4 main resources of a computer system

1. CPU
2. Disk
3. Memory (RAM)
4. I/0

1. Process Management

Main roles of the OS

1. Virtualizer: takes complicated resources such as hardware and turns them to virtual resources.
2. API: provides a system call library for programs
3. Resource Manager: ensures fair and safe sharing of resources by programs/users

Bash via commands

1. Delete all the C files in the current directory
   rm *.c
2. List all files in the current directory including hidden files and save the output to foo.txt
   ls -a > foo.txt
3. List all the C files in the current directory
   ls *.c
4. Starting with the previous command, also list all C files in all the directories below the current directory
   ls *.c */*.c
5. Show the first 7 lines of the file bar. Show the last 7 lines.
   head -7 bar
   tail -7 bar
6. Create a tar file named hw1.tar from a folder called homework1 in your current directory
   tar cf hw1.tar homework1
7. Expand the archive file 'hw1.tar' in your current working directory
   tar xf hw1.tar
8. Compress file 'msh.tar'.
   gzip msh.tar
9. Uncompress file 'msh.tar.gz'
   gunzip msh.tar.gz
10. List all the files in the current directory, sort their names, and output the results to a file called sorted.txt
    ls | sort > sorted.txt
11. Change the file foo.txt to give the owner read, write, and execute permissions, give the group and others read permissions using octal mode.
    chmod 744 foo.txt
12. Change the permissions on file bar.txt so that only the owner can read and write (use octal mode).
    chmod 600 bar.txt
13. List all files in the current directory including hidden files and save the output to foo.txt
    ls -a > foo.txt
14. Using a bash for loop, Print all the C files in the current directory
    for i in *.c; do echo $i; done
15. List all C files in the directories below the current directory
    ls */*.c
16. At the bash command prompt, write what you'd enter to see the number of lines in file foo.txt.
    wc l foo.txt
17. At the bash command prompt, write what you'd do to copy file 'files.tar' to your home directory (don't use the absolute pathname of your home directory).

cp files.tar ~

18. Changes working directory to the parent of the parent directory.
    cd ../..
19. What would you enter to discover the file location of the command 'tar' that bash would use?
    which tar
20. Using octal mode give directory 'mystuff ' (in the current working directory) the following permissions:
    owner can read, write, and execute
    group can read and execute
    others can read and execute
    chmod 755 mystuff
21. List all files in the current directory, long format
    ls -l
22. Create a tar archive file named 'code.tar' from all files .c files in subdirectory 'src' of your current directory
    tar cf code.tar src/*.c
23. Append the number of lines of file 'foo.txt' to the end of file 'counts.txt'
    wc -l foo.txt >> counts.txt

Linux commands for files: touch, cat, mv , rm , cp , cat , head,tail etc

directories: ls, mkdir, cd, mv, pwd, rmdir

Permissions

1. r - read
2. w - write
3. x - execute

Examples

-rw-r--r-- : this files allows the owner read/write permission, group can read, other people can read

drwxr-xr-x : this directory allows the owner read/write/exec permission, groups to read/write, and others to read/execute

Octal Mode/Symbolic Mode for setting permissions

CPU - Central Processing Unit

Executes instruction in main memory
Fetch
Decode

Memory Layout for a Running Program

1. **Text Segment**
   Stores Programs Code
2. **Data Segment**
   Stores Static Variables
3. **Heap Segment**
   Stores Dynamically allocated variables
4. **Stack Segment**
   Stores program call stack

CLI - Command Line Interpreter
    Display a prompt
    Accept user input
    "Parse" input to get command/parameters
    run the program
    repeat

Bash builtins: cd, pwd , echo, help

Three standard file descriptors:
0
standard input (stdin)
1
standard output (stdout)
2
standard error (stderr)

A process: running/executing program

Data Structure for Process Management

1. Process id
2. Process state
3. Process register values
4. Size of process memory

Mechanism: Used by the OS to stop/start processes.
Policy: used to decide how to schedule processes.
Fork() makes a copy of the current process.
Wait() waits for a child process to terminate.

