Grading System

midterm + quizzes
final + quizzes
Assignments
Attendance
Projects/research

h_zarandi@gmail.com
h_zarandi@aut.ac.ir

Class Content

1. Introductions to Operating Systems

References

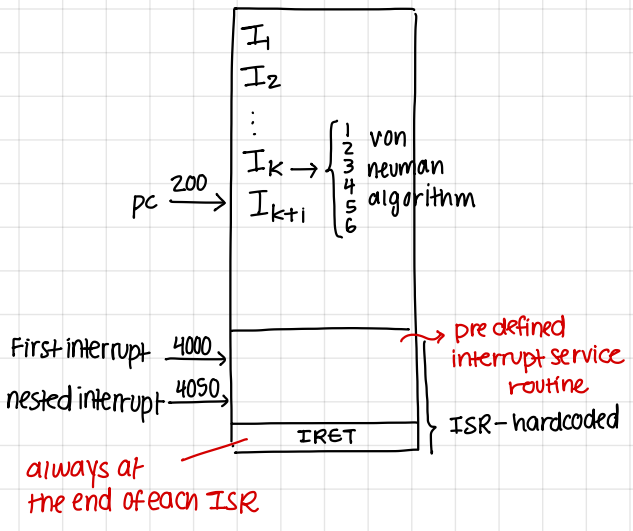
1. Silberschatz 2018
2. Stallings 2018
3. Tanenbaum 2015

یکشنبه ۷ آبان جلسه ۱+۲

لیست سوال جلسه

Interrupt

* جلسه اول - سه شنبه ۴ مهر *

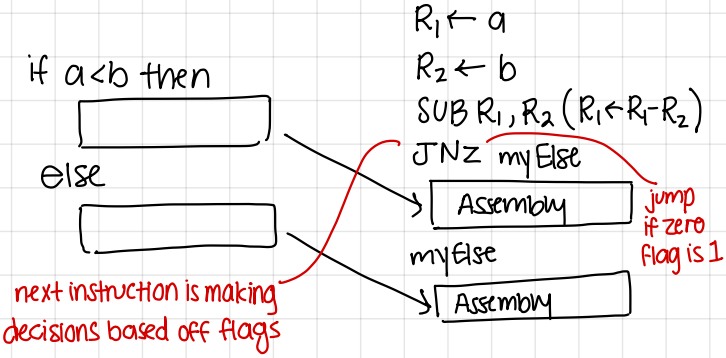


if we are in the middle of the von neumann algorithm, we reach the end first

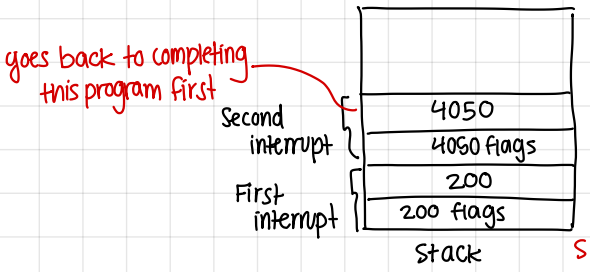
Nested Interrupts:

- when we interrupt a program, we store the pc and the current flags of that program and then move to the corresponding ISR
  - → the flags of I_k might be used in I_{k+1}

Converting high level languages to assembly:



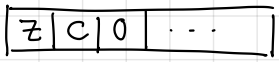
Nested Interrupt



- we use the stack to store the pc and flags

A single register used for storing all flags

word status



- stack also can store program data
  - → call by value - copy of the data stored in stack
  - → call by reference - address of data

slower, more pops

- the stack is very big and can almost be seen as infinite for the use of interrupts, so there won't be a problem for nested interrupts

IRET vs. RET

- IRET is used at the end of ISR's
  - → pops pc AND flags from stack (2 pops)

Don't need the flags of the previous program

- RET is at the end of any function to signify the program is complete
  - → pops pc from stack (1 pop)

while waiting for interrupt, we can't continue the same program, might have data dependency

→ we look for a different program to run

- otherwise we have "Busy waiting" which is not productive

can't check each time like in software, slows down processing

A single CPU runs multiple programs CONCURRENTLY, but appears it is working in parallel

Operating Systems { Resource Allocator ~ manages memory, hardware, TIME, etc.
controls programs

tries to maximize time to run the most programs

OS → A middleman or translator from the humans requests to the CPU

- intermediary between user and hardware

Car analogy: OS → driver, CPU → car } OS not NECESSARY, but makes using the car easier

most embedded systems don't have an OS

Writing operating systems for a CPU

→ what does the user want? multitasking? multiple cores available?

when we change the CPU, the OS needs to change as well

* جلسه دوم - یک شنبه ۹ مهر *

- Operating systems are interrupt DRIVEN
  - the OS program is divided into multiple interrupt service routines
  - completely made up of interrupt service routines

OS : a program divided into pieces in the Interrupt service routines → just like event driven software code
↳ the OS needs to communicate with the hardware
↳ important part of the OS is the BIOS program → boots the system
Bootstrap program
↳ loaded at power up
↳ Firmware
↳ sets pc ⇝ initializes all aspects of the system

Interrupts:
- Program → tells us warnings in the program
- Timer ⇝ sleep, wait, etc.
- I/O → reason why interrupt came to be
- Hardware failure

Interrupt is STARTED by the CPU (hardware) but ends with the IRET command determined by the OS (software)

I/O devices and CPU can run in parallel

each I/O device has certain ports defined for error and getting input
> In the motherboard the ports for I/O devices are hardwired and defined
ex.
printer port is always the same on all OS's written

printer port number 25H → defined by the printer itself

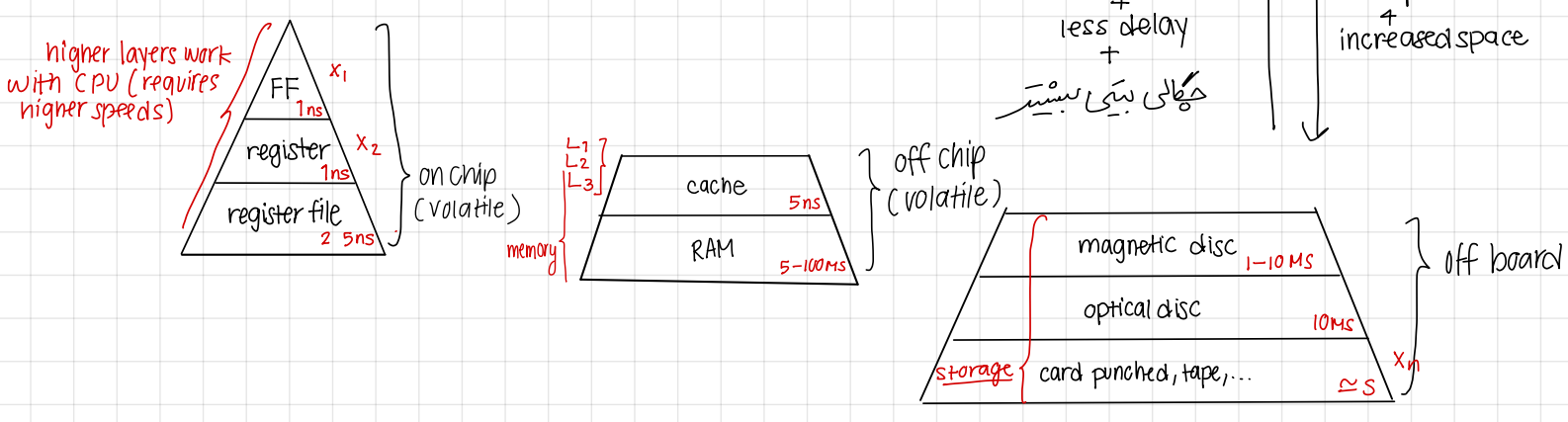
IN, AX5 → the register we will store data in (error) → tells me the status of the printer's job

- Interrupt table / vector → stores all the interrupts currently happening
  - exceptions = software interrupts
- if the next instruction is write or (not read). we can continue the same program
  - ↳ if the next instruction is read, no matter if theres data dependency or not, the program cannot continue
    - ⇝ can't check for dependency, slows down hardware
- nested interrupts - good to make sure the priority interrupts are executed first
- OS must follow these steps for each ISR:
  - save any program data we might need
  - process interrupt
  - restore process state information → Program Status Word
  - restore old PSW and PC (IRET)

  ↳ The ISR must be initialized before we start the interrupt

RST
↓
a function call with high priority that (most likely) won't be interrupted again

Storage definition and notation

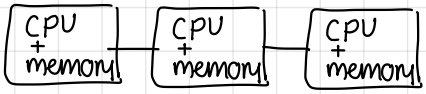


★ جلسه سوم - یکشنبه ۱۹ مهر ★

"parallel systems" → more throughput ⇝ more done in the same amount of time

Computer System Architecture

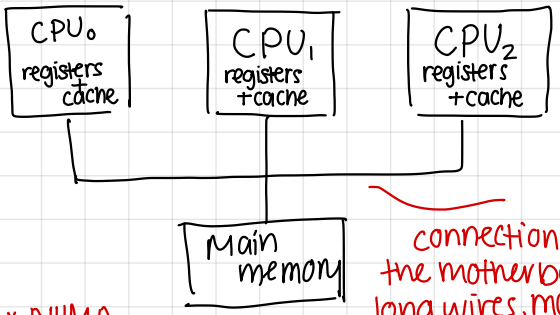
① multicomputer



Loosely coupled → 2 computers
vs.
Tightly coupled → 2 processors on a chip

② multiprocessor

→ multiple CPU's
→ harder to communicate ⇝ slower



connections are on the motherboard, long wires, more power consumption

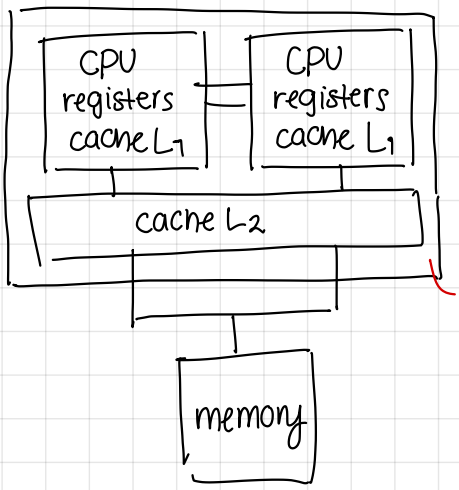
* NUMA have different access time for each CPU

Multiprocessors
- Asymmetric multiprocessing (boss worker, assigns each processor different tasks)
- Symmetric (all processors do the same task)

