Table Of Contents

Computing Systems

xV6 0.3

1) Processor;

* **Interrupts;**
  - Hardware signals (INTR signal)
  - They call Interrupt Service ~~Request~~ subRoutine (ISR) in the code. (Each interrupt calls a specific ISR)

* **IVT (Interrupt Vector Table)** (x86 architecture)
  - It stores all the ISR's addresses in order so they can be addressed.
  (This IVT is present at lower end of memory, 0x00... to ....)

---

* **Note; Polling**
  ⇒ Continuously checking status registers to check if its free. ~~to interrupt~~

  Interrupt driven;
  ⇒ Unlike polling it doesn't waste clock cycles.

---

2) Memory;

- Supplies instructions & data to the processor.

3) I/O Devices;

- Hard drives etc

Note; BIOS is present in our memory (ROM or Flash RAM) which has the bootloader (microcode), so the system can be boot strapped.

---

*Note; Boot Sector has the loader which loads the microloader in BIOS.

It resides at the top of (Level 0) (Sector 0) hard disk.

(On Boot; the bootloader is loaded into memory first by BIOS, then the bootloader loads rest of kernel)

Note; Read real mode vs protected mode of x86 processors.

Which ISR to call?

** Note; There is only one INTR pin (hardware interrupt)

** so we need to know which ISR needs to be executed, when we get an interrupt request on that pin (ie, which device generated the interrupt)

** ⇒ For this we give all our devices control to the address bus of the processor, so when a device sets the INTR, it puts its address on the address bus, so we can figure out which ISR to use

⇒ There exists a master ISR, which will tell us which ISR to use.

⇒ If multiple devices interrupt, then we use a bus arbitration protocol to decide which device to process first.

Note; OS = Kernel, Programs talk to the kernel only via system calls.

(Kernel has control of hardware) → (BASH) (Shell does this for us)

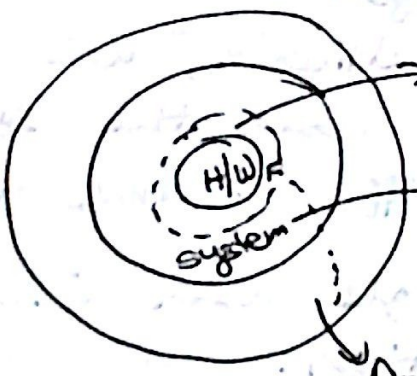
*Multitasking OS; (Mechanism to do this is a context switch) (Policy is how we assign priority)

- When a program requires data reads from hard disk & other needs the processor, then they can both be run simultaniously & when the data read is done, an interrupt is generated & the processor attends the first program again.

(If we only had polling, then the processor would unnecessarily be busy, so it couldn't run the 2nd program on the processor).

— x —

Note;

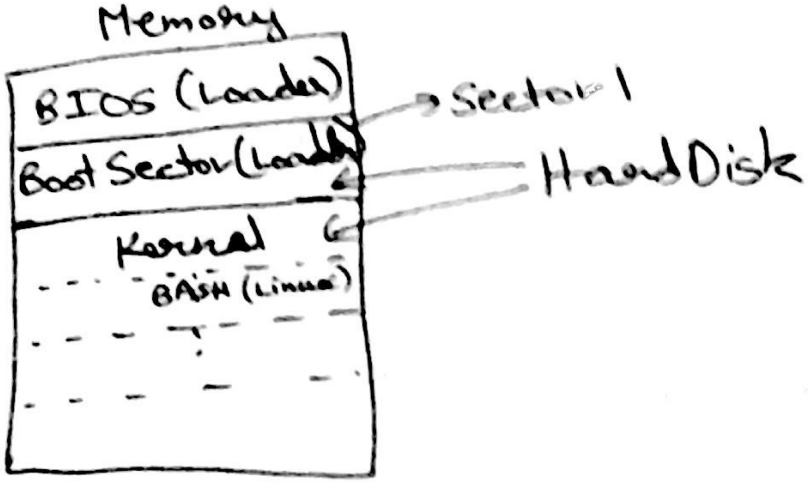


(can interact either via system calls to kernel or directly interact with system software

library calls

— x —

Bootstrapping:



(MSDOS)

Note: Micro kernel → ~~Bootstrap~~ shell is within kernel
Monolithic kernel → Shell seperate from kernel (Linux)

(OS Requirements)

Note: Requirements;

⇒ In Multiuser systems we need user isolation. (By multiuser, we mean more than 1 person uses the system, not necessarily at the same time).

⇒ Even in single user O.S, we need kernel isolation.

⇒ Program isolation, so that on context switches, other programs dont get affected by some buggy program.

⇒ From multi-tasking systems, we try move to time sharing systems.

⇒ File system is required.

⇒ Device abstraction (Make it similar to files)

* Time Sharing System;

- Similar to a multitasking system but the context switch is done based on an interrupt timer (So all processes are given equal chance to execute)

—— x ——

* Process Abstraction; (Virtualization)

- Done mainly via virtual memory abstraction.
- Each process also thinks the entire processor is for itself because of CPU multiplexing (Time Sharing System)

—— x ——

Note; Each process has a page table (per process page table), which maps virtual memory to actual memory map. This page table is protected, so only the kernel can change it.

—— x ——

* Inter Process Communication;

- A process can tell the O.S that it wants to be mapped to same physical memory to which other program is mapped to, but the program cannot specify the location directly in the ~~actual~~ physical memory to map to.

—— x ——

*** Note; Each program has a page table entry in the virtual memory that corresponds to the kernel in the physical memory. (Since every program needs access to the kernel)

OS Requirements;

1) Multi user, Multi process, reactive
2) Process isolation
3) Kernel isolation
4) File system
5) Device abstraction
6) Efficient & fair resource allocation
7) Inter-process communication.

Note: Useful Commands

1) htop } Memory usage
2) sar -B t → time } To see page faults
3) size --format=SysV a.out

* Memory Management; (Base-Bound approach) (Does not answer how all processes reside in memory) so we go for

(Without Virtual Memory) (Old Systems)

⇒ Every core has a base & offset register, when a context switch occurs & a process is loaded, the two registers are set from the 'proc' structure of the process.

⇒ The hardware checks for every data instruction if we are within range else it rises an exception.

⇒ Only the kernel has access to base & offset registers.

