another process because the user address spaces of two processes are by default _disjoint_ (or separate, **private**, distinct)
4. Discuss why a microkernel design can improve the reliability of an operating system compared with the Unix design in which all the OS subsystems run inside the kernel.
If an OS subsystem in the Unix kernel is buggy, it can corrupt another OS subsystem (because kernel code has unrestricted access privileges). On the other hand, a microkernel runs different OS subsystems as separate user processes, which are protected from one another by default.

Process Creation: (if retval of fork() != 0 is parent)

- New processes are created by fork() system call in UNIX-based machines
- Each process can create another process, forming process tree
- Processes can execute **concurrently**
- Parents & children belong to different virtual address space, but they can share resources & communicate as well using shared memory
- Children is a **duplicate** of parent upon creation (same var, instr, code)
- Parents have to wait for children using **wait()** but not other way round. (wait for child to terminate; when child exit(), it may return something to process that called it. (child return value is stored in process table) parent have to call wait() to harvest return value of child process, if not parent will just exit & child entry in process table will not be delete (even tho child process is terminated already))
- Whether parent or child runs first depends on scheduler

Process Termination:

- Done via **exit()** sys call, resources will be deallocated by OS
- Parent processes can abort child processes (if child exceeds resources (time/mem), if task no longer required, parents want to exit)
- Child exit via sys call exit(status) {status: an int indicating whether exit process is suc)
- Parent process can receive output from child process when it waits for child process to exit: pid_t| wait(&status)

Zombie Processes

- Child processes whose parents already terminated become zombie processes
- Parent node terminated & all children processes have nobody to return to, hence turning into zombie processes.
- OS can help abort zombie processes by themselves, otherwise they will take up resources until computer restarts

Interprocesses Communication:

Shared Memory –
1. sys call to create shared mem with fixed sized
2. P1 & P2 can r/w to the shared mem in user mode.
3. need synchronisation protocol (if no step 3, overriding (p1 and p2 can be run in any order depending on scheduler))

Socket - (array location in kernel space)
1. sys call to send
2. sys call to receive

in kernel, can create new space. Overhead: need to do a sys call everytime.

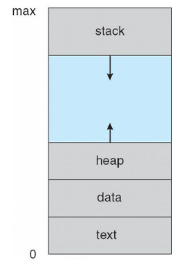
Process Creation

- Processes form a family tree!
- **Parent** process creates **child** processes, which in turn create other processes, forming a tree of processes
- Generally, process identified and managed via **process identifier** (**pid:** positive integer)
- Parent/children can share resources (e.g., opened files) in different ways
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
  - How about the parent/child in your Ubuntu shell? Do they share stdout (standard output terminal of the process)? Working directory?
- Execution
  - Parent and children execute concurrently
  - Parent can wait for children to terminate (**wait**() system call)

Process Concept

- An operating system executes a variety of programs
- Process – a program in execution; execution of one process progresses in sequential fashion
  - Program is static (a file, e.g., "/bin/ls"); process is *dynamic* (a *running program*)
  - You can run the same program *n* times (e.g., edit *n* files): one program, *n* processes
- A process defines a line of *concurrency*, includes:
  - program counter
  - stack — A program doesn't change over time, passive
  - data section
  - dynamically allocated memory in *heap* – Java manages it automatically for you; for C, you use **malloc(3)** and **free(3)**
- Textbook uses the terms *job* and *process* almost interchangeably

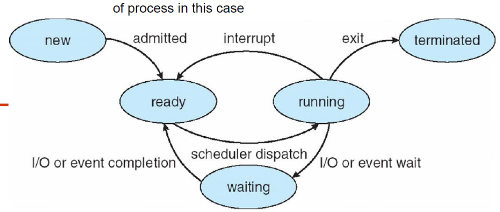
- Process also defines an *address space* (of memory)
  - Address space is *private*
  - Not accessible (by default) from another process
- Hence, process couples **two** abstractions
  - Concurrency
  - Protection
- Can OS determine direction of stack growth?
- Advantage of stack and heap growing in opposite directions?