③ multicore

→ on chip
→ have a common level of cache
→ 1 chip, multiple processors
→ internal connection

shorter wires, less distance, faster, less power consumption



one chip

Operating System Structure

Single program computers leave the CPU idle a lot ⇝ not resourceful

① Multiprogramming
→ one job is focused, when it has to wait, switches to another job

② Time sharing
→ all programs are run together (feels interactive for the user)

user might abuse access if they have the pc:
→ you give the CPU control to a program and they don't return it
→ change/delete pre existing programs (even the OS)

can't fix these problems with software

OS has power while it has the pc

process vs. job = program = task

process
↓
a program loaded in the memory

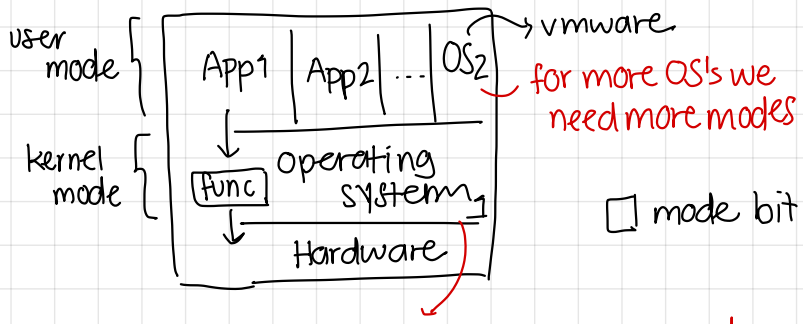
one program can have multiple processes

only OS can change it

Solution 1: OS always loads its program in a specific address
Solution 2: we get a flag, telling us if its an OS program
Solution 3: Dual-mode operation

* جلسه چهارم - سه‌شنبه ۱۸ مهر *

Dual mode operation — user mode / kernel mode

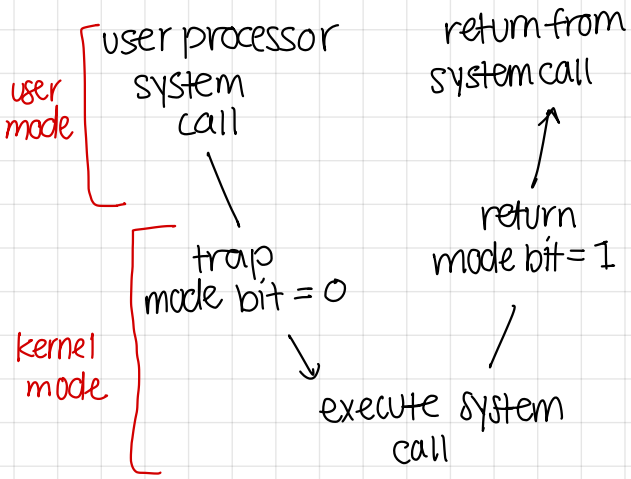


for more OS's we need more modes

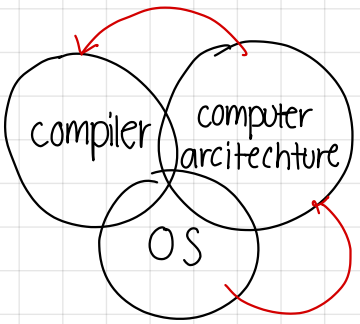
☐ mode bit

OS1 has more power/control + speed than OS2

OS2 is viewed as an application to OS1



in pipelining, when we want to run two consecutive instructions, there might be data dependency, so CA asks the compiler to fix this



goes from 1 to 0 by software
goes from 0 to 1 with hardware

uses "watchdog timer" to return pc

software can't make sure external programs don't do destructive things and returns the pc

asks CA to have ISA that can only be done by the OS

"priveteged instructions"

To prevent infinite loop / process hogging resources

↳ A timer that runs an ISR each time it ends

↳ only OS can control this timer ~ program could keep reseting the timer

process communication vs. virus

both programs tell the OS they are going to share n kb beforehand

hard to determine when and how much of a program to load

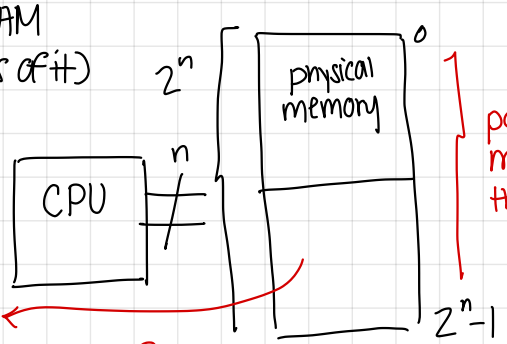
Memory management

- allocate programs/data when needed
  - ↳ new CPU's don't load the entire program in the RAM to start it (we might only be executing 10/100 ms of it)

Storage Management

↳ not the OS's main job but can cause efficiency and convenience

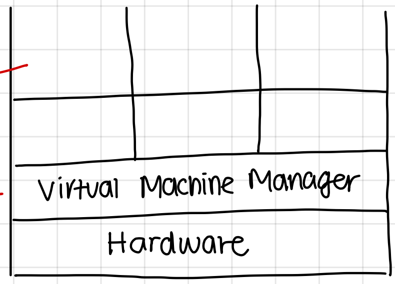
- organize directories, access controls,



جلسه پنجم - یک شنبه *

guest OS ←

hosting OS ←



~ any software that works with the hardware
↓
operating system

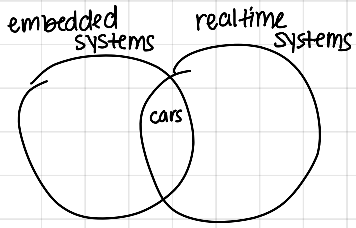
can windows be installed on a mac? No
(virtualization

if the guest OS can be run on the CPU directly, its virtualization
(if not → emulation

شرط زمانی

کوش محدود زمانی داریم

Real time vs. Unreal time



embedded system with no OS → bare metal

os-book.com ~ silberschatz

OS Structure

- We want OS to provide us with services and user interface
  - services: program execution, error detection + security, resource allocation, I/O operations, communication
  - user interface: command line, GUI
  - ↳ software architecture provides components and their interconnects

- OS interface command interpreters can be kernel based or use a system program
  - ↳ kernel-based
    - \+ faster
    - \+ not a seperate program
    - \+ user can't interfere
    - − if there is a software bug the entire OS crashes
    - − can't be updated
  - ↳ system program
    - \+ the entire OS doesn't crash
    - \+ easily upgradable
    - − can be manipulated / altered — entirely deleted
    - − slower

Structure vs. Architecture
- decomposing parts of the OS
- also speaking on the quality of said parts

- communication in the OS
  - message passing (safer), like emails
  - shared memory — race condition (faster, simpler); an entire block, can't share small amounts
  - ↳ the OS doesn't allow two programs to access the same address space — dangerous (data loss)

- System calls
  - ↳ programs the OS runs to provide services
  - ↳ written in C, C++, Assembly or — usually written in C, doesn't need to be altered for different ISA's like assembly (we have C compilers
  - ↳ common system calls
    - > process control
    - > file manipulation
    - > device manipulation
    - > Information maintenance
    - > communications
    - > Protections

جلسه ششم - سه‌شنبه *

- Structure 1 : Monolithic
  - ↳ single large program in kernel mode
    - if a module crashes, the entire OS crashes

- Structure 2 : layers
  - ↳ inner layers don't crash when outer layers crash
  - ↳ layers communicate -slow

* sometimes non-consecutive layers communicate
  - ↳ breaks layer structure

- Structure 3 - Microkernel
  - ↳ no more layers → one core that can access all external OS source code
  - ↳ each program/service run seperately ⌣ if program crashes, kernel doesn't

- Structure 4 - Modules
  - ↳ modules use interfaces to communicate -quicker than message passing -more direct (API)
  - ↳ different programs loaded in kernel

- Structure 5 - Hybrid
  - ↳ reliability - less crashing
  - ↳ security - safe message passing
  - ↳ usability - easy updating and using old source code

- Debugging the kernel
  - ↳ use logs to see where errors occur
  - ↳ hard because we can't 'test' → failures are expensive.

Processes

A program loaded in the RAM to be executed

program → passive
process → active

- A program progresses sequentially — why java is slow
- The code we write is a program, JVM is the process using the program

process: generated → ... → terminated
(generated: new() ; ...: wait() ; terminated: kill())
can use tag to pause → define a state ex. waiting for I/O

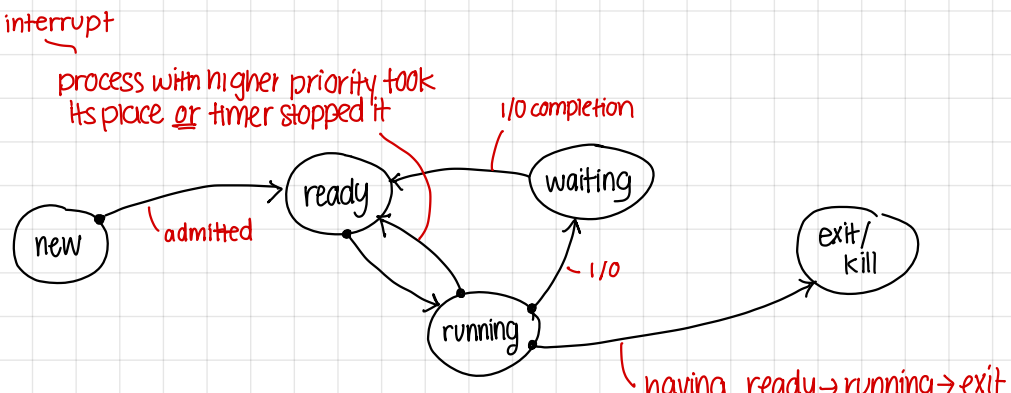
- Whats happening when CPU is not being used:
  - ↳ clock progresses but Von Neumann algorithm doesn't continue
- HLT (halt)
  - ↳ sends CPU into idle state
  - ↳ exits idle and continues with either RST or interrupt

- process components: source code, function call stack, code data (heap+stack)
- context switch: changing our pc, flags, and data to switch a process
- Process State
  - ↳ new
    - > allocate program space
    - > generate PCB
  - ↳ running
  - ↳ waiting
  - ↳ ready: A program that is ready to be executed but the CPU is occupied
    - > has a queue/list because we have more processes than processors
  - ↳ terminated
    - > deallocate memory

Alot of programs are I/O bound (games)
↳ DBMS

① Loader (loads program in mm (DMA))
② Assign PC
③ Start execution
④ System calls / I/O device

