OS is a program (made of codes) -> acts as an intermediary b/w users & hardware.

OS Goals: (Hardware <- OS <- App Prog <- Users)

1. Executes user programs & make solving user problems easier
2. Make comp system convenient to use
3. Use computer hardware in efficient manner

* Multiple CPU share same RAM -> OS manages them

Dual Mode: (Hardware support required)

- User -- (sys call (want access to hardware) ↘
- Kernel (uninterruptible)

How Does OS Run?:

Comp ON (loads BIOS) -> BIOS Scan (prep all attached D to CPU in a state for OS) -> BIOS loads MBR (Master Boot Record contains info abt disk partition & bootstrap code) -> MBR Loads Bootloader -> OS Starts (OS bootstraps & starts)

Purposes of OS: (Resource allocator & control P exec)

1. Hardware & I/O Control (RA)
2. Handles Interrupt (RA)
3. Managing Storage Hierarchy (st data integrity & consistency is guaranteed) (RA)

Reg > cache > main mem > sec stor > Disk C > Disk

4. Multi-programming: Process managem (scheduling + context switch) (both)

Clustered Sys -> ✓ ↑ service availability

✓ must write programs that utilises // programming

5. Security (CPE)

Why Multiprogramming?

- so that it gives illusion that machine can multitask = being efficient
- Allows timesharing: context switch so rapidly that users still see it as interactive computing

(* response time<1s but not too rapid cus CS is an OVERHEAD ie CPU cannot do actual meaningful work when doing context switch)

- Rules: 1) A single program cannot keep CPU busy at all times; 2) CPU always have sth to do
- Switch from Pi to Pj if Pi is waiting for I/O or turn for Pi is up
- Kernel owns Process Table (might be on cache/ram/disk)
- Context switch (1. Save all states (regs,cache,stack) of Pi; 2. Load all states of Pj; 3. Resume Pj)

Process Management:

program: passive entity, code/data

Process: active entity (program in execution)

- Create & delete both user & sys processes
- Suspending & resuming processes
- Provide mechanisms for process synch/comm/deadlock handling

System calls grouped into process control, file manipulation, device manipulation, information maintenance, communications, and protection.

How I/O Devices Work:

- CPU, I/O device controllers, and I/O D act independently of one another
- Each I/O D has a C that comes with a local buffer
- I/O Ds and Cs can run code an starts activity on its own

Need coordination to do (w interrupts):

1) CPU wants to move data to/from DC buffer from/to RAM

2) I/O happens when data moved f/t DC buffer t/f device

Diagram of I/O

If not in kernel mode, can interrupt an interrupt if u hav higher priority (but need CS) (but cannot interrupt kernel)

Vector I -> upon interrupt will look at interrupt vectors to see which D made interrupt

Polling I -> go thru each device & check who made req

Interrupt handling is done in kernel mode, by the interrupt handler for the device type

OS services (helps user use computer thru UI)

>*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&errordetectn-handles debugging facilities (isolate/recover from error)
>*Resourceallocation (support concurrency & //ism)
>Accounting (keep track of processes)

(*These 3 must be done in kernel mode (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)

To make system calls through OS interface:

1. in User mode, program calls kernel
2. A system call is associated with a number
3. Each # refers to a specific service (I/O, time, date, file op. etc)

General methods of Parameter Passing:

> Pass in registers (fastest, but may #param>#reg)
> Param pushed onto stack by program & popped off stack by OS kernel
> Params stored in a block/table, in mem, & addr of block passed as a param (eg C pointer) in a reg

(Block & stack mtddo not limit #params being passed)

printf("h"): stack; printf("%f hi",v): table

API: helps dev use OS services (sys call)

Sys Prog: helps users use OS services (are NOT ENTIRELY run in uninterruptible kernel mode, may be in user mode just that they are carefully written to have root access.)

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

Layered Approach: UNIX

- 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)

General Idea of OS Design: -Varies widely.