—— x ——

* Privilage Levels;

1) User Mode (Programs run in this mode (general))
2) Privilaged mode (Kernel mode) → Can manipulate hardware registers etc (Context switches are done in this mode)

—— x ——

* Note; The proc structure remains in the kernel (It has the number of processes that a processor can run at max) (Array).
(So for every process it can set values in the proc structure, like the "nice" value)
↓
Priority values for process

—— x ——

Note; The proc structure also has two variables base & offset assigned from hardware.

Note; (Memory management)
⇒ When we swap out a few process to create a new one, then it would be more efficient to swap out just a part of the process to be more efficient.

ex; A = 100 MB & B = 100 MB
We need to add C = 150 MB
then it would be efficient to swap A & half of B out.

(On context switches)
* Note; Schedule function is when kernel modifies base & offset registers of each process which its bringing into memory.

- When we schedule, we also store the base & offset registers so that when we get back, we can acquire the base & offset registers again.

** Program Relocation Problem; (Static)
1) Relocation Table; (Static)
- After bringing a process back after a context switch if we place it in a different location, the we need to modify base &

offset registers, and the data in proc structure.

(This is solved if we use PC relative addressing modes, but we need a solution with absolute addressing).

⇒ Solved with Relocation Table;

- Relocation Table stores a patching offset and when the program comes back in, we use that to maintain our registers. (For all instructions we have to patch offset)

* Software patches all instructions

— x —

2) * Segment Addressing Mode; (Dynamic) (Segmentation registers)

- All Data, Stack, Heap and Instructions segment have a base & offset, So when we move process out & back in, then only corresponding segment registers are modified appropriately.

Hardware

(Here instead of patching all instructions while executing, it just adds the value of Data Segment / whichever segment needed from hardware registers & executes instruction).

— x —

(Due to all these problems, and inefficiency we move over to virtual memory & paging).

— x —

Virtual Memory;

- Each process assumes it has the entire memory. (address space) or more for itself as virtual memory.
- This memory is divided as pages
- Physical memory is also divided into pages.
- Process shares address space with kernel.
- Process / kernel isolation must happen with virtual memory.

—x—

Process Memory Map; (Segments)

1) Code
2) Heap } Dynamic variables
3) Stack → Base address is randomized
4) Data } Global variables

→ Static (Does not change) (Base address are static)

—x—

Note;

⇒ With this virtual memory, we wont have problems like

a) Over-allocate
b) Under-allocate
c) Granularity of process (swapping out & in at context switch)

of memory for processes.

d) External fragmentation

—x—

*Locate Free Memory; (In Actual Memory)

1) First-Fit } Just traverse all your list of free memory locations.

2) Worst-Fit } Heap to just check & allocate maximum free space location.

3) Best-Fit } Nearest but greater than required.

——— x ———

*External Fragmentation; (This problem occurs when we dont use virtual memory)

- When small chunks of free memory is present here & there, we do Compaction so that we move the free memory to a certain location.

(With virtual memory however, we can use all those small parts of free memory without the need of compaction) (Due to paging)

——— x ———

(These are all the issues we face, so we move on to virtual memory)

——— x ———

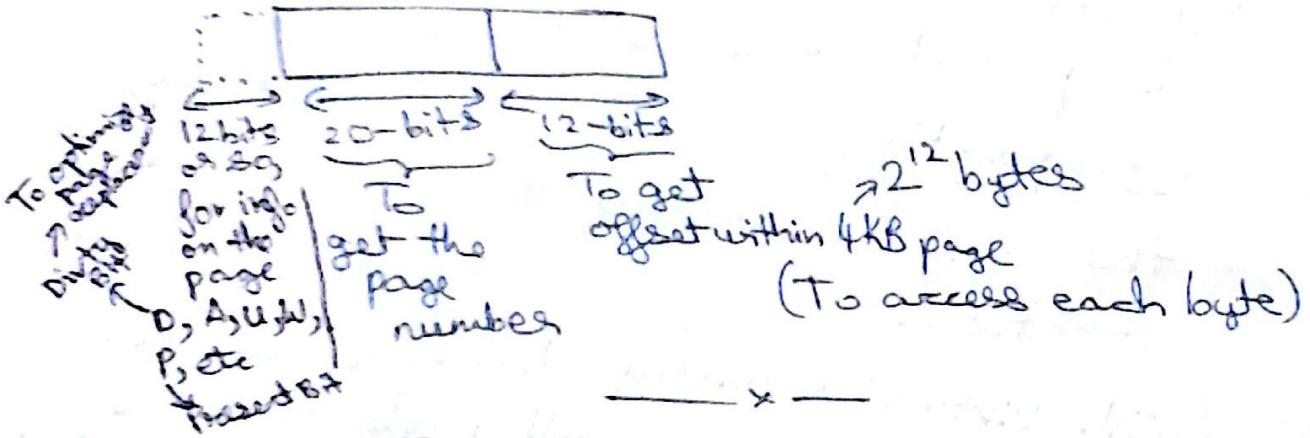
Paging; (Virtual Memory)

- Logical memory is divided into pages (in order of KB)
- Physical memory is divided into frames

→ Outside processor, memory is always (MMU) physical address based (so we can access memory properly), but within processor, they stay as virtual & physical address.

⇒ Page-Table Base register holds physical address.

Note; We can use a 32-bit identification for accessing pages



Address Translation;

- The offset bits remain the same.
- The other bits are fed into the page table, and at that location we get corresponding bits to the page number of physical address.

(So we can use this page number & offset, to access the byte in physical memory).

---

* Note; Page table is stored in our physical memory.

⇒ So in the proc structure, (CR3 register) contains the base physical address of the page table.

(Always has correct base physical address of P.T)

⇒ As we context switch a process, then the CR3 register is overwritten with the base physical address of the page table of this process (new).

— x —

* Note; Each process has its own page table & its base physical address of P.T is stored in the proc structure.

— x —

Stored in kernel space of the physical memory (RAM)

(when more kernel space is needed, other memory can be marked as kernel space)

— x —

** Note; Page Fault; (Major fault)

- When the present bit is 0 in page table entry (page number) or the physical address is invalid in the page table for some virtual address (page number), then a page fault occcurs & the kernel allocates a physical page for the virtual page. (address).

— x —

(It can be invalid even if it got swapped into the swap space).