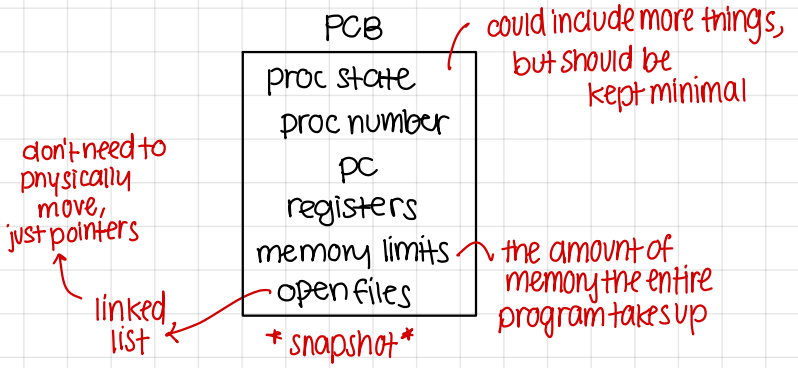
↳ we can use any algorithm to pick the next process (OS)
↳ FIFO, LIFO, random, etc



having ready → running → exit usually doesn't occur
↳ All CPU, no I/O

- **Process Control Block**
  - ↳ A data structure used to store process information in the OS
  - ↳ stored for multiple processes
    - > OS memory overhead
  - ↳ All important information required for context switch
    - * CPU scheduling info — scheduling queue pointers
    - * Memory management info — the memory allocated to the process
    - * Accounting info — CPU used + time limits + time since started
    - * I/O status info — I/O devices and files allocated to the process

important to not allow overlap between processes + process data is stored in the right place



  - ↳ we continuously save state and reload the PCB for each process when we context switch

- **Process scheduling**
  - ↳ The process scheduler selects from the available processes for the next execution on CPU — to go from ready to running
  - ↳ scheduling algorithm requires priorities and scheduling queue pointers
  - ↳ Each I/O device has its own queue in the OS

A process can go from the CPU ready queue to an I/O device ready queue, and back to the CPU ready queue

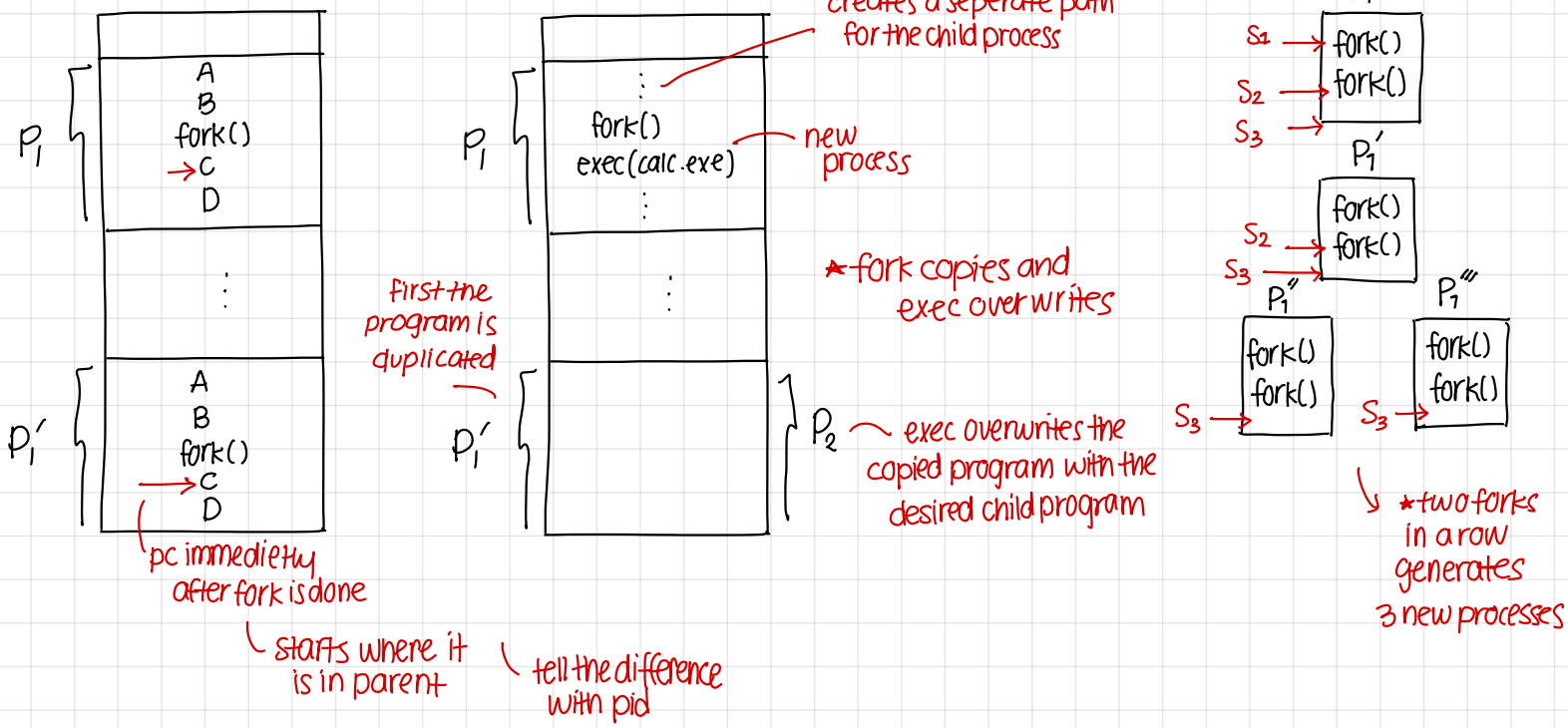
- **Schedulers** — An operating system MUST have this scheduler
  1. short term scheduler (CPU) : selects which process to be executed next and allocates CPU
     - > event based
     - > empties queue and assigns processes to the CPU
  2. long term scheduler (Job) — selects which processes to be in the ready queue
     - > not event based → periodically looks at the queue and removes CPU intensive processes
     - > temporarily lessens load of ready queue to speed up other processes ~only the larger program is delayed
  3. medium term scheduler
     - > taking things out of the ready queue
     - > remove the process from the memory, store on disk, and bring back from disk for execution
     - > swapping — taking out and bringing back a partially executed program

- processes can be I/O bound or CPU bound (intensive) — long term scheduler strives for a good mix of the two
  - ↳ I/O bound : gaming → requires high refresh rates
  - ↳ CPU bound : computations → consistent use of ALU

- **Context Switch** — doesn't count as CPU utilization
  - ↳ when switching between processes we use the PCB's
  - ↳ context switching time is the OS overhead
    - > users program does not progress during this time → must be quick
  - ↳ some hardware have multiple sets of registers → makes context switch quicker

why use stack for interrupts when we have the PCB? quicker, PC can be used to see which process to continue

- Process creation
  - ↳ two types { Manual → user starts a process
    Programmed → another process starts a process
  - ↳ In programmed process creation, we have a Parent and Child process — causes the generation of the child process
  - ↳ Resource sharing options
    1. parent and child share all resources
    2. children share subset of parent resources
    3. parent and child share no resources
  - ↳ sometimes the parent and child have nothing to do with each other — usually parent wants something from child – otherwise there is likely a problem
  - ↳ fork : creates a duplicate of the current parent with pc at the next line

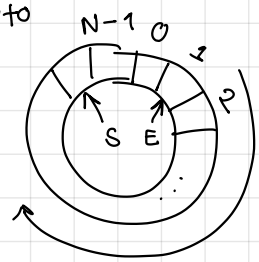


- The parent should (logically) wait for the child
  - ↳ otherwise an extra process was created that no one is waiting for
  - blocking — wait ⇝ waits for children — can also catch the status returned
    exit(n) ⇝ returns n to parent
  - ↳ the parent creates the child and can kill it
- A process deallocates resources either with exit(n) or return() — in main
- abort ⇝ kills child processes
  - ↳ might do this if → child exceeds allocated resources
    - ↘ task assigned is no longer required ~ ex. another process found the solution
    - ↘ the parent is exiting (some OS's do not allow a child to continue if its parent terminates)

- zombie process – the parent doesn't wait for the child process
- orphan process – the parent doesn't wait and the parent process has finished executing — more dangerous – can't even abort no one is watching the child

★ A browser is a multiprocess program
  - ↳ each tab is a child

- A process can be independent or cooperating
- we need ways to communicate between two processes - cooperating processes فرآیندهای همکار
  - ↳ shared memory - part of the memory is used by both processes ← inconsistent ناسازگار
  - ↳ message passing - a buffer is allocated in the RAM by the kernel and processes add and take from this buffer ← handled by OS
    - > might also need to define what process they are sharing to
    - > circular buffer (queue)

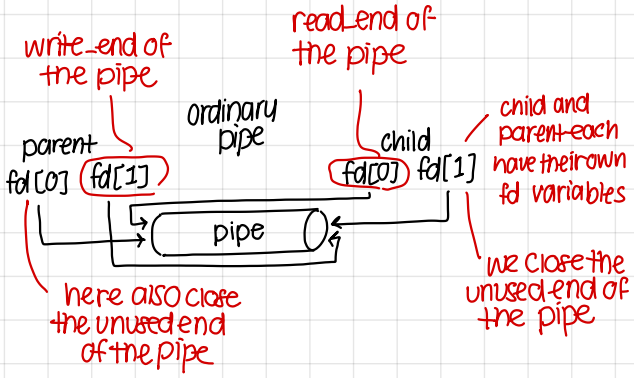


(i=i++) % N ← for new index
S → start = out
E → end = in
(e++) % N

    - > can have two types:
      - ① Direct (unidirectional)
        - ↳ one process sends while the other recieves
      - ② Indirect (uni and bidirectional)
        - ↳ each message is directed (has a recieptent) ← mailbox system
        - ↳ can be used between multiple processes

- communication can be — blocking / synchronous ← reciever waits until it recieves a message
  \ non blocking / Asynchronous ← reciever recieves either the message or a NULL message

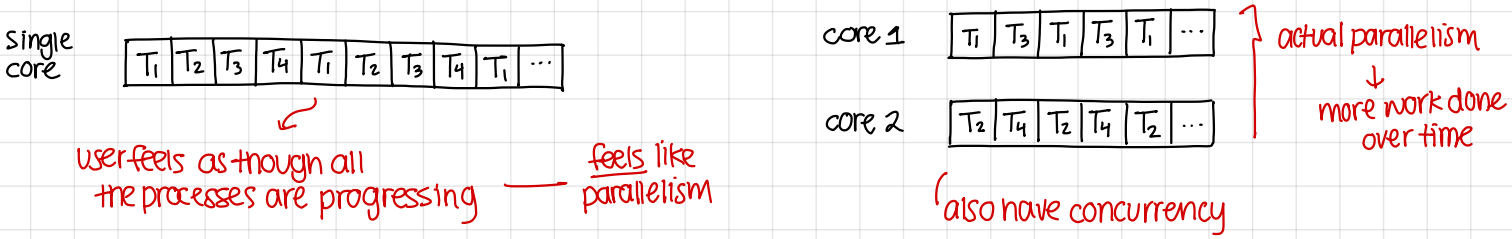
- Server Client communication
  - ↳ sockets - endpoints for communication
  - ↳ RPC - can call methods in another server ← someones always listening for requests
  - ↳ Pipes - for communication between two processes
    - > can be uni or bidirectional
    - > Ordinary pipe - used for communication between parent and child processes ← uni directional
    - > Named pipe - used by any two processes ← bidirectional