Exec() code of the current process is replaced.
Traps - software interrupts
Interrupt table - initialized when the OS boots

Scheduling Algorithms

Turnaround time - Amount of time between job arriving and job finishing.
Response time - Amount of time before arriving and system first responding to the job

1. Shortest Job First -SJF

minimizes average turnaround time (if jobs arrive at the same time)

Example: A = 10, B = 20, C = 30
0 → 10 → 30 → 60

Response Time: 0 + 10 + 30 = 40 → 40/3 = **13**
Turnaround Time: 10 + 30 + 60 = 100 → 100/3 = **33**

2. First In First Out - FIFO

Process jobs in the order they arrive, regardless of time they need.

Example: A = 20, B = 30, C = 10
0 → 20 → 50 → 60

Response Time: 0 + 20 + 50 = 70 → 70/3 = **23**
Turnaround Time: 20 + 50 + 60 = 130 → 130/3 = **43**

3. Round Robin - RR

Example: A = 10, B = 15, C = 30

Response Time: Each job = 1ms → 3/3 = **1**
Turnaround Time: 1(30) + 3(15) + 5(10) = 125 → 125/3 = **42**

4. Shortest Time to Completion - STC

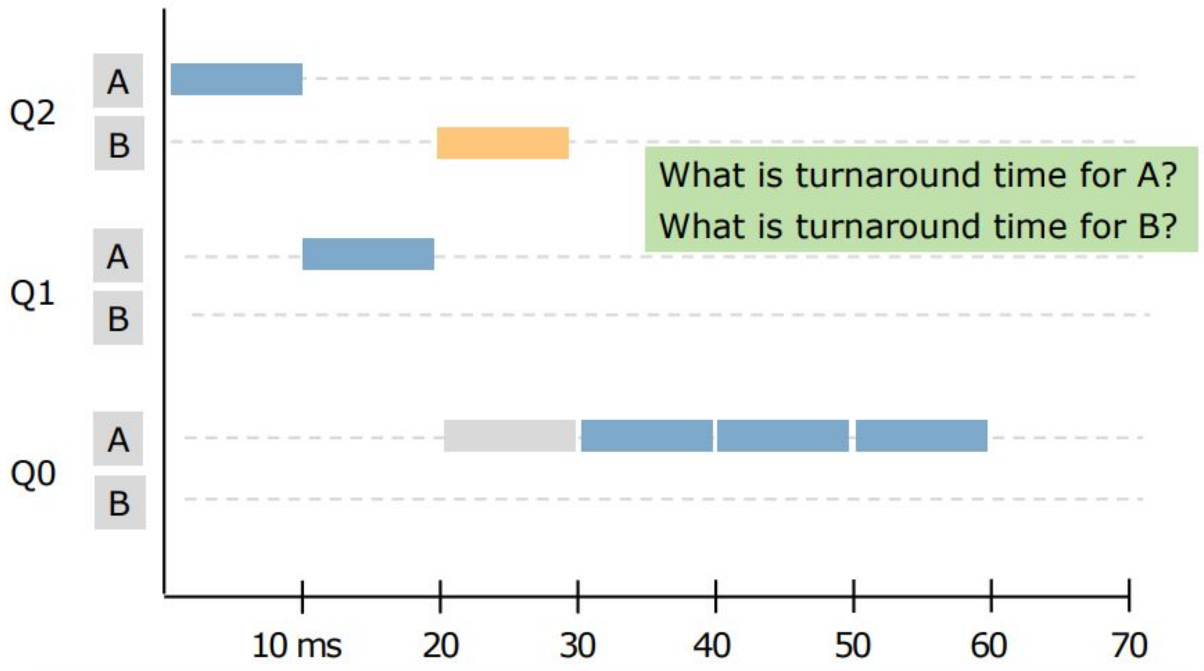
Whenever a job arrives, run the job that would finish first.

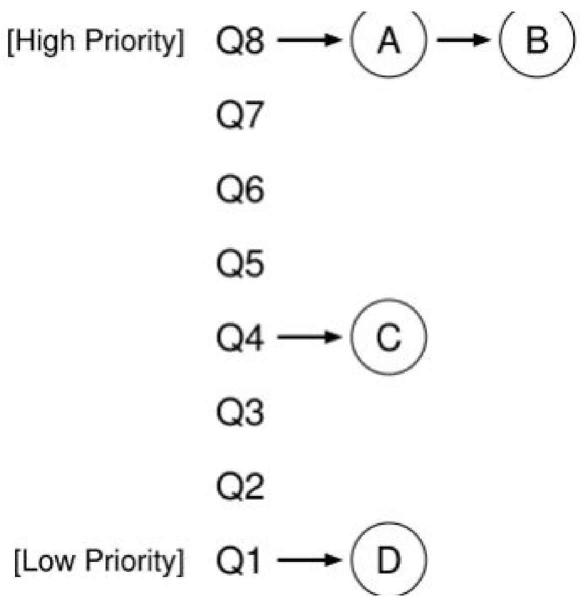
5. Multilevel Feedback Queue (MLFQ)