(If this happens, the process is put in blocked state & the kernel fetches the info back (then running state)

* Swap Space; (In hard disk)

- When we require more pages than the physical memory, or the physical memory is full we swap out some pages into the swap space of our hard disk.

—— x ——

Note; In x86 processors, Physical Address Extension (PAE) is used to access more than 4GB ram (2^{32} bits exceeded) ↓To address data.

—— x ——

* Note; Every context switch, we will have to erase the cache. Since the new process doesn't use the cached data of the previous process

—— x ——

Note;

* Process Sharing; Virtual page addresses of different processes can point to the same physical page address if they need to share memory.

—— x ——

* Internal Fragmentation; (A small downside of Virtual memory)
(At max 1 page is wasted per process)

- When the fragmentation occurs not with pages but within pages (ie. of 4KB page only a part of it is used)

—— x ——

Note; a.out files etc are ELF type files.

—x—

Note; bss section contains un-initialized variables.
data section contains initialized variables.

—x—

Note; a.out files become large if there are lots of initialized variables/memory.
(size --format=SysV a.out to check memory map).

—x—

Note; Proc filesystem is not actually present on the hard disk, but its generated as a request is made to it on the fly (no file made). Therefore we can't modify these "magic" files.

—x—

~~Note; Physical Address Extension (For larger memory access (RAM))~~

—x—

Note; When page fault occurs, the same instruction is run again and the executing instruction is aborted.

—x—

Working Set of Process;

- The working set of a process is the minimum amount of memory needed for the process to run. (Without swaps to disk).

—x—

Thrashing;

- When $\sum_{i} WS_i \geq$ Physical Memory
  (Working sets)
- Then many swaps to disk happen so the system is slowed down a lot, this is known as thrashing.

—x—

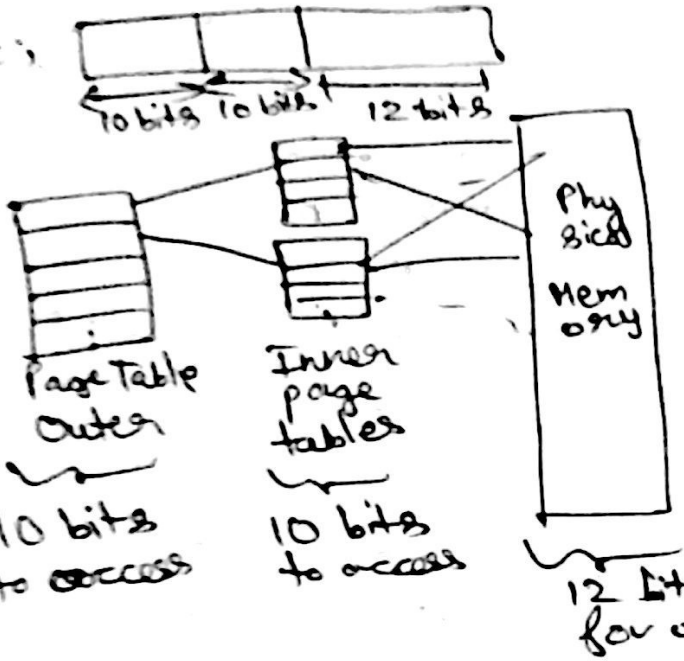
Note: Swap space is optimised to access by the OS. (So pages can be accessed quickly & efficiently). The filesystem structure of swap space is different

—x—

Two level page table; (Disadvantage of 3 memory accesses)

- Page the page table itself,

ex:



(Outer page table = page directory)

(Inner page table = page table)

- This helps us save space since we need not store some inner level page tables if they are not needed (holes are present, ie. they dont use those addresses), so we dont need to store page tables at those addresses)
- We can also swap inner page tables into swap space if needed.

—x—

Note: Inverted Page Tables (For Mid 1)

—x—

* Note: In Intel processors, Segmented paging is done (virtual address has segment & offset)
→ We get a translated 32-bit address from segment translation and then get physical address from page table.
⇒ The paging is optional (since there is no paging on boot time).

———x———

Translation Lookaside Buffer; (TLB) (Access in 1 clock cycle)

= (on chip data structure)

- Its a small buffer which stores some page table entries, for fast access.
- If TLB has mapping, we use it else we use the page table and we also store that in the TLB.

———x———

Note:
TLB Miss: When a mapping of virtual→physical address is not found.

———x———

Note: TLB's are associative memory.

———x———

* Note:
At a context switch, the TLB is flushed since there is only 1 TLB for all process, and the mapping is different from one process's virtual addresses to another process's virtual address.

———x———

Page Replacement Policies;

(To move a physical page into swap space)

The "victim" page (For TLB, to remove entry)

=> On a page replacement to swap space, the kernel stores where in the swapspace the page is present in.

1) First in - First Out;

=> This is bad as from Belady's anomaly, increasing the page table size (number of page frames) increases page faults.

2) LRU replacement;

=> Replace page which has not been accessed for the longest time. (Least Recently used)

=> this is an online algorithm

(There is another offline algo (we have entire reference string), so we remove farthest page which is ~~least~~ used in the future).

—— x ——

* Note: Swap Daemon / Page Daemon
⇒ Pro actively swaps pages into swap space.
(kswapd man page in linux)

— x —

*** Copy-On-Write (COW);

- When a process is forked, then the virtual memory map ~~is~~ for child is a replica of parent's page table's ~~physical entries~~. So that we dont need to allocate physical addresses for the child. But they are all marked with COW bits.
- So when either parent or child try to write to that entry, a new location is allocated with the value copied to this new physical page (The old entry is removed in this process page table)

— x —

*** mmap (system call) (Faster than fread)

- Given a file descripter & file size from (fstat) mmap can copy the data onto some memory (addresses) in the kernel & we can access that memory to print to the screen etc.