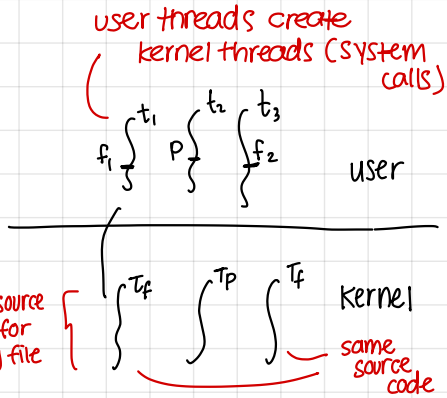
Threads

- A lighter process → if they both come from the same source, we don't load the program twice + less data in PCB → (processes are resource intensive) (today's users require more multiprogramming)
  - ↳ seperate (things unique to a thread): Thread ID, PC, register set, stack
  - ↳ shared (two PC's in one source code): code section, data section, OS resources
  - (context switch takes less time)
- Threads used for server client application – each client requires a responsive and lightweight process → thread
- Thread advantages
  1. Responsiveness – allows us to see progress frequently
  2. Resource sharing – memory and resource sharing between threads
  3. Economy – fast → don't need to buy stronger CPU's for multiprogramming
  4. Scalability – we can have more threads than processes
- multicore / multiprocessor challenges (the programmer has to deal with these issues)
  - Dividing activities – our job must be dividable into smaller parts
  - Balance – must be divided into a balanced way to actually be beneficial
  - Data splitting – data and task must be independent to seperate them
  - Data dependency – race condition
  - Testing and Debugging – could lead to inconsistent answers

Concurrency همروندی vs. Parallelism توازی / موازی سازی



- Amdahl's law: $speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$ (overhead of the OS; series → not parallel; assuming we have N processors)
- Parallelism
  - Task parallelism – doing multiple different tasks on one data
  - Data parallelism – doing the same task on a lot of different data



- Hardware threads
  - ↳ have register sets for multiple threads ~ can load multiple threads while others run
  - ↳ switching threads is easier
- user and kernel threads
  - Many-to-One ↛ same kernel thread for many user threads (one thread blocks others)
  - One-to-one ↛ each user thread gets a kernel thread (has large overhead)
  - Many-to-Many ↛ combination of the two (the OS itself creates a sufficient amount of threads; Two-level: can also bind a user thread to a kernel thread)

(two choices)

1. create a new thread for each user thread system call
2. for threads that want to make the same call, we wait for the first thread to finish

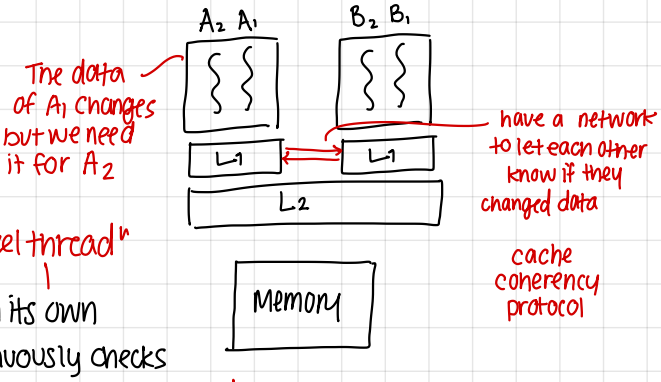
- Thread states / operations
  1. Spawn – similar to process "new" → when a process is spawned, a thread is also spawned for it
  2. Block – when a thread needs to wait for an event
  3. Unblock – when event occurs and thread moves to ready
  4. Finish – thread completes and deallocates

Implicit threading (simpler for the programmer) {
- Threadpool → take a thread from pool whenever we have a thread
- Grand Central Dispatch → have a sign to say "run this as a thread"
- OpenMP → use # to talk to compiler

good for when we don't have control over thread creation process

- Thread local storage – allows each thread to have its own copies of data
  - ↳ static data unique to each thread
  - ↳ visible across function invocations of a thread

- cache conerency protocol
  - ↳ for communication between threads/processes on seperate cores
  - ↳ used when threads/processes are cooperative



- Thread termination
  - Async cancellation – each ends on its own
  - Deferred cancellation – thread continuously checks if it can terminate itself ↝ checking if it was deferred

"cancel thread"

Windows threads	vs.	Linux threads
• one-to-one mapping • seperate user and kernel stacks • private data storage area		• thread creation done with clone() ↳ creates a copy and we set permissions for the child process with flags

- When to use: امثال تداخل بيشتره
  - ↳ multithread
    - > when we have shared data
  - ↳ multiprocess
    - > when we don't want everything else to crash if one crashes

Process Synchronization

هم‌گام‌سازی

- Cooperating - both threads and processes require synchronization (occurs when we have read + write)
- concurrent running could lead to data inconsistency
- Synchronize processes → orderly execution where there is a shared logical address space

- race condition : when shared access to data leads to inconsistent results (solved with serial execution)
  - ↳ if more than one process is accessing and manipulating the same data concurrently

ناصر بدادی

- Critical section : A part of the code that if run concurrently, leads to race condition → needs to be run serial (changing common variables, updating table, writing file, etc.)
  - ↳ the OS can't recognize these, programmer must tell the OS where it is
  - ↳ only one process can be in the critical section at once → has entry and exit section

our critical section must always be minimum

```
do {
    ENTRY SECTION
        critical section
    EXIT SECTION
        remainder of code
} while (true)
```

- Requirements for critical section solutions
  1. Mutual Exclusion – only one process in a critical section at a given moment (انحصار متقابل)
  2. Progress – At least one process is progressing ~ some decision is eventually made
  3. Bounded waiting – we garuantee that a process waiting will eventually be let in (انتظار محدود)
     - ↳ theres a limit to the number of other processes let in before a certain process is let in

- Preemption – temporarily interrupting a task being carried out, with the intention of resuming later
  - ↳ When we have preemption in the OS race conditions can occur frequently ↝ responsive
  - ↳ In non preemptive OS's, since we dont context switch in the middle of code → no race condition ↝ real time programming (we want our response immediety)

- Critical section solutions

1. Peterson's solution
   - ↳ software solution
   - ↳ correctness depends on system hardware → must have Atomic instructions (a algorithm must have corresponding atomic instructions in assembly)

n size array for n processes

```
P0
do {
    flag[0] = true;
    turn = 1;
    while (flag[1] && turn == 1);
        critical section
    flag[0] = false;
        remainder
} while (true);
```

چرا turn ضروریه؟

once its done, the second condition of the while in j

```
P1
do {
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn == 0);
        critical section
    flag[1] = false;
        remainder
} while (true);
```

causes race condition but not an issue in this case

اگر نباشه atomic مشکل تداخل and پیش میاد؟

* if you prove an algorithm works for 2 processes, you can prove it for n processes

Ex. Is this algorithm a critical section solution?

```
P0
do {
    r = rand()
    while (1 ≤ r ≤ 0.5);
    critical section
} while (true);
```

```
P1
do {
    r = rand();
    while (1 > r > 0.5);
    critical section
} while (true);
```

Has mutual exclusion ✓
Has progress X
No bounded waiting X

(non-interruptable)

② Hardware solutions – adding atomic instructions that are run in one instruction that "locks"
- ↳ when you disable interrupts in uniprocessors, critical section problem is solved
- ↳ we provide atomic hardware instructions for multiprocessors
  - ↳ can't disable interrupts, wasting CPU

can be solved by adding logic to code to prevent it

- Test_and_Set() : chooses an address space for some lock, checks and returns the value of the lock

all run as one instruction

```
bool test_and_set(bool *target){
    boolean rv = *target;
    *target = true;
    return rv;
}
```

if return is true then the lock is activated and we cannot enter the critical section

```
do{
    while(test_and_set(&lock));
    [critical section]
    lock=false;
}while(true);
```

bounded waiting problem

one process keeps taking access of the lock

- Compare_and_Swap()

to solve bounded waiting issue in TAS

all run as one instruction

```
int compare_and_test(int *value, int expected, int new_value){
    int temp = *value;
    if (*value == expected){
        *value = new_value;
    return temp;
}
```

look at the lock, if its a certain value (unlocked) otherwise its locked

```
do{
    while(compare_and_swap(&lock,0,1)!=0);
    [critical section]
    lock=0;
}while(true);
```

bounded waiting problem

previous ones are complicated and inaccessible to application

process keeps spinning around itself

spinlock

③ Mutex lock (OS solution)
- All above solutions (including mutex) involve busy waiting
- This OS solution uses the hardware locks to implement acquire() and release()

use the locks to run these atomically

```
acquire(){
    while(!available);
    available=false;
}
```

busy waiting

```
release(){
    available=true;
}
```

no difference with HW solution, just cleaner

still doesn't solve bounded waiting and busy waiting

```
do{
    acquire lock
    [critical section]
    release lock
}while(true);
```

④ Semaphore
- solves busy waiting with a more sophisticated OS solution

negative value means we can't

* wait() and signal() run atomically
  - ↳ otherwise race condition

```
typedef struct{
    int value;
    struct process *list;
}semaphore;
```

list of processes waiting to enter critical section

```
wait(semaphore *S){
    S->value--;
    if(S->value<0){
        add process to S->list;
        block();
    }
}
```

no longer busy waiting

leave the ready queue

we dont context switch to them

```
signal(semaphore *S){
    S->value++;
    if(S->value<=0){
        remove process from S->list;
        wakeup(P);
    }
}
```

interrupt

wakeup another process to complete the critical section

- semaphores can be
  - binary (no list, same as having mutex)
  - counting semaphore (multiple resources)

could use unique case of this for having limited access to a resource
(ex. only two writers)

- can be used for synchronization problems → we want two processes to run in a certain order

```
P1:
    S1;
    signal(synch);
```

```
P2:
    wait(synch);
    S2;
```

to use for 3 or more processes, add another synch semaphore

- Problems with the provided solutions
  - → Deadlock – none of the processes can continue
  - → Starvation – if our semaphore queue is LIFO, some processes will never be run (بن بست / قحطی)
  - → Priority inversion – A process with a higher priority is waiting for a lower priority process to release the lock

P_0	P_1
wait(S); ← gets stuck	wait(Q);
wait(Q);	wait(S); →
⋮	⋮
signal(S);	signal(Q);
signal(Q);	signal(S);

DEADLOCK!