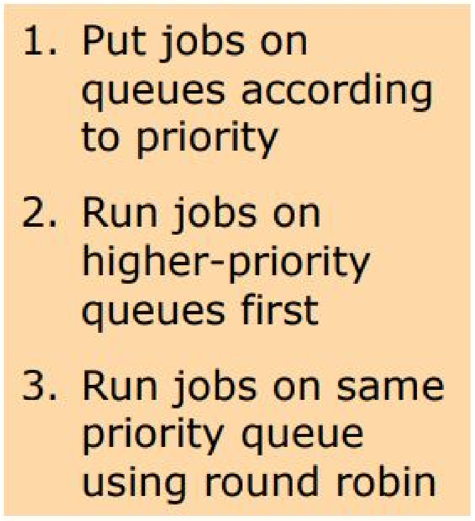
MLFQ rules for updating priorities

condition	change to priority
job starts	highest priority
job uses up time allotment at a given level	reduce priority level by 1
end of time period S	set all jobs to highest priority (boosting)

process A runs for 50 ms; shows up at time 0.
process B runs for 10 ms; shows up at time 20. 10 ms per time slice







1. Put jobs on queues according to priority
2. Run jobs on higher-priority queues first
3. Run jobs on same priority queue using round robin

2. Memory Management

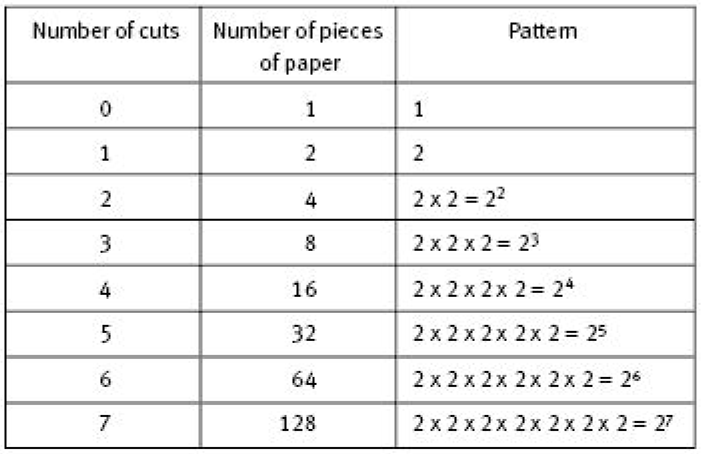
AWK

1. A .csv file usually has the same number of fields on every line. Display the possible values of the number of fields in file brain.csv, assuming fields are separated by whitespace.
   awk '{print NF}' brain.csv | sort | uniq
2. output the values of the second field of file dat.txt
   awk '{print $2}' dat.txt
3. Fields 1 and 2 of file data.csv have a name and a count, respectively. A name can appear on multiple lines of the file. Produce as output the sum of the counts for each name, with name in output field 1 and total count in field 2.
   awk '{sum[$1] += $2}END{for (nm in sum) print nm" "sum[nm]}' data.csv
4. print fields 1 and 4 of the file baz.txt, separated by a comma
   awk '{print $1","$4}' baz.txt
5. write a standalone awk script that print fields 2 and 3 of every line of that contains the text "Run JOB"
   /Run JOB/ {
   print $2, $3
   }
6. write a standalone awk script that prints the sum of the (numeric) values in field 5
   { total += $5 }
   END { print total }