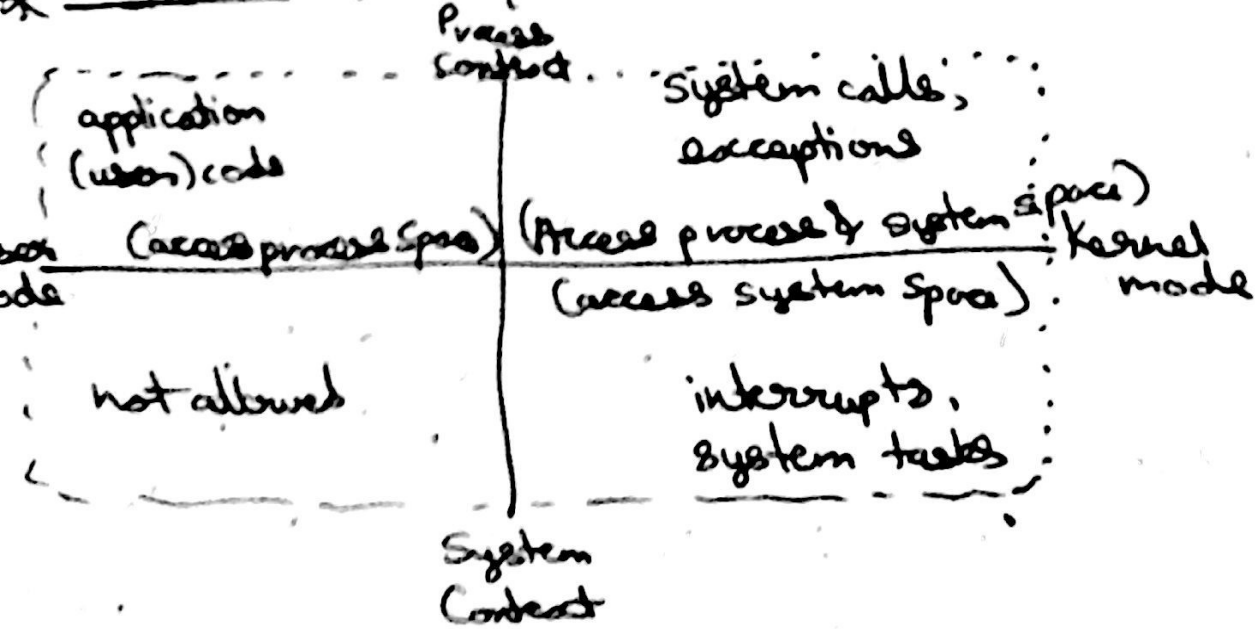
— x —

* Per process data structures:

* 1) u-area (only when process is running, this is accessible)
   - ⇒ Has file descriptor tables (Files process is using)
   - ⇒ Stores process control block (hardware context) when process is not running.
   - ⇒ Kernel Stack (Per kernel stack) → when kernel executes system calls.
2) proc structure:
   - ⇒ Process id
   - ⇒ Location of u-area.
   - ⇒ Current process state (READY, STOPPED, BLOCKED

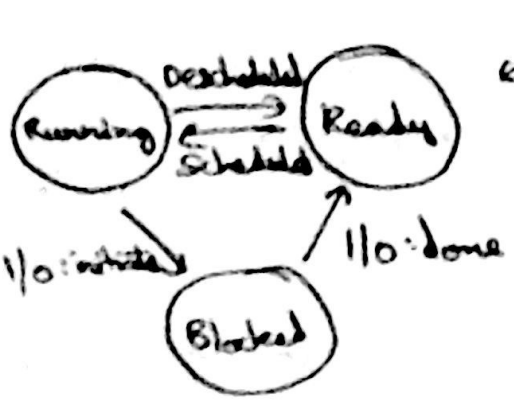
---

* Mode | Space | Context;



Q) Why can't kernel also use user stack when executing system calls/etc in a process?

Process State Transitions;



← Simplified version.

———x———

* Note: When a sleep(x) is present in child after fork(), generally parent is executed first, but it is possible due to high load, these processes are given the CPU only after x-seconds so now randomly the child can come first as well.

———x———

* Scheduling; (Context Switches)

1) Preemptive Kernels; (Schedules)

- When the time slice allocated for kernel is over, we context switch out.

(This has problems)

* (Synchronization problem: Suppose we are removing a node from double linked list, if we get swapped out inbetween, we might not have completely removed the node yet.)

(Locking techniques are used with multicore systems)

2) Non-preemptive Kernel; (Scheduler)

- Another process cannot preempt the kernel even if its time slice expires.
- This solves problems seen before (synching problems)
- But there are better ways to do this, which we will see later.
- However if an interrupt occurs, then the context switching will be preemptive).

Note: Scheduler is just a program, so who loads the scheduler? (This is the successive problem (scheduling problem) we encountered before).

* we want to call the scheduler when,
  a) Time slice expires
  b) I/o request
  c) Sleep

---

* Note; In a context switch the scheduler stores process state of p1 & then loads process state of p2. This is the cost of context switch (We want to reduce this time).

---

* Scheduling Metrics;

1) Interactive Applications, (Shells, Editors etc)

⇒ Response time ⇒ (first response time − arrival time)

2) Batch Applications (Compilers etc)

⇒ Turnaround time ⇒ (completion time − arrival time)

3) Real time applications (Video players → soft real time etc)
⇒ Strict deadlines
Missile trackers → Hard real time

(Note: jobs ≡ processes)

Note: In batch applications, if scheduling is done based on first come first serve, then if a long process comes first, it blocks small process that come later. So turn-around time is not minimized.
(This is the convoy effect)

Shortest Job First Scheduling; (Batch Applications)
(Optimize turnaround time)

- We assume we know time each job takes, so we will always schedule the shortest job first.
- In this we will get the minimum turnaround time, so convoy effect is not present.
- If a new process comes in which takes lesser time than the running process remaining time, then we execute the new process. (shortest job first).

Fairness; (In Scheduling strategies)

- Shortest job first scheduling is not fair always since a long job might end up starving if small process keep coming in.

Round Robin Scheduling; (optimizing response time)

- We use time slices to allocate each process some CPU time.
- We access all jobs as soon as they come in so our response time is small, but

the turnaround time for this approach is not good.

- Increasing time slice length, turnaround time decreases but response time rises

- This uses a lot of context switches which wastes time.
- Starving does not occur here generally, so its fair most of the times.

Multi Level Feedback Queues: (Scheduling in real life (We dont know if its batch or interactive)

- With 2 jobs A & B,

a) If Priority(A) > Priority(B), then the process A is scheduled

b) If Priority(A) = Priority(B), then the processes A & B run in round robin scheduling.

* ⇒ Priority Assignment,

- When job enters, its given highest priority
- If job uses entire timeslice, then priority is ~~decreased~~ dropped. → Batch Application.
- If job gives up CPU before timeslice is over, it stays in the same priority level.

⇒ This priority assignment ensures that interactive process are getting round robin and batch processes are getting shortest job first scheduling (similar).