$L(R) < M < H(R)$ — A higher priority process is now waiting for the lock

low priority process holds lock

we context switch to an unrelated higher priority process

> priority-inheritance protocol – we give a high priority to a process in its critical section

- Classic synchronization problems
  - → bounded-buffer problem ✓
    - > use wait for adding to the buffer (empty/write) and removing (full/mutex)
  - → readers-writers problem ✓
    - > use wait and signal so there is only one writer at a time
    - > also have a reader_mutex for changing the read mutex → also lock write_mutex so they don't write while we read
  - → dining-philosophers problem ✗
    - > if all philosophers pick up their left chopstick at once, they all cant pick up a right chopstick → DEADLOCK

→ semaphore bad usages :

signal(mutex);	wait (mutex);	wait (mutex);	
critical section	critical section	critical section	critical section
wait(mutex);	wait(mutex);		wait(mutex);
no mutual exclusion	no progress	no progress	no mutual exclusion
STARVATION	DEADLOCK	DEADLOCK	STARVATION

⑤ Monitor

- solves the problem of only running part of code
- condition is like creating n semaphores
- only one process may be active within the monitor at a time
  - ↳ has a queue of processes waiting to enter a monitor, and have seperate queues for each condition

→ dining_philosophers with monitor

> starvation → left and right keep passing the middle chopstick — solution : randomize the order of picking up the left or right chopstick

- Alternative approaches
  - OpenMP → use # to talk to compiler
  - Transactional memory → atomic memory changes
  - Functional programming languages

CPU scheduling

usable for any real life queue's!

sometimes go against each other

- Scheduling can help work towards: Maximize CPU utilization vs. Better response time for user
- Scheduling helps manage different sequences of CPU and I/O bursts
  - ↳ CPU/IO burst: process execution consists of multiple consecutive CPU execution or I/O wait
    ex. training neural networks
  - ↳ 2 millisecond CPU bursts occur frequently

process/thread/CPU scheduling

- short-term scheduler: whenever CPU is idle, a process must be chosen from the ready queue (consists of PCB's)
- Schedulers — preemptive – forcefully context switching (could cause race condition; time slice expiration)
  - non preemptive – when switching happens during:
    - running → waiting (I/O request)
    - running → ready (interrupt)
    - waiting → ready (completion of I/O)
    - process termination

  (switching processes here does not count as preemption)

  اعضای گیتی
- Dispatcher – responsible for context switching
  - ↳ the short-term scheduler gives the chosen process to the dispatcher
  - ↳ should be fast
    - > dispatch latency: time it takes to stop one process and start another

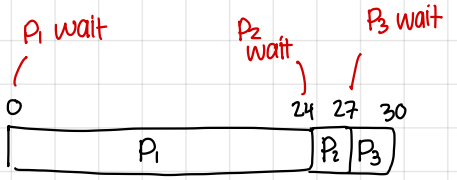
- Criteria for defining a suitable scheduler
  - بهره‌وری ↳ utilization (only the users programs are considered (not OS overhead))
  - برون‌ده ↳ throughput — do the shorter jobs
  - ↳ Turn around time — time it takes for a process to complete (all waiting + running in CPU)
  - ↳ Waiting time — only considering time waiting in ready queue
  - ↳ Response time — زمان پاسخ‌گویی

OS { synchronization, scheduling (deadlock!), memory management }

- Scheduling Algorithms

① First come, First Served (FCFS)

Process	Burst time
P_1	24
P_2	3
P_3	3

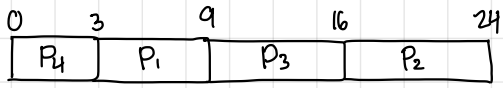


Avg waiting time: $\frac{0+24+27}{3} = 17$

- > simplest
- > long average waiting time
- > Convoy effect – short processes held behind a long process
  ex. multiple small I/O tasks held behind one large CPU bound task

② Shortest Job First

Process	Burst time
P_1	6
P_2	8
P_3	7
P_4	3



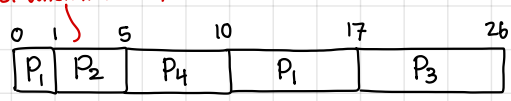
Avg waiting time: $\frac{0+3+9+16}{4} = 7$

- > chooses based on length of next CPU burst
- > minimum average waiting time
- – can't always guess next CPU burst
- – can cause starvation

*Shortest remaining time first (preemptive)

Process	Arrival	Burst
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

had shorter burst when it arrived



Avg waiting time: $\frac{(10-1)+(17-2)+(5-3)}{4} = 6.5$ (P_1, P_3, P_4)

- > chooses based on remaining time of CPU burst
  - ↳ both preempted processes and newly arrived processes
  - ↳ check queue again once a new process arrives

- Predicting the length of the next CPU burst

t_n = length of n^{th} burst

τ_{n+1} = predicted length of $n+1^{st}$ burst

α = how sure we are $0 \le \alpha \le 1$ (usually set to $1/2$)

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

③ Priority

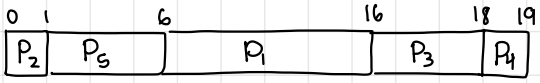
here our "priority" was based on finishing time

> general case of SJF
> here we associate a priority to each process
↳ internal
- time + memory limits
- open files + IO/CPU bursts
↳ external
- importance of process
- funds being paid

can cause starvation → must include aging

Process	Burst	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

avg waiting time = 8.2



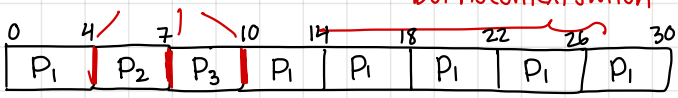
④ Round-Robin

> CPU allocated to a process for a limited amount of time (time slice = q)
↳ too long – FCFS
↳ too short – context switch overhead
> 80% of CPU bursts should be shorter than q ~usually 10-100 ms

Process	Burst
P_1	24
P_2	3
P_3	3

context switch

algorithm runs, but no context switch



⑤ Multilevel Queue

> seperating ready queue → foreground (RR) – interactive processes
↳ each its own scheduling → background (FCFS) – batch processes
> queues seperated by priority

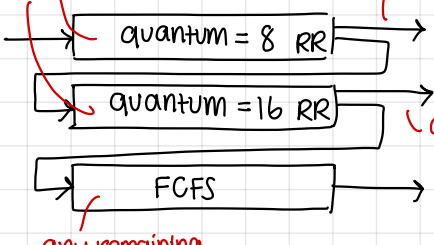
⑥ Multilevel Feedback Queue

can implement aging

> multiple queues but a process can move between various queues
> identifying factors:
↳ number of queues
↳ each queues algorithm
↳ determining when to upgrade/demote process

higher priority

if the process completes after 8 ms



completed in 16 ms

any remaining longer processes

◦ Multiple-processor scheduling — AMP / SMP

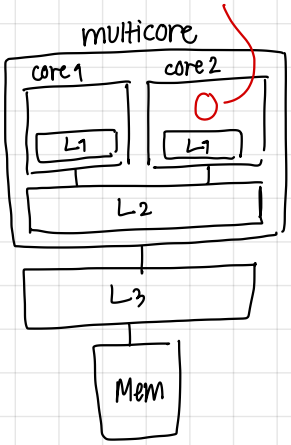
Affinity
load balancing
private or shared queue

↳ Asymmetric Multiple Processor
> Master processor – accesses data structures, PCB, etc. → no more need for data sharing

to swap cores we'd have a lower hit rate

↳ Symmetric Multiple Processor
> common vs. private ready queue — a single processor might have a long queue
> Load balancing: every processor participates in getting work done
↳ prevents aging of a single processor
↳ utilizes all of our CPU's

we are fetching the programs instructions, data, etc.



↳ Processor Affinity: the cache of a CPU is already filled with the processes data so we prefer to run the same process on the same core (private queue)
> Hard Affinity – ALWAYS on the same processor
> Soft Affinity – breaking affinity and the process runs somewhere else
* migration goes against affinity → use threshold for imbalance
↳ push migration – task periodically pushes to a CPU
↳ pull migration – idle CPU pulls a waiting task

Affinity > Load balancing
Affinity < Load balancing

↳ NUMA – Each core does not have equal access to memory ~ some have quicker access time
↳ Coarse grained – context switching to run different processes between instructions ~ instruction w/ long
Fine grained – using CPU for another process mid-von Neumann algorithm ~ cycle w/ long

second process run in between when we aren't using CPU

CPU	M	CPU	M	CPU	M
IF	ID	RO	EXE	WR	
~M	~CPU	~M	~CPU	~M	

Creates HW threads
4 cores → 8 threads

- Real time scheduling (زمان بحرانی)
  - ↳ processors that must respond quickly to events (بیمارستان، رستوران، ماشین، هواپیما...)
    - SW – timer
    - HW – external interrupts
  - ↳ Hard real time : task MUST be run by deadline — only importance → DEADLINE, other criteria doesn't matter
  - ↳ Soft real time : gives us a deadline, not guaranteed to complete by then
  - ↳ Event Latency : time from when an event occurs to when it is done running

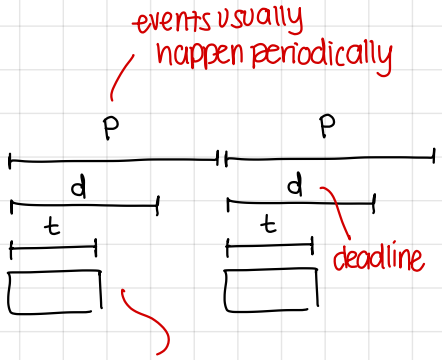
Event latency = interrupt latency + Dispatch latency

conflict: preemption of any kernel process + release of resources by low priority processes needed by high priority processes

interrupt latency (from arrival of interrupt to start of ISR):
+ checking for interrupt (HW)
+ finds ISR (HW)
+ copies data (HW)
+ context switch (SW) — OS checks which registers, where, etc.