1. Policy (what will be done, impt for all resource allocation, egscheduling policy, memory allocpolicy) 2. Mechanism (how to do it, technical details to implement policy. Eghow to implement FIFO scheduling policy (queue))

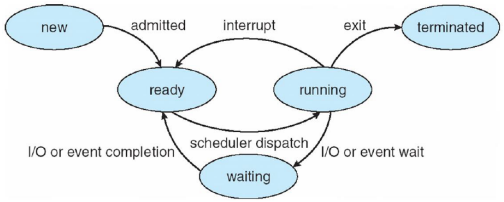
-Depends on hardware & purpose (practicality & efficiency)

1. Microkernel: provides minimal process & mem managem, communication facility & some native I/O op. Eth else moved to sys & user prog

✓ Easier to extend a microkernel
✓ Easier to port OS to new architectures
✓ Better reliability & modularity (processes are protected from each other)
✓ User-space code easier to debug
✓ doesnt need to be updated much since ops are basic (don wanna update kernel cus if anything wrong, cannot be interrupted)
✗ Perf overhead of comm b/w user space & kernel space

2. Monolithic: contains layers, many services offered in kernel mode
3. Object oriented JVM

Processes: Can run same program n times (1 program, n processes). A process is protected from bugs present in another P cus user address spaces of 2 Ps are private. Within text section of a P's address space: executable code.

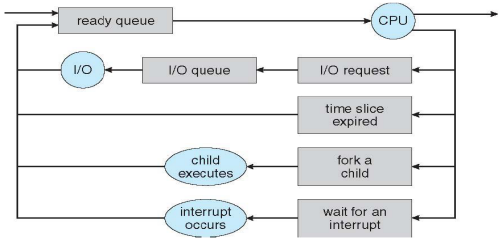


Process Control Block: a data struct that stores info abt each P for scheduling purposes. OS scheduler manage scheduling & keeps track of all P's PCB in Proc Table. Info (Process state, PC, CPU reg, priority/scheduling protocol, memory managem, I/O (open files, connected D), Accounting (time running, R used)) impt for CS

pid_t pid; // process identifier
long state // new, ready, running, waiting, terminated
// process's parents, children, address space

Queues (serve as a fast lookup data struct for scheduler to select a P)

Job queue - set of all P in system
Ready queue - set of all P residing in main mem, ready & waiting to execute
Device queue - set of P waiting for an I/O device (1 queue for each device)



Parents & children belong to different address space but can share R & comm using shared mem. Child is a duplicate of parent upon creation. Parents can wait for children using wait(). (child->exec->exit->parent(waiting) resumes)
Parent can abort child P (if C exceeds R(mem/time), task no longer needed, P wants to exit) while C can exit using exit().

Zombie Processes: Child whose parents already terminated become Z P since they have nobody to return to. (Entry of C remains in Proc Table altho it has been terminated). OS can help to abort ZP or else they will take up R until comp restarts.

Inter Processes Communication:

Shared Memory – 0 syscall, 0 amt data copied
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 couples **concurrency** (interleaved execution) & **protection** (address space isolated from one another, each process in its own VM) {expensive}

Thread gives **concurrency without protection**
Each thread has own reg&stackwithin process' space. Shares code, data & files. Eg.if thread 1 is modifying data, thread 2 will see changes (requires sync)

Kernel Thread - Known to & scheduled by OS kernel, take up kernel data struct, more exp

User Thread - Not known to OS kernel, scheduled by thread scheduler, less exp

One to one: allows multiple threads to run on multiprocessor, provides more concurrency than manyto1 but kernel thread creation overhead.

One to Many: cheap, no need for kernel thread S, cannot make use of multicore. entire P will block if a T in that P makes a blocking sys call.