NB: *text* section is executable code (i.e run by the process)

Process Termination

- Process executes last statement and informs OS (via **exit()** system call)
  - Output data from child *C* to parent *P* (via *P*'s executing **wait()** system call)
  - Process's resources may be deallocated by OS
    - But sometimes, process needs to still exist as "zombie" (e.g., *C* terminates before *P*'s completes **wait** for *C*)
- Parent may terminate execution of child processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent exits
    - Some operating systems don't allow its children to continue
      - All children terminated - **cascading termination**
    - UNIX has *job control* feature that defines different behaviors
      - So a UNIX **job** means a related group of processes, not a synonym of process in this case

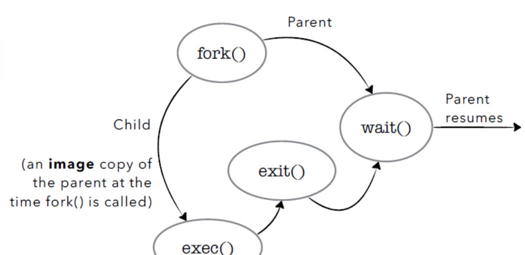


- Address space
  - Child gets its own address space, whose content is initially a duplicate of parent's (including text section that stores the program being run)
  - Child then usually loads a new program into its address space
    - E.g., shell process creates child process, but child wants to run (say) "**ls**" program instead of being a duplicated shell
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call (used after a **fork**) loads new program (e.g., **ls**) into the process's memory space

Interprocess Communication

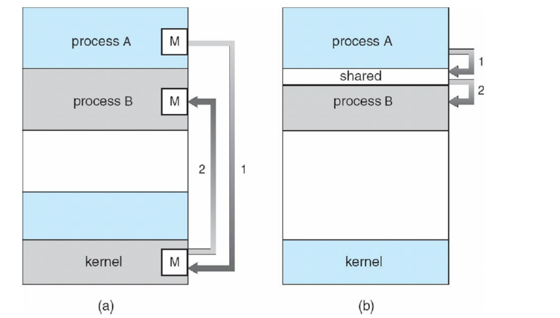
- Processes are by default **independent,** but they can also agree to be **cooperating**
- Cooperating processes may affect each other, mainly through sharing data
- Reasons for cooperating processes:
  - Share information
  - Speed up computation
  - Achieve modularity (hence protection) in spite of cooperation
  - Convenience: e.g., "**ls -l | wc -l**" roughly counts how many files you have – *try it on your Ubuntu shell*
- Cooperating processes need **interprocess communication** (**IPC**)
- Two basic models of IPC
  - Shared memory
  - Message passing

FORK() SYSTEM CALL



1. exec() replaces the current process image with a new process image.
2. Loads the program into the new process space and runs it from the **entry** point

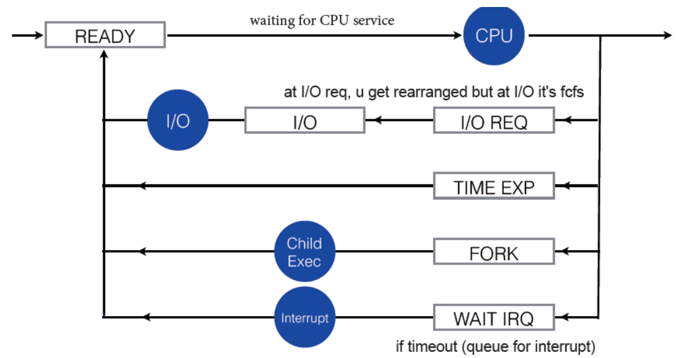
Communications Models



1=system call to send message
2=system call to receive message

[already used system calls (see next slide) to map shared memory in address spaces of both A and B]
1=simple **store** instruction (no system call needed)
2=simple **load** instruction (no system call)

PROCESS QUEUING DIAGRAM



System Programs	User Programs
Used for operating comp hardware	Used to perform a specific task as req by user
Installed on comp when OS is installed	Installed according to user requirements
User doesn't typically interact w system software bcus they run in BG	User interacts mostly w user programs (app software)
System programs run independently (doesn't run in virtual env cus they hav root access eg antivirus)	Cannot run indptly, runs in virtual env
Provides platform for running app software	Cannot run w/o presence of system programs
More EG: compiler (usually only for common prog lang like java, C/C++ we can also dl compilers but these will not hav root access)	EG. media player, games