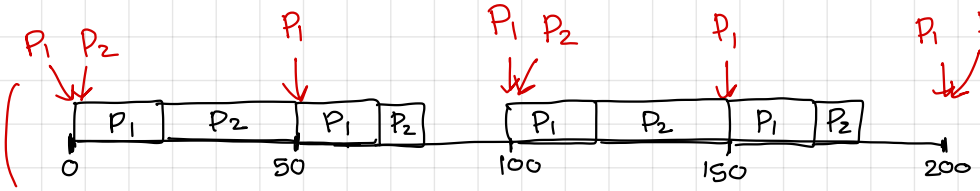
Dispatch latency: taking a current process off and placing another on the CPU

- Priority based scheduling
  - ↳ real time scheduling revolves around preemption and priority based scheduling
  - ↳ periodic processes – occur again periodically — period must be larger than deadlines
  - ↳ still can't guarantee Hard real time



1. Rate monotonic scheduling (1/p frequency)
   - ↳ the priority is determined by the rate of the periodic task
     - > shorter period = higher priority
   - ↳ problem: doesn't care about the deadline + CPU utilization is bounded (a max)
   - ↳ CPU utilization : $U_{max}(N) = N(2^{1/N} - 1)$ → if $U_{sys} < U_{max}$, RM is feasible — otherwise must check

copy of that task occurs again (for one individual task)



larger rate/priority

Process	Period	Burst	Utilization
P_1	50	20	20/50
P_2	100	35	35/100
			= 75%

↳ Hyper loop : if our scheduling is correct in our hyperperiod it'll work for the rest (ک.م.م پریودها)

Hardware scheduler? Hardware synchronization scheduler?

2. Earliest Deadline First scheduling
   - ↳ the earlier the deadline, the higher the priority
   - ↳ usually better utilization

3. Proportional share scheduling
   - ↳ CPU broken down into shares → T shares used amongst processes — guarantees each process recieves N/T of the total processor time
   - ↳ Each application recieves N shares where N < T

- Algorithm evaluation
  - Deterministic modeling → keep running to see different results
  - Queuing models → mathematical modeling
  - simulation → programmed model of algorithm

input frequency, wait time: $n = \lambda \times W$ (n: number of people in queue)

- Implementation → launching new schedulers has high risk + cost (could be bad) — time, company credibility
  - ↳ environments vary (different HW, CPU's, etc)
    - > flexible schedulers can be modified per system
    - > API's to modify priorities

$\frac{1}{P-d}$ for priority

good algorithm?

Deadlock

- Deadlocks prevent sets of concurrent activities from happening
  - ↳ we don't know peoples intentions ~ could be harmful
  - ↳ must have an outer overview of the system to see and prevent deadlock → OS
  - ↳ identifying deadlock is not easy - how long should our time out be? ~ could just be a very long process taking up resources
- Deadlock is a rare occurance → programs must be run in a very specific order to happen
- Deadlock: { To wait for a resource which is acquired by another process that is waited for a resource of the requesting process (circular)
  Never finishing wait state
  Circular dependencies between processes }
- 4 necessary conditions for deadlock to occur: ~ without one, no deadlock
  1. Mutual Exclusion – only one process can use a resource at a time
  2. Hold and Wait - a process holding a resource is waiting to acquire another resource held by another process
  3. No preemption – a resource can only be released voluntarily by the process itself, after it is done using it
  4. Circular wait – There is a set of waiting processes where P_0 waits for P_1, P_i waits for P_{i+1}, and P_n waits for P_0

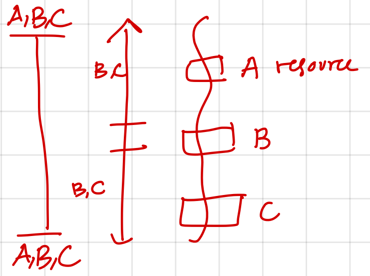
* recognizing deadlock in Resource-allocation graph → cycle occurance { one of each resource – deadlock
multiple instances of a resource – possible deadlock }

1. **Deadlock prevention** – trying to prevent deadlock by not having one of the 4 deadlock conditions in our OS:
   - removing mutual exclusion – would have to share all files → race condition X
   - removing hold and wait – would have to only start a process when we have all of its resources X (very strict + generally not possible)
     - > under utilization of the CPU and resources – many processes may require the same resources
     - > starvation – all resources of a process may not be available at once
     - removing concurrency
     - removing those benefits from the OS entirely
   - removing no preemption – would have to allow either: X
     - > self preemption: each process releases its own resource ~ don't know when to release
     - > destination process preemption: the process needing a resource releases the other processes using it – not safe
     - > save and switch resource status: not possible – large overhead
   - removing circular wait – requesting a resource in an increasing order of enumeration ✓
     can request to resource R_i, we can request R_j if $F(R_j) > F(R_i)$
     if $F(R_j) < F(R_i)$ release all resource types R_k where $F(R_k) \geq F(R_j)$
- Deadlock prevention leads to resource underutilization and reduced throughput

2. **Deadlock Avoidance**: the four conditions could happen, but we prevent the event/danger (large overhead for checking all running processes)
   - Resource allocation graph (single resource)
     - only add the resource if we remain in a safe state or no cycle after
   - Banker's Algorithm
     - ↳ safe state: there exists a sequence of all processes in the system

under utilization — هرکدوم از اون مسائل باعث میشد

Hold & Wait



30% ازمنابع استفاده

Throughput → سرعت تموم کردن کم میشه

چون resource رو نمیتونیم share کنیم

- Available resources → ✓
- resources allocated to processes → ✓
- Future request ??? ✗

Safe state – ترتیبی از اجرا فرآیندها که بتونند پشت سرهم اجرا بشوند

A → تا 12 نمونه

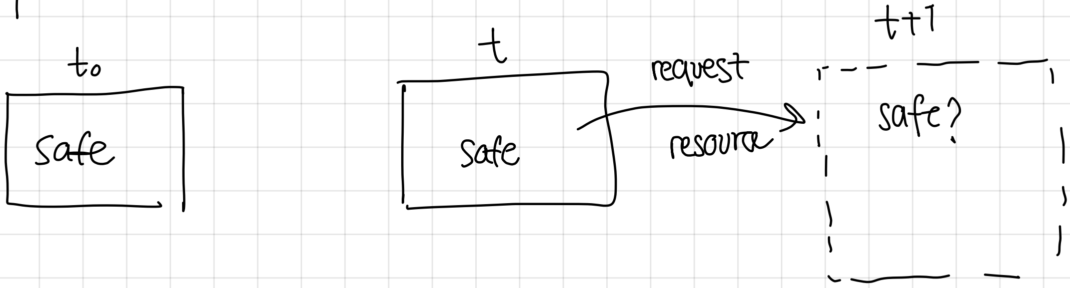
resource انقدار می خواد تا تموم شه ← max

	max	current	
P_0	10	5	5 تا در آینده نیاز داره
P_1	4	2	2
P_2	9	2	7

فرض می‌کنیم وضعیت جاری safe است بررسی می‌کنیم آیا پس از درخواست همچنان safe هستیم یا نه

بله: اجابت درخواست

خیر: جلوی درخواست گرفته می‌شود

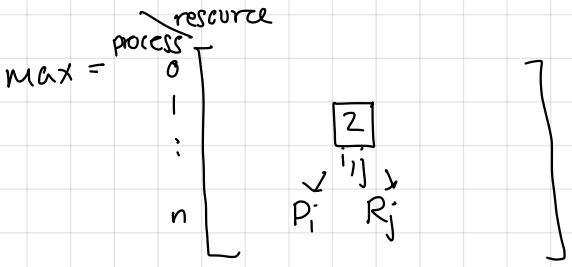


وجود دو رضایی موجب دو لاک میشه که فقط از هرکدوم یه نمونه داشته باشیم

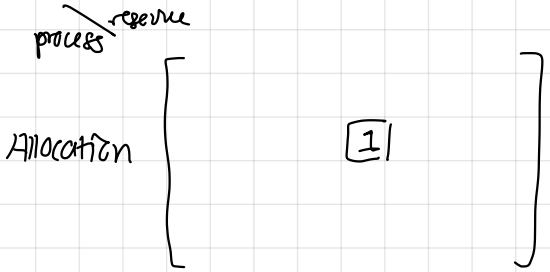
resources

A B C D

Availables = [4 5 3 2] در دسترس



process i needs at most 2 of resource j



process i has 2 of j's resources

Need = Max − Allocation

Bankers safety algorithm :

Loop all processes and look for processes that and resources

finish[i] = false && $Need_i \leq work$

if this was true we do

work = work + allocated

(running the process)

Bankers algorithm (resource-request)

is run each time a resource has a request

for each request we pretend to allocate the request, and check if its safe

Allocated += requested

Need −= requested

Available −= requested

Dec 10

اولی میشه — دومی نمیشه

if the request is not accepted, the process waits

- Deadlock detection and recovery

- Deadlock detection
  - ↳ by removing the resources from the RAG to create a corresponding wait-for graph
    - > we periodically run cycle detector ~ or could be run whenever we request a resource
      - the algorithm is time consuming for a large number of nodes

Request = max - need

Work → copy of Available
if nothing is allocated at the moment, finish = true ~ what if it needs resources later

* in safety algorithm, we didn't about the requested amount

32 → works
33 →

- Recovery
  - ↳ Abort all deadlocked processes
  - ↳ Abort one process at a time until deadlock cycle is eliminated ~ there might be other cycles
    - > which should we abort? — priority
      - how long its been computing
      - the one whos used the most / least resources
      - resources it needs to complete ~ more or less both argueable
      - number of processes that will be terminated
      - is process interactive or batch ~ large calculations ~ have lots of resources + wasted time

in a singular cycle one is enough

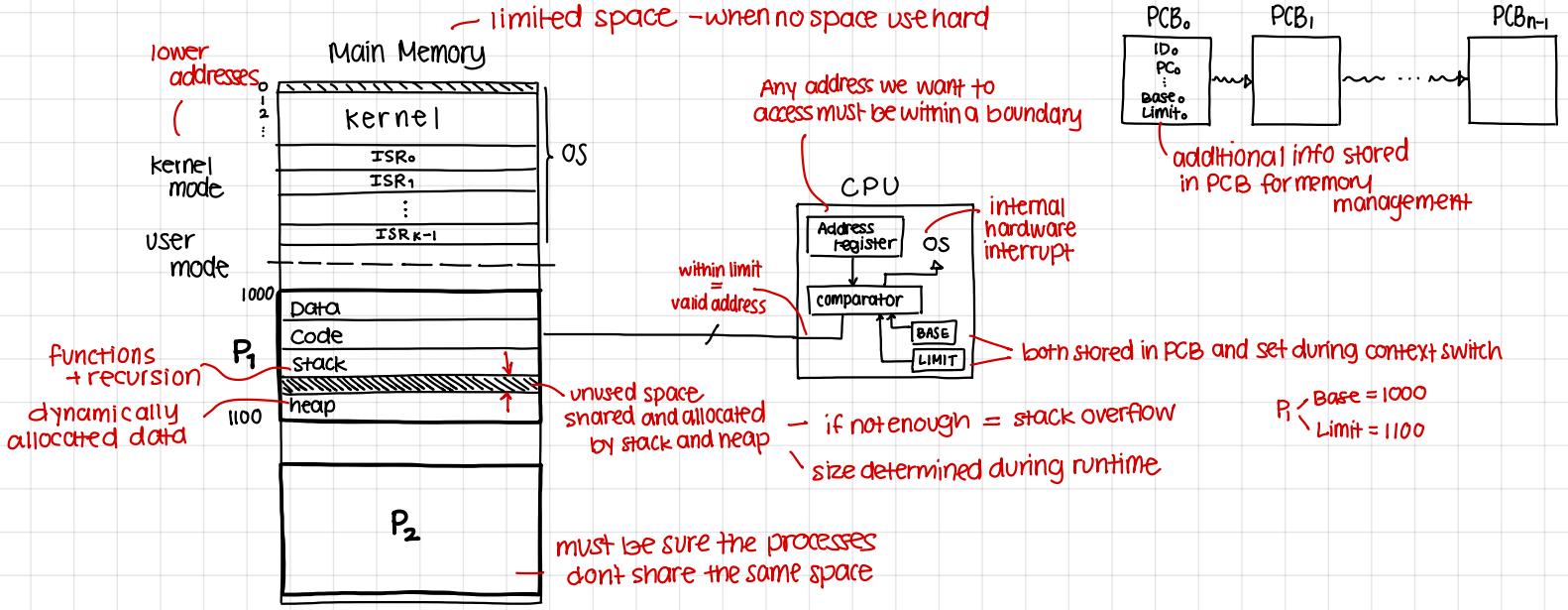
a process could be a common node in two or more cycles