7. print lines of bar.txt that are no more than 72 characters long
   ```
   awk '{if (length($0) <= 72) print}' bar.txt
   ```
8. Write a standalone awk file that will output the percent of lines in which the value of field 1 is less than 100. Consider only lines with at least one field.
   ```
   {
   if (NF > 0 && $1 < 100) m++
   if (NF > 0) n++
   }
   END {
   if (n > 0) print 100*(m/n)" %"
   else print "empty file"
   }
   ```
9. output the number of lines in temp.txt that have exactly two fields
   ```
   awk '{if (NF == 2) n++}END{print n}' temp.txt
   ```
10. list all the unique values that appear in the second field of file foo.txt
    ```
    awk '{print $2}' foo.txt | sort | uniq
    ```
11. using only awk, print the last line of file baz.csv
    ```
    awk '{x=$0} END{print x}'
    ```
12. Pipe the output of 'ls -l' to the awk script you've stored as 'summary.awk'
    ```
    ls -l | awk -f summary.awk
    ```
13. Write a standalone awk script that prints the sum of the values in field 5, only add field 5 to the sum if field 4 is greater than 0.
    ```
    {
    if ($4 > 0) total += $5
    }
    END { print total}
    ```

Paging

Virtual Address - Contains VPN and Offset



Number of cuts	Number of pieces of paper	Pattern
0	1	1
1	2	2
2	4	$2 \times 2 = 2^2$
3	8	$2 \times 2 \times 2 = 2^3$
4	16	$2 \times 2 \times 2 \times 2 = 2^4$
5	32	$2 \times 2 \times 2 \times 2 \times 2 = 2^5$
6	64	$2 \times 2 \times 2 \times 2 \times 2 \times 2 = 2^6$
7	128	$2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 2^7$

- Has VPN(Page No) and Offset
- Example: Tiny Virtual Space of 64 bytes.4 Bytes for VPN. How many bits for offset & VPN
    - Sol: 2^6 = 64 = > total 6 bits.
    - 64 Bytes for both offset and vpn. Each page is 64/4 bytes => 16 bytes.
    - 2^4 to represent 16 Bytes so 4 bits for offset.
    - 6-4=2. 2 bits for vpn. **OR** 2^2=4 bytes for VPN so 2 bits to represent.

1. If the size of the virtual address space is 32 KB, and each page is 8 KB, how many page bits in a virtual address? (enter an NBP value)
   32KB ~= 32000 and 8KB ~= 8000 32000 = 2^15 ,8000 = 2^13
   15-13=**2**

2. How many offset bits? (see previous problem) (enter an NBO value)
   13

3. If a page is 4 KB, and there are 128 virtual pages, how many page bits in a virtual address?(enter an NBP value)
   4KB ~= 4000. 4000 = 2^12, 128 = 2^7
   7

4. How many offset bits in a virtual address? (see previous problem) (enter an NBO value)
   12

5. If the size of the virtual address space is 32 MB, and there are 1024 virtual pages, how many bits in the VPN part of a virtual address? (enter an NBP value)
   10

6. How many offset bits in a virtual address? (see previous problem) (enter an NBO value)
   32MB ~= 32000000 = 2^25 25 -10 = **15**

7. If a virtual address is 10 bits, and there are 32 virtual pages, how many page bits in a virtual address? (enter an NBP value)
   5

8. How many offset bits in a virtual address? (see previous problem) (enter an NBO value)
   5

Free Space Management

Memory allocator API:

malloc (n) get a pointer to (at least) n bytes of memory

free (ptr) return memory to allocator

Three Policy Design Choices

first fit: use first item that's large enough //fast
best fit: use smallest item that's large enough //try to keep large chunks
worst fit: use largest item (that's large enough) //avoid fragmentation

1. Translation Lookaside Buffer (TLB) is located in MMU (memory management unit)

2. What is the AMAT (average memory access time) for a TLB with a miss rate of 0.5%, hit time of 1 clock cycle, and miss penalty of 50 clock cycles? (answer in clock cycles)
   Answer: .995(1) + 0.005(50) = 1.25 clock cycles

3. Concurrency

Main goal: coordinate the execution of processes
  mutually exclusive access to critical sections
  avoid deadlocks
  processes wait efficiently
Synchronization primitives
  locks, condition variables, semaphores
  some operations are blocking operations

Fair Lock - If a thread requests the lock, it will eventually get it.

a. Initialize lock: Pthread_mutex_init(&c->lock, NULL);
b. Put Lock: Pthread_mutex_lock(&c->lock);
c. Remove lock: Pthread_mutex_unlock(&c->lock);