⇒ We also modify (decrease priority level) if process has "CPU usage" which is very high, so that starving processes are addressed.

⇒ Priority value of process = Base priority + nice value + CPU Usage

(Lower priority value ⇒ more priority)

$CPU_i = CPU_{i-1}/2$

$P(i) = \alpha(CPU_i) + (1-\alpha)P_{i-1}$

- $P(i)$: Priority value when the process comes in to execute the i^{th} time. → (Lower $P(i)$ → more priority)
- CPU_i: Current CPU usage time (Increases as time passes) when process is using CPU
- P_{i-1}: History (Decay Function) → History decays.
- α: Convex combination of CPU_i & P_{i-1}

Note: Credit based scheduling

Schedule function; (of Kernel)

- When a running process goes to sleep or ends then, the processor goes from user mode to kernel mode.
- Then the schedule() function is called from the scheduler (The one with highest priority is scheduled next).

Note; On a context switch all user area details of a certain process are stored in one memory & we just store an entry in over page table which we can access later when the process is brought back.

---

* Note: We can use tools such as perf to see details about page faults of your program etc - (perf stat -e cs,migrations file)

---

* Note: If we don't use CPU consume times with decays, then a batch process can disguise itself as an interactive process by relinquishing the CPU just before its time slice ends so its treated as interactive & has high priority (But with accounting CPU consume time, the priority is dropped as more CPU is used (time duration) within timeslice)

Note: An application can either have a single process or a collection of processes.

---

* Note: When we execute a set of processes instead of a large single process (in multicore), then if one of the processes is doing a file I/O, the other can continue executing, on with the next process and so on. (In one large process the CPU stays idle on page faults, I/O etc) (So we can overlap I/O with computation).

Inter process communication; (IPC)

1) Pipe; When all processes are on the same system.

2) Socket: Processes talk via the ethernet. → Processes can be on different machines

3) Remote Procedure Calls (RPC)

4) Files (Bad performance)

5) Signals

6) Shared Memory

* Threads: → Effective alternative of processes (generally). → Output is non-deterministic (No order of threads)

- Each thread has its own stack but the same heap, initialized & uninitialized data segment etc (Address space is same).
- So they can independently execute the same process.
- Threads are implemented via the clone system call.

Note: Each thread runs on a single core of a processor. Generally threads use core with least load, so threads keep changing cores as needed, but on specifying a certain argument, we can force the thread to run on the same core.

*Note; We use threads since they are not as expensive as processes created by fork().

Note: The clone system call is not very expensive, so pthread_create() etc use this system call.

---

* Note: User Threads → [N:1] } The user sees it as N threads but the kernel sees as 1 thread
  Kernel Threads → [1:1]
  Else don't even get the advantage of I/O, compute overlap

Threading Model of System

- Since the kernel views all user threads as one, there are no actual context switches. (No performance benifits since kernel views it as only one thread)

⇒ So nowadays, linux only uses kernel threads since they get larger time slices compared to user threads.

⇒ Hybrid threads → [N:M] } Some O.S's use this. (And we also have performance benefits)

- User threads are managed by user-level library (Not in course)

Note: User threads are faster than kernel threads since they are essentially just a function call.

* Advantages/Disadvantages of Threads:

Processes	Threads
1) IPC is hard here	1) IPC is easy here.
2) Each process has its own address space, so large address space.	2) Since the same address space is used, the stack is shared among threads, so small address space.
3) Offers complete isolation since one other process cannot effect this process.	3) If the stack of one of the threads is messed up (malicious), it can end up affecting other

---

Since any halting system call from a single thread blocks all other threads since the kernel only sees one thread. (User threads model)

* (So user threads model is useless).

↑ OS might also end up poorly scheduling these threads since it doesn't know about them

Note; To the kernel, threads are essentially just processes. (But they were created easily).

- Threads were together forming a process.

* Locking in threads; (Mutex)

- pthread_mutex_lock(&lock)
  Critical section
  pthread_mutex_unlock(&lock)

} With a mutex

—x—

*Note; As we increase the granularity of the locks, (smaller critical sections), then the parallelism increases.

—x—

*Note; If a thread puts a lock & then on a context switch, what would the other thread do if there is a lock? (We will see later that there are 2 types of locks) → It just goes to sleep.

—x—

* Note: Process context switch is expensive since it changes address spaces of executing process whereas a thread context switch only the registers etc are changed, so its fast.

* • Note critical sections should be as small as possible (least number of instr.) (We should not have I/O, etc functions in this critical section).

—— x —— (since they can end up holding lock for too long)

* Coarse Grain vs Fine Grain Locks:

- Coarse Grain → ~~Less~~ Less parallelism
- Fine Grain → Too many calls to lock & unlock function calls. (Almost 5x time). in some cases.

* Variable Privatization: (To reduce lock calls).

* ⇒ When our global variables used in our threads are reducible, then just use a local variable in each thread & then at last update the global variable. (This might not always work).

⇒ Compilers might also do this automatically.

—— x ——

*** (In parallel prog) (Cache Problem).

Note: Write-Invalidate-Stove [ping-pong problem]

** (Try this!)

- Suppose we split / ~~parallelize~~ parallelize a function which sums up integers / $\sum_{i=1}^{N} i$.
- We tend to keep a global array, lsum[4], and split our task to 4 threads, each updates lsum, and once all threads are complete

we just print the sum.

⇒ The problem here is that the entire lsum[4] is stored in a single cache block, so each thread invalidates the cache of other cores on write to its lsum[threadno]. So this wastes a lot of time (Parallel ends up being worse than serial).

- However if we use seperate variables most likely this problem will not occur. (So parallel will be better or equal to serial) (timewise)

↳ Or we can use local variables & add to global variable at last with small lock.

Variable privitization.

---

* Lock Implementations;

1) If we are using a single core processor, disabling & enabling interrupts might work ↗ This can work in single core but its dangerous since a crash would ruin the system. (This is bad) ↘ in program

⇒ Disabling timer interrupt can stop context switches

Note: We cannot context switch out of an ISR until its complete.

---

* Deadlock; (Beware)

- If our code locks a variable that an ISR needs to use, then the system goes into a deadlock. (ISR wont finish).

(This generally can happen when programming with kernel code).

#2) In lock call, make it wait until lock is available and then consume the lock.