the user will be dissatisfied but batch remains

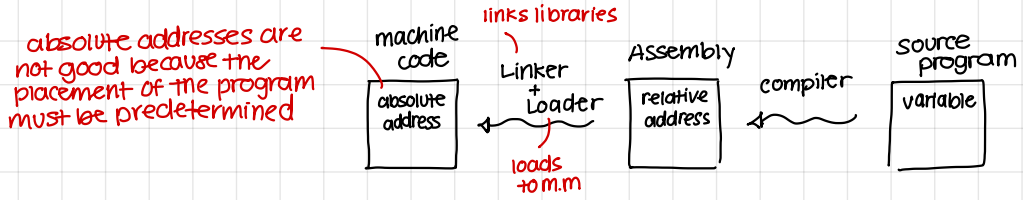
- Resource preemption
  - ↳ selecting a victim - minimize cost ~ least change / damage
  - ↳ Rollback - returning to a safe state that we can continue the process from ~ a valid state where the resource changes are re-doable
  - ↳ Starvation - might preempt resource from the same process repeatedly

Memory management

- Why memory management?
  - ↳ increase degree of multiprogramming — can fit more processes in m.m at once
  - ↳ increase speed/performance + CPU utilization + throughput — less time wasted waiting for resources (memory)
- Protection: protecting processes from accessing each other – preventing collision — unwanted access could be damaging
  - ↳ Hardware's job
    - > when we want to access an address space we are mid-Von Neumann algorithm → can't be solved with OS interference
    - > to check with software, we have performance penalty and latency
  - ↳ must keep each process' memory space seperate
    - > Legal address: the address space in which a process can safely operate in and access — base ≤ legal address < base+limit
      - base
      - limit

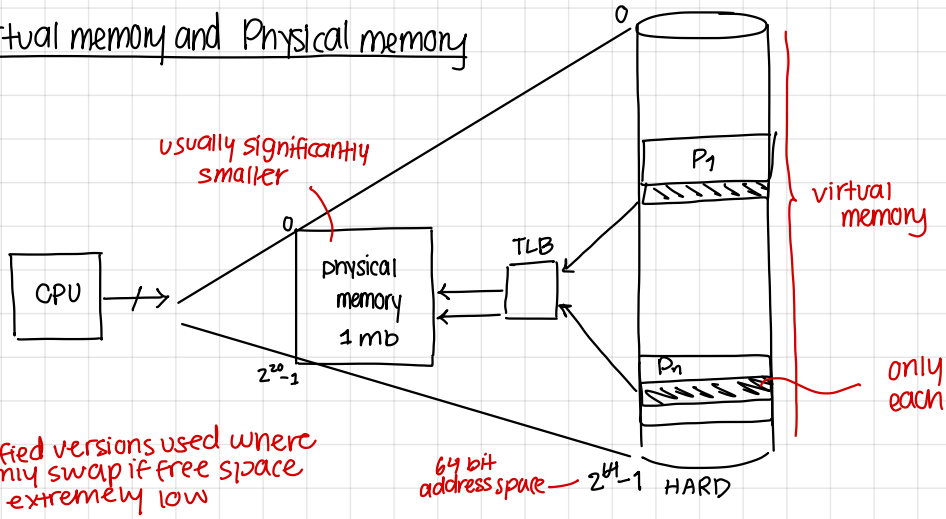


- Address binding
  - ↳ Input queue: Processes on disk waiting to be brought onto RAM — part of ready queue (ready queue also has processes in the main memory waiting for context switch)



  - ↳ Addresses can be determined at different times:
    1. Compile time – we know the memory location when compiling — otherwise generates relocatable code
    2. Load time – we know memory location when loading
    3. Execution time – absolute addresses are not determined until runtime — when our virtual and physical address spaces are not equal
  - ↳ MMU: responsible for mapping virtual memory to physical memory
- Dynamic loading: a part of a program is not loaded (to M.M) until it is called → better space utilization — loading main() and only loading what it needs to run step by step
- Dynamic linking: libraries are linked when running the program → better space utilization
  - ↳ opposed to static linking where libraries are added at compile time
  - ↳ stub: program that finds dynamic linked libraries that will be replaced with a linked subroutine

- Virtual memory and Physical memory



* we have a $2^{64}-1$ addressable space but a small limited RAM
(map V.M to M.M

only the necessary part of each program is in the RAM

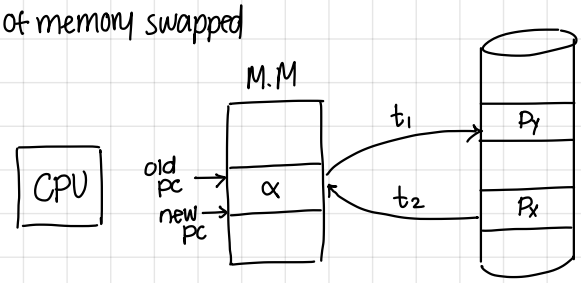
modified versions used where we only swap if free space is extremely low

- Swapping : swapping processes in and out of the physical memory to meet memory management needs
  - ↳ backing store : part of a fast disk used for our virtual memory
  - ↳ pending I/O issue : our DMA is using cycle stealing to transfer some data and the CPU cannot interfere → problem will occur if we swap in/out to a different location
    - > solution 1 : making sure we swap in back to the same location
    - > solution 2 : double buffering, creating a copy of the transfered data in the OS and copying that buffer everytime we swap out → VERY large overhead (space and time)
  - ↳ swapping cost : transfer time proportional to the amount of memory swapped

(the medium term scheduler was for scheduling needs (also didn't involve swapping in)

Memory Allocation (3 types)

- Criteria
  - → CPU utilization : there should be enough processes in the RAM to keep the CPU busy + minimal overhead



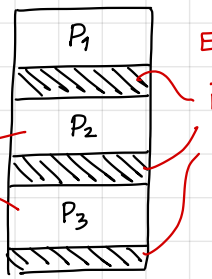
$$\text{CPU utilization} = \frac{\alpha}{\alpha + (t_1 + t_2)} = \frac{\alpha}{\max(\alpha, t_1 + t_2)}$$

usually moving same amount of memory ↳ takes 2t

for cases where swapping is done in parallel

not a correct equation, the α in the denominator might not be productive CPU time

→ Fragmentation — External - empty spaces between processes
\ Internal - empty spaces within processes



also can't shift/move processes = large overhead + all processes must be paused to move around

External fragmentation ↳ if empty spaces were all together we could also fit P_4

→ Static and Dynamic sparsness
- > Static : too much space allocated for things that could be determined at allocation time ex. too much data or stack space
- > Dynamic : too much space allocated for things determinable only at runtime. ex loading too much code that won't all be run

→ Code sharing and protection : must be able to communicate while keeping processes protected

① Contiguous : processes are loaded to a single continuous section of memory

parts show degree of multiprogramming

- Multiple-partition memory allocation → each partition is a process
  - ↳ variable partition sizes
  - ↳ creates holes : blocks of available memory that might be unusable to to small size
  - ↳ the OS keeps track of allocated partitions as well as holes (free partitions)
    - > First fit → first hole big enough — ⅓ unusable → for N blocks, .5 N blocks lost
    - > Best fit → smallest hole big enough — better for speed and utilization
    - > Worst fit → largest hole

must search the entire list

→ Fragmentation / X External : large holes created
\ √ Internal : only allocates as much space the process needs — if we need extra space during runtime we increase limit

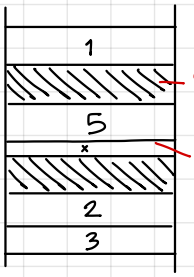
→ X CPU utilization : limited amount of processes in the M.M at once
→ X Protection : to share some data between processes we give access to the entire other process
\+ processes are all together and is very simple
\- involves changing hardware for a software problem ~ base and limit comparison in CPU

② Segmentation : process is brought into the M.M as logical units
- Each program has a segment table that stores the base and limit of each segment

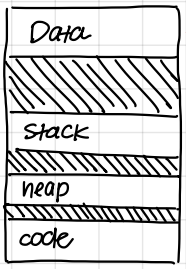
	base	limit	valid
1	1000	20	1
2	2000	40	1
3	2040	50	1
4	1020	20	0
5	1100	50	1

to share a common data space we add a segment to both processes segment tables

could be stored in registers (hardware) but table size is dynamic so we store in M.M (slower)



can fit other small segments
shared variable — must share segment



each have a base + limit
comparator must have OR

Address | 3 | 5
segment | offset
program variables are addressed relative to the segment (can be placed anywhere)

→ Fragmentation / X External : smaller and less holes, but still exists
\ √ Internal : only allocates enough space necessary — if we ever need more space for a process we add another segment ex. we need extra heap during runtime

→ √ CPU utilization : dont need to swap out entire processes to increase degree of multiprogramming
→ √ Protection : to share some data between processes only share a segment
\- large segmentation tables have large overhead for OS → take up lots of space

Static √ Dynamic X no.