Many to many (best): user can create as many user T. Multiple kernel threads required for //lism if system has multiple core

Processes	Threads
More overhead to create (bcus copying over entire VM space, including code, data & files)	Lesser overhead to create (shares data, code & files)
Requires private address space, has protection against other processes	Easier to create since they share address space, only differ in program execution
Processes are indpt of one another, no concurrency issues	Requires careful programming since threads of same process share data structures & hence are interdependent
No synchronisation overhead, easier to program and work as intended	Can potentially suffer from synchronisation overheads, harder to program & work as intended
Can benefit from parallel execution on multiprocessor system	same
A process can have many threads executing different tasks	Good for responsiveness of a process

Producer-Consumer Problem:
P&C threads have to sync st: 1) P does not write more than rate of read of C
2) C only reads when buffer has content

Race Condition:
-Behaviour of system where its output is dependent on sequence or timing of uncontrollable events(EG. thread library scheduler or kernel process scheduler is out of app developer's control) if fn call is not atomic

Rules of Critical Section:
1) Mutual Exclusion: preventing race condition, if T1 is exe-ing its CS, no other T can execute their CS
2) Has Progress (impt! otherwise if you just require mutex, a trival soln will be to allow NOBODY to enter CS ie. zero progress) If no thread/process in CS, then select process in queue that can enter CS as soon as possible
3) Bounded Waiting
Each CS has finite length, there's max # times other threads/processes are allowed to enter their CS after a P/T has made a request to enter its CS & before that request is granted. Assume that each P/T executes at nonzero speed

To prove property 1, we note that each P_i enters its critical section only if either `flag[j] == false` or `turn == i`. Also note that, if both processes can be executing in their critical sections at the same time, then `flag[0] == flag[1] == true`. These two observations imply that P_0 and P_1 could not have successfully executed their `while` statements at about the same time, since the value of `turn` can be either 0 or 1 but cannot be both. Hence, one of the processes —say, P_j— must have successfully executed the `while` statement, whereas P_i had to execute at least one additional statement ("`turn == j`"). However, at that time, `flag[j] == true` and `turn == j`, and this condition persists as long as P_j is in its critical section; as a result, mutual exclusion is preserved.

To prove properties 2 and 3, we note that a process P_i can be prevented from entering the critical section only if it is stuck in the `while` loop with the condition `flag[j] == true` and `turn == j`; this loop is the only one possible. If P_j is not ready to enter the critical section, then `flag[j] == false`, and P_i can enter its critical section. If P_j has set `flag[j]` to `true` and is also executing in its `while` statement, then either `turn == i` or `turn == j`. If `turn == i`, then P_i will enter the critical section. If `turn == j`, then P_j will enter the critical section. However, once P_j exits its critical section, it will reset `flag[j]` to `false`, allowing P_i to enter its critical section. If P_j resets `flag[j]` to `true`, it must also set `turn` to `i`. Thus, since P_i does not change the value of the variable `turn` while executing the `while` statement, P_i will enter the critical section (progress) after at most one entry by P_j (bounded waiting).

```
while (true) {

    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

}
```

Does not work for multicore system because each CPU has diff cache, each thread sees diff copies of flag[] & turn. You can interrupt a thread in CS!

Synchronisation Problems
- CS is a section where only 1 thread at a time can access: **mutual exclusion**
- Some CS must be uninterruptible, some might not. DO NOT CONFUSE CS WITH UNINTERRUPTIBLE INSTRUCTIONS
- Sometimes you want a **MAX of T threads** that can access a section at the same time or asynchronously, instead of just 1 thread at a time. This is called **condition synchronisation**. (eg.HP of a player)

sem.acquire()* -> asks kernel if semaphore>0, if yes, sem-- & let thread continue, else thread wait & put to the kernel's semaphore's queue

sem.release()* -> called by thread that is done w CS, kernel will sem++, pick some thread from queue (wake up)

Deadlock
1. Mutex 2. Hold resource & wait
3. No pre-emption 4. Circular wait
If all 4 happens, deadlock MIGHT happen. Deadlock can be prevented by removing either conditions (necessary conditions but not sufficient)

Pi --> Rj (Pi req instance of Rj); Pi <-- Rj (Pi holding instance of Rj)