⇒ This can be a problem if there are 2 threads waiting for a lock on different cores, so once its available, both the threads can end up entering the critical section. (Since they are on different cores, they run parallelly)

- So this is a problem,

#3) Peterson's Algorithm: (2 threads who want to access lock)

- Each thread tries to be generous. (Initialize turn=0)

```
void lock() {
    flag[self]=1;
    turn=1-self;
    while((flag[1-self]==1) && (turn==1-s
        wait here;
}
void unlock() {
    flag[self]=0
}
```

⇒ Self =0 for 1st process and self=1 for 2nd process.

- This solves the problem since if flag[other thread]==0, then the one thread which requires the lock

can use it as soon as its free.

- If both threads want the lock, then the thread whose's turn it is waits and acquires the lock as soon as its free.

** Fairness; (or Liveness)

⇒ Its fair since if two threads are trying to acquire a lock, as soon as 2nd process(1) acquires it & uses it, then flag[1] becomes 0, so the other thread can enter the critical section. Even if for some reason the 2nd process comes again & makes flag[1]=1, it would make turn = 1-1=0, so the 1st process (0) would use the lock.

(Hardware feature)

*4) TestandSet; (Not always fair)

- This works will any number of threads.
- Since the instruction 'TestandSet' is an atomic instruction, only one thread will get the lock at a certain time.

⇒ TestandSet(&lock->flag, 1)
would do; old = *(&lock->flag)
*(&lock->flag) = 1
return old.

- All this is done atomically.

⇒ In the lock function,
while(TestandSet(&lock->flag,1)==1)
we wait.

- So now only one of the waiting threads for the lock acquire the lock & set the flag to 1. (Atomically).

⇒ this is not always 'fair' though.

—— x ——

5) FetchAndAdd (This hardware instruction is much more expensive to add to a hardware

- Another atomic instruction.

⇒ FetchAndAdd(int *ptr){
    int old = *ptr
    *ptr = old + 1
    return old; }

- On locking, each thread gets its ticket, by, myticket = FetchAndAdd (&lock->ticket)
⇒ And it waits until it gets its turn

- On unlocking, FetchAndAdd(&lock->turn) happens

⇒ So void lock (...) {
    myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ..... (spin)
}
void unlock (...) {
    FetchAndAdd(&lock->turn)
}

—— x ——

After Mid2

Problem with Peterson's Algorithm:

- When instructions are reordered in each of the processor, then the checking of peterson algo can occur before assigning the turn ~~variable~~ (suppose we use 2 variables he_wants & shee_wants, then both threads can end up entering critical section).

⇒ There instructions, he_wants=1

```
while (she_wants && turn==his)
    //Git
) she_wants=1
:
while (he_wants && turn==his)
    //Git
```

⇒ They get reordered, so both threads end up entering critical region.

- Individually (within each thread) there are no dependencies b/w the instructions so re-ordering seems reasonable to the processor, but concurrency becomes a prob

Solution: (Hardware instruction) (Fence)

⇒ We use memory fences, to solve this problem. When we add a memory fence instruction, then this forces the CPU/compile to ensure that no instructions are re-ordered across the fence.

- she_wants = 1;
  turn = his;
  Fence → _sync_synchronize();
  while (he_wants && turn==his)
  :

- he_wants = 1;
  turn = hers;
  Fence → _sync_synchronize();
  while (she_wants && turn==his)
  :

Note: We will also face a problem if the two ** instructions (she_wants=1, turn=his) and (he_wants=1, turn=hers), are interchanged as well. So we would think of using another fence, but memory model tells us that writes wont be reordered with other writes. (So we wont get this problem)

* Spin vs Yield:

- If we will get the lock in the near future then yielding is pointless since it wastes Context switches. Spinning would be better in this case.

1)
```
void lock(){
    while(1){
        if(TestAndSet(...)==1) count+=1
        else return
        if(count==1000){
            count=0; sleep(1);
        }
    }
}
```

- This algorithm is fine but going to sleep no process is sure when it will get the lock. (No fairness)

(Its chance to get the lock might have been lost when it was sleeping)

(Code in slides (Yielding locks using Queues))

#2) To ensure fairness, we add a queue, and, if threads don't get access, we park them in the queue. Then when the lock is released, the unlocking thread unparks the thread at the head of the queue.

⇒ Park() puts the thread to sleep ready to be awaken when Unpark() is called.

- To maintain concurrency in the queue, we use a guard lock when modifying or accessing the queue.

Note:
⇒ Using a spin lock for the guard is fine since our critical section is very small.

a) Problem with this: If thread is added to queue and guard is set to 0, and then before parking context switch occurs, and another process unparks this and removes it from queue, and the context switching back to the original thread, it parks and the process stays parked forever.

Solution: We add a setpark() before making m→guard=0 and then park(). (Setpark() tells the kernel that we are going to park soon so don't context switch until we park()).

Note: These park(), setpark() etc are Solaris based not Linux.

---

* (Condition Variables) (Use volatile ints for globals) (Code

* ⇒ We could just spin wait until child finishes execution with a global variable but it wastes time. (Due to spinning).

* 1) We use pthread_cond_signal & pthread_cond_wait since this mechanism is a lot faster. Semantics specify, that we need to use a lock around these functions.

* * The pthread_cond_wait releases the lock before going to sleep and then when it is signalled by the child thread, then it will wait until lock is released by the child & then acquire the lock again. ~~(This is done atomically)~~

⇒ So a single function pthread_cond_wait releases lock before sleeping then acquires lock on waking up. (Atomic)

Note: We dont want to call pthread_cond_wait if there is no child to signal, so we use a global variable 'done'. (In code), so that we call the wait only if done==0.

Note: ~~⇒~~ If we dont use locks with these condition variables, then context switches can mess things up. → See code without locks to understand why.

⇒ Producer Consumer Problem; (1:1)

- When there are 1 consumer & 1 producer, then we can use a single condition variable.
- If item is read, writer is signalled. If writer writes reader is signalled.

—x—

(Linearisability) (Isolability) (Correctness of concurrent programs)

Linearisability;

- This is required in concurrent processes.
- The manufacturer specifies memory consistency models, ex: Relaxed, Sequential etc. → These are always satisfied, but we need to add additional synchronization primitives inorder for our concurrent programs to work.

⇒ Not reordering all instructions & restricting this reordering is called Linearisability.