★ best option

③ Paging : physical and logical memory are divided into fixed and constant sizes
→ Fragmentation — √ External : There are never any holes since all frames/pages are the same size
\ X Internal . Might have extra/unused space in a page
→ √ CPU utilization : no large overhead for swapping and large degree of multiprogramming
→ √ Dynamic sparsness : we have demand paging so we don't take up unnecessary space that won't be used/run
→ √ Static sparsness
→ √ protection : can share a page to share data

used to access index of page table

Address | p | d
page number | page offset
m−n (64−10) | n (10)

page table exists for each process in kernel

4	55
5	1

all pages are the same size so no need to store base + limit — less overhead

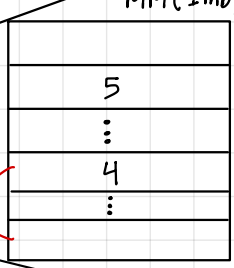
has this much addressable space

CPU 64 — TLB — limited but faster

pid	page	frame	valid
K	4	55	1

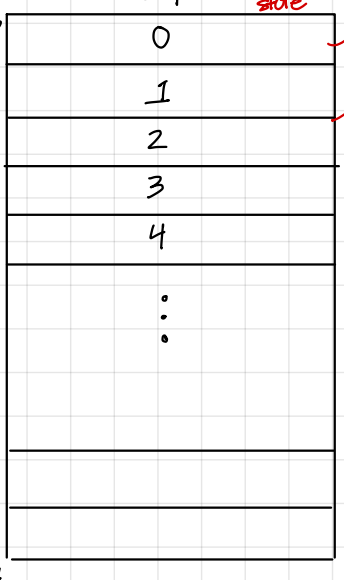
1 ~ means this page is in the M.M

MM (1 mb)



frames
each frame is 1Kb
$\frac{2^{20}}{2^{10}} = 2^{10}$ frames

Virtual Memory or backing store



we page the V.M
main() of our program
we don't need all pages of a program in the M.M at once

- V.M is divided into pages and the M.M is divided into frames page size = frame size

  byte < word < block < page/frame < chunk < buffer

- page table : translates logical to physical address for each process (≃ 1kb, 2kb, ...)
  - PTBR : base or address of the page table
  - PTLR : length of the page table
  - ↳ dynamic size → must be kept in the memory — can be overhead
  - ↳ in order to access data from a page, we once access memory to check the page table and access again for the data
  - ↳ good for sharing code, dont need to repeat common pages

- Translation Look-aside Buffer : A table containing the mapping of pages to frames (A fully associative cache)
  - ↳ Done with hardware/registers → limited space/rows
  - ↳ the larger the TLB, the more memory mapped
  1. check if p is in our physical memory
  2. if in memory, check frame number
  3. if not in memory, check page table (from memory), check frame from there (on TLB miss, new value is loaded to TLB) (LRU)
  4. access physical memory with found frame number

- Hit rate : percentage of times a page is found in TLB

$$EAT = h \times \alpha + (1-h) \times 2\alpha$$

(h: hit ratio, α: memory access latency)

- Some TLB's have a pid column to uniquely identify each process in the table
  - ↳ otherwise we must flush with each context switch

- page tables take up lots of space ⇝ if all space in logical memory is used up
  - ↳ solutions to huge page tables
    1. Hierarchical paging – two level paging tables (or more) — سؤال جيز *
       - ↳ no longer takes up large contiguous memory
       - ↳ we also divide the page table into pages
    2. Hashed page tables (search is slower)
       - ↳ the logical page number is hashed into a page table
       - ↳ chain of elements hashing to the same location
       - ↳ the entries don't necessarily have to be next to each other (no contiguous memory allocation)
    3. Inverted page table
       - ↳ flipping the page table → instead storing the active page tables in the m.m
       - ↳ lessens the memory delay

(same amount of space being taken up)

Virtual Memory

- swapper swaps entire processes, pager (demand paging) swaps individual pages
- demand paging benefits — less memory needed
  - faster response
  - better CPU utilization
- pure demand paging → starting with no pages in memory

Demand paging steps: (service the interrupt / Read the page / Restart the process)

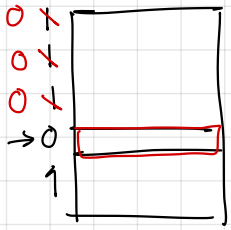
1. Trap
2. context switch
3. Determine interrupt is a page fault
4. Find page on disk
5. read from disk to a free frame (Takes time)
6. Allocate CPU to another
7. recieve interrupt from (5)
8. context switch
9. Determine intp. is from disk
10. Update the page table
11. wait till we switch back
12. restore context switch

$$(1-p)\left(\text{mem. access}\right) + (p)\left(\underbrace{\text{page fault overhead} + \text{swap in} + \text{swap out}}_{\text{page fault service time}}\right)$$

COW – Copy on Write : parent and child share resources until a copy is needed

Page replacement
1. swap out victim
2. change bit to invalid
3. swap desired page in
4. reset page table for new page

→ FIFO – Beladys Anomoly
→ LRU
→ Optimal
→ Second chance – keep looping the frames until you reach a frame that has been checked before
↳ all frames are marked 1 when initially swapped in



Global replacement → frame of a process can swap out another processes frame (process execution time varies greatly)
↳ greater throughput

Local replacement → only replace your own frames
↳ under utilized memory

Thrashing → A process is spending more time paging than executing

- High page fault rates because a process doesn't have enough pages
  - ↳ low CPU utilization
  - ↳ OS thinks its bc there's not ENOUGH processes and adds another

Local replacement focuses thrashing down to one process

Locality model → مجموعه‌ای از page‌های که با هم زیاد استفاده می‌شوند "cache"

* we must allocate enough frames in the locality

Working set → The set of pages in the most recent Δ page references
- ↳ too small → not the entire locality
- ↳ too big → multiple localities
- ↳ ∞ → entire program

Working Set Size

$D = \sum WSS_i$

↳ total demand for frames

$D >$ memory frames → thrashing will occur

the OS chooses a process to suspend

* OS gives the program enough frames for its working set

Δ = 1000 references

- A page is in the working set if it is referenced in the working set window

Page fault frequency : define upper and lower bounds on pfr
- upper → give an extra frame to process
- lower → remove a frame to process

Thrashing = high page fault rate

* if we reach upper bound and no free frames are available
  - ↳ remove process

Memory mapping a file → treating the file like v.m

Kernel memory uses a seperate memory pool from the available frames for users
1. kernel uses memory conservatively
2. to make sure its contiguous for certain I/O

Buddy System

allocates from a segment of free space

↳ requests rounded up to a power of 2

\+ can combine buddies to make a larger segment

\- internal fragmentation

Slab Allocation

slab = 1 or more contiguous pages

cache = one or more slabs

using caches to store slabs of the same size

each data structure has a cache so no fragmentation

Mass Storage Systems

- positioning time { seek time → moving the arm to get to the right cylinder
  rotational latency → rotating the disk to get to the right sector
- transfer time → rate at which data flows from disk to computer
- Attached via I/O bus
  ↳ host controller in the computer uses BUS to communicate with disk controller

Access latency = Average Access time = avg seek time + avg. rotational latency

$$\text{Average I/O time} = \text{avg. access time} + \frac{\text{amount to tranfer}}{\text{transfer rate}} + \text{controller overhead}$$

$$\text{Disk band width} = \frac{\text{\# of bytes transferred}}{\text{time of first request of transfer} - \text{time of completion last transfer}}$$

① FCFS (not good)
② SSTF (starvation) ~ good performance
③ SCAN (one end to another) ~ even if requests are dense on one side they all have to wait
④ C-SCAN → goes immediately back to start once it gets to end (no stops in between)
⑤ LOOK
⑥ C-LOOK → same thing except only goes up to the smallest

③ ④ less starvation ↓ good for systems with heavy load on the disk

Disk Attachment

① Host attached storage - accessed through I/O ports
  ↳ I/O bus architecture IDE or ATA → two drives per I/O bus
  ↳ FC : high speed fiber channel arch.
② Network attached storage ~ I/O operations have latency
  ↳ remotely attaching to file systems
  ↳ NFS and CIFS protocols
  ↳ using RPC over TCP on IP network
③ Storage-area network (SAN)
  ↳ for large storage
  ↳ multiple hosts connected to multiple storages
  ↳ connected with FC switches

Disk Formatting

Disk Formatting

- Disk must be divided into sectors → low-level formatting
  Each sector is filled with a data structure containing:
  - → Header
  - → Data area (512 kb)
  - → trailer

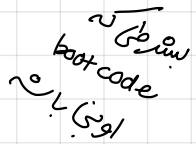
  (Info used by the disk controller)

  ↳ sector number
  ↳ error correcting code → recalculated to see if the sector is bad

- OS must record its own data structures on the disk
  ↳ partition into 1 or more cylinders → each partition treated like a seperate disk

- logical formatting : creating file system → initialize file system data structures
  ↳ file systems group blocks into clusters for I/O

- Boot cluster : where our bootstrap program is
  Boot/system disk → disk with a boot partition



Bad blocks

- list of bad blocks → recognize bad block by writing and trying to read again
- sector sparing – we have extra sectors for back up
  ↳ logically done

swap space = backing store → Normal file system / raw file (not visible)

swap map

RAID → extra disks so there's no data loss

No loss

Stable-storage implementation

↳ data is never lost

WAL : write ahead log – all modifications are written to a log before it is applied (requires stable storage)

Failure

① Successful completion
② partial failure – midst transfer previous and new data corrupted
③ Total failure (better) ↳ previous data still intact

For stable storage → system maintains 2 physical blocks per logical block
① write to 1st
② if successful, write to 2nd
③ declare complete after 2nd write

* NVRAM used as 1 of the physicals

← چرا باید حافظه مدیریت کنیم؟ utilization و performance و protection و multi processing

← چرا protection توسط Hardware مدیریت میشود؟ چون اگه نرم‌افزاری باشه latency داریم

← فرق input queue و ready queue؟

← چه زمان هایی address binding ممکن است رخ دهد؟ ① compile time ② load time ③ Execution time

← کار MMU؟ مپ کردن آدرس های مجازی به فیزیکی

← Dynamic loading؟ توی حافظه نیست تا لازم بشه

← Dynamic Linking؟ کتابخانه کمکی نمیاریم تا لازم شه

← Stub؟ فضای درنظر گرفته برای کتابخونه‌ها که جاش کتابخونه‌های لازم میاد و اجرا میشه

← static linking؟ کتابخونه‌ها موقع کامپایل به برنامه link میشوند

← swapping issues؟ I/O solutions؟ pending

← transfer time؟

← روش های اختصاص حافظه؟ contiguous، segmentation، paging

← ملاک های اختصاص حافظه؟ utilization، sparsness، fragmentation، protection

← منطق پشت worst fit؟

← حافظه از دست رفته توی First fit؟ $1/3$ بیهوده چون به ازای N تا، 0.5N تا بیهوده میشه

← compaction چیست؟

* Also keep track of free frames