—x—

Note: We say a datastructure/etc is linearisable if at any instant, a snapshot of the system would give us an interleaving of instructions from multiple threads.

—x—

(Without unwanted reordering)

We want valid reordering if any. ← without reordering which might make our system not work

⇒ We also should be able to assume all instructions are atomic in our given ordering & then it is linearizable.

—x—

For a linearisable DS and it works out fine

⇒ Even if 2 cores run instructions A & B simultaneously we need to guarantee that either A, B happened or B, A

—x—

Not half of A, then B etc ...

Producer - Consumer Problem (N:M) (Code is slide)

(Code 1) (No fairness) (Condition variables)

When we use same conditional variable then producers can end up waking other producers or consumers wake up other consumers.

———x———

- Note: If there are no waiting processes, signal does nothing.

———x———

** Note: Model checking (Exhaustive checking) is used to verify that our code is correct (Not a mathematical proof) (Exhaustive state space enumeration).

———x———

⇒ In code 1, suppose there are 2 Consumers & 1 producer, then suppose C1 goes to wait, then P1 signals so C1 goes to ready to run state but waits for lock. Suppose C2 consumes lock after P1 releases it and then reads the data removing it. Then C1 acquires the lock but there is no valid data. (This is a problem).

- The way we fix this is by replacing "if" with "while" in both producer & consumer, so that only one consumer can consume the data. (So consumer awoke consumer etc just puts them to sleep again)

~~Bad: no fairness~~ ~~Thread starvation~~

* • Due to this we can end up going into a deadlock state. (when there are multiple producers & consumers).

* ⇒ Suppose a producer is in wait, and a consumer reads data, then goes & signals another consumer then goes to sleep, so that would try to read data, but since there is nothing, it also goes to sleep, so all go to sleep and we get a deadlock.

———x———

(Note: We can fix this with 2 condition variables)

———x———

* Note: Signal → Wait → Ready To Run → Scheduled → Got Lock → Come out of wait

• The problem occurs when another process gets lock & changes state of shared variable

Here we solved with while loop.

———x———

(code 2)

* II) Using 2 condition variables & while loops our code works.

• A consumer only signals the producer and vice-versa.

———x———

* III) When there are n-locations in a circular queue. (Code 3)

• We use Similar code as (II) but we just change put(i) & get() functions

———x———

* Semaphores: (Code in slides) (Readers-Writers problem)

- How are post, wait implemented in semaphores (pthread source code).

⇒ We can implement the readers writers problem with semaphores.

* (Problem in code 1)

I) The code given must ensure that the put() & get() functions are guarded since they can simultaneously be accessed by multiple producers/consumers

* II) (Problem in code 2)

- If Consumer sleeps with guard mutex, the producer cannot get the lock, so we get a deadlock.

* III) (Code 3)

- This code works fine, since we only lock critical section. (Generally its bad to sleep/wait while holding a lock).

———— x ————

———— x ————

Note: Read about Model Checking.

*Dining Philosophers Problem;

(Code in slides)

⟹ The problem is that all philosophers can hold their left fork & wait for right fork, so we get a deadlock.

Solution: Make one philosopher the leader who gets the right fork first whereas all the others get the left fork first.

———x———

Concurrent Datastructures

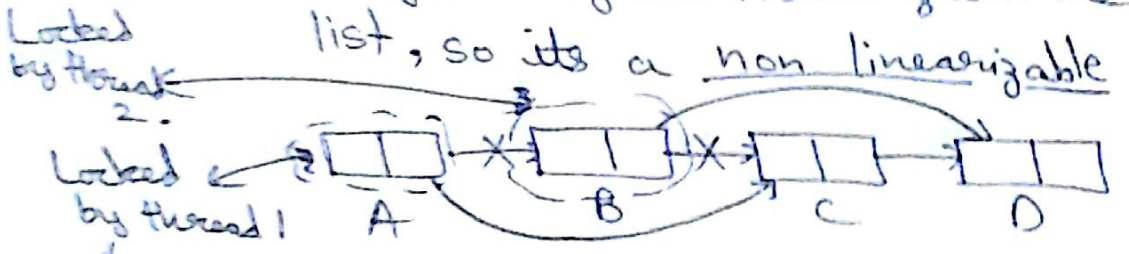
I) Making Linked Lists concurrent: (with Locks)

(Slides have diagrams)

1) Using lock variables in the struct of a node itself. Each thread needs only 1 lock to modify that

⇒ Suppose each thread can access functions like add, remove etc on nodes only when they have acquired the locks.

- When 2 threads try to remove adjacent nodes, we might end up removing only one of the nodes from the linked list, so its a non linearizable thing.



So we can change pointers of A.

⇒ So now our linked list is A, C, D. So C has not been removed.

2) ~~bite use 2 locks~~ (Hand on hand) (Slides for diagram)

Same as above, but each thread needs to hold 2 locks to insert or delete a node. (The node & its predecessor)

⇒ So now, one of the thread would have to wait till adjacent deletes/additions have been complete.

(But if they are not adjacent, then both deletions/additions can happen parallely).

Note; We could just use a global lock so only one thread can modify the list at a time but the level of parallelism would be very low then.

⇒ In our hand-on-hand method, the threads can act parallely if they are not modifying adjacent nodes.

---

* Lock free - Datastructures;

I) Concurrent Stack; (Slides)

- We use the CAS (Compare & Swap) operator to implement this. ← Atomic so we try use this.

* 1) Lock-free push:

- First we do node→next = head;
- We use CAS to compare if node→next is equal to head, then we make head = node.

⇒ We perform the above 2 operations in a loop, so if another thread adds a node to stack after we do node→next = head then CAS fails, so we go and loop and try to make node→next = head', CAS succeeds & we make head' = node.

* 2) Lock-free pop:

- First we do current = head } temporary pointer.
- Then in a loop we do,
  - ⇒ CAS(head & current) → break on success
  - ⇒ current = head. ← on failure.
- So if head & current point to same node, then CAS makes head = current→next.

⇒ If CAS fails, it loops and tries again.

** Note: CAS is atomic, (execution), but
** setting up its parameters is not atomic (since we need to put them in registers).
(This is called the ABA problem)
⇒ This can end up messing everything where we used CAS till now.

—x—

* Note: In the pop operation, ABA problem
** will make the operation fail if previous push operations have used recycled memory, so context switches b/w assignment of arguments & execution of CAS can mess up the stack.
(So whats the solution? → ~~Dont know!~~)
↓
We use loadlinked, Storelinked instructions which would automatically update the registers if we are using recycled memory, so that no problem occurs with the pop operation.
~~Still~~ Still this is not very clear.

—x—

*Recap:

Note: Interrupts & DMA controllers speed up the CPU (ie no wastage of CPU time).

Note: Memory mapped I/O, just assigns parts of memory to I/O device registers, so load/store instructions work & its fast. In just I/O, we use IN, OUT instructions to copy data to/from device registers to our registers, this is also quite fast, but limited in no. of available registers.

- Device drivers must be given along with devices so they can be used.

Note: library function → system call → V.F.S (virtual File system) → FileSystem → Device drivers. → device controller → Device

(Suppose we try to read/write files)

*Disk:

- Platter → Track → Sector } Granularity

⇒ Read/write happens at sector level.

⇒ Parameters affecting read times:

1) Seek time: To bring read/write head onto correct track.

2) Rotational delay: To get to the correct sector (Latency)

3) Transfer time.

→ The device drivers can do all sorts of tricks in assigning sector reads/write so as to minimize average read times.

---

* Block Size (No. of sectors in a block)

→ As we increase block size, fragmentation may occur as blocks may not be full.

---

* Track Skew:

- We keep a little offset of next sectors on a new track (i+1) once sectors of track (i) are full since we don't want to pass the sector before our seek head changes track.

---

* Disk Head Scheduling (Tracks)

1) Greedy (Nearest track first) (Unfair)
2) First-come-first serve. (Slow)
3) SCAN (Go to leftmost & then rightmost & repeat).

Secondary Storage: (Non-volatile persistant storage)

1) Capacity
2) Persistency
3) Performance (R/W speeds)
4) Price

Making wide a simple source drive

Redundant Array of Inexpensive Disks: (RAID)

I) RAID 0:

- Multiple hard disks

⇒ Increased capacity & R/W can happen in parallel.

(No fault tolerance)

II) RAID 1:

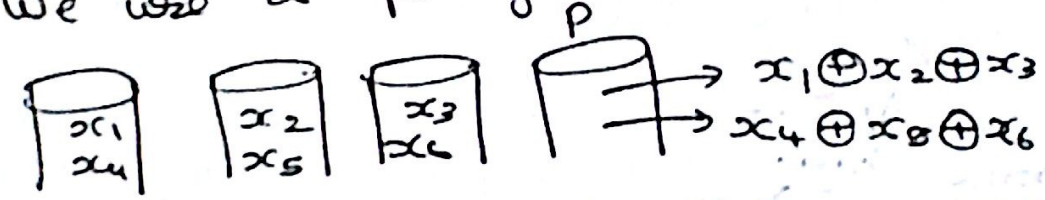
- 2 Disks store copies of the other 2 disks.

⇒ Good failure tolerence

- Bad capacity.

III) RAID #4:

- We use a parity disk.



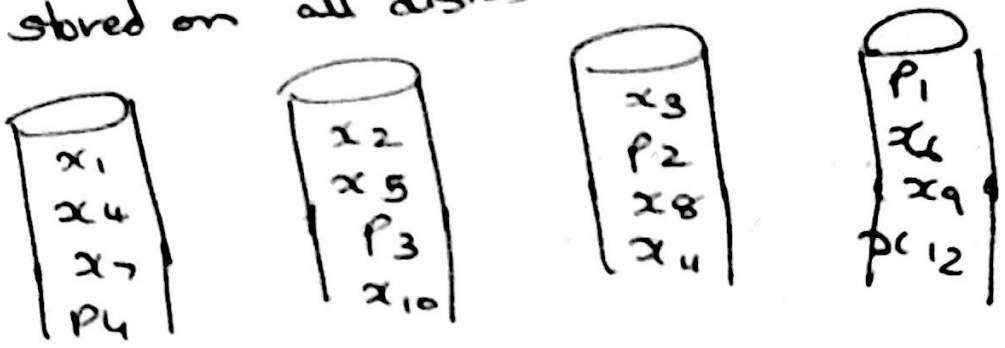
⇒ $x_1 \oplus x_2 \oplus x_3 \oplus P = 0$, if this is false, then this means there is an error.

- Failure tolerance upto one disk

⇒ Bottleneck on parity disk.

IV) RAID 5:

- Interleave the parity disk. so its stored on all disks



⇒ So the parity bottleneck is gone.

—— x ——

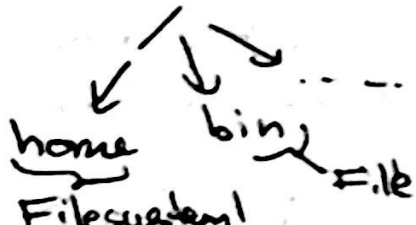
Note: Storage area networks

—— x ——

* Filesystems;

—— x ——

- Note: we can stitch multiple file systems together to make a directory structure.



—— x ——

* Note: Bootblock, Superblock, DMap, Imap
(See Mail) ↑ Sector 0
Metadata
Info about filesystem
Blocks that are allocated & empty
Contains inodes ↓ stores permissions etc for files
Copies are stored to be used on failure
Metadata & metadata
Sector 1

Structure of filesystem →

* • The memory allocated to a certain file is stored. These inode index structures will be small, So we use indirection (or double or triple) so that we can store memory allocated to large files.

⇒ Hard Links count is also stored in inode.

⇒ Different users can give hard links to a certain inode file, so that it can be shared by multiple users. (Essentially a copy)

—— x ——

Note: Hard links cannot be made across file-system boundaries. straddle across.

—— x ——

Note: Symbolic links can be made across file system boundaries. (inode hard-link count does not change).

—— x ——

⇒ To add memory to a file:

1) Update DMap & get a free block.
2) Update inode index structure by adding this blk
3) Write data to the block.

⇒ If a crash occurs after step ①, then we will have a block which will be wasted & never get unallocated. (space leak)

⇒ We want to do all these three instructions atomically.

- fsck is a file system check which happens on boot of the system (Generally) (Time consuming).

—— x ——

* Transactions: (As in DBMS)

- Journalling file systems, which keep track of the steps performed.
- When a crash happens, we try undo partial transactions.

(A transaction consists of the 3 instructions we have seen before).

—— x ——

Note: Memory is freed only when all hard links referring to that memory are removed.

—— x ——