What is the minimum number of threads in a process? **1**
Pthread_cond_wait - The blocking operation in the pthreads api

Condition variables: Threads use to signal other threads in the pthreads api.
Critical Section: A piece of code that accesses a shared resource, such as a shared variable.
Race condition: When multiple threads are running, and the output depends on the timing of their execution.
Mutual exclusion: At most one thread at a time can be in the critical section.
In the Anderson/Dahlin style of designing shared objects, each object has one lock
  True
Condition variables can be implemented with locks

False

Condition variables

A **condition variable** is a synchronization object that lets threads wait efficiently
pthread_cond_wait (cond ,lock)

1. Lock is released, calling thread is suspended and put on the condition variable's waiting list
2. Lock is re acquired before wait returns

pthread_cond_signal (cond)

1. Takes a thread off the waiting list and marks it as "ready"
2. If no thread on the waiting list, does nothing

pthread_cond_broadcast(cond)

1. Like signal, but takes all threads off the waiting list

Semaphore Syntax

sem = Semaphore(1)
sem.signal() //i.e. Increment
When a thread increments the semaphore, if there are threads waiting, one of them gets unblocks
sem.wait() // i.e. decrement
When a thread decrements the semaphore, if the result is negative, the thread blocks itself and can't continue until another thread increments the semaphore

4. File Management

Main goal: virtualize persistent storage
Storage is virtualized as files and directories

Drive Performance
Reading/Writing of a drive involves 3 steps

1. Rotational Delay
2. Seek
3. Transfer

"access time" = rotational delay + seek time

Hard Drive

Given: access time is 10 ms, transfer rate is 50 MB/s

How long will it take to do 200 random reads of 32 KB each?

hint: 200 * (access time + transfer time)

transfer time: $\frac{1\text{ s}}{50\text{ MB.}} * \frac{1\text{ MB}}{1000\text{ KB}} * \frac{1000\text{ ms.}}{1\text{ s}} * \frac{32\text{ KB}}{} = .64\text{ ms}$

200 * (10 ms + .64ms) = 2128ms = 2.1 s

Q1

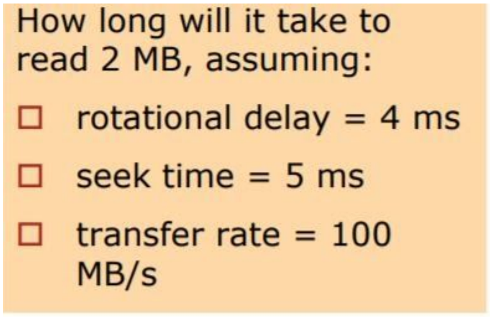
If a disk spins at 7200 RPM, what is the average rotational delay?

so about 4.2 ms per ½ revolution

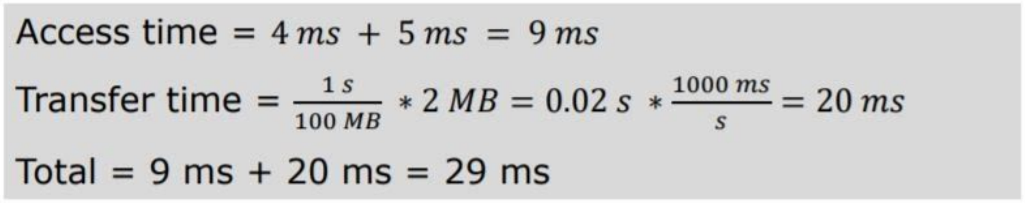
in one go:

$$\frac{min}{7200\,rev} * \frac{60\,sec}{min} * \frac{1000\,ms}{sec} = \frac{600\,ms}{72} = 8.3\frac{ms}{rev}$$

Q2


How long will it take to read 2 MB, assuming:

- rotational delay = 4 ms
- seek time = 5 ms
- transfer rate = 100 MB/s


Access time = $4\,ms + 5\,ms = 9\,ms$

Transfer time = $\frac{1\,s}{100\,MB} * 2\,MB = 0.02\,s * \frac{1000\,ms}{s} = 20\,ms$

Total = 9 ms + 20 ms = 29 ms

Hard Drives

A hard drive has 8 heads and 4 Platters.

Physical unit of storage = a block.

Reading/Writing of a drive involves 3 steps:
1. wait for sector to rotate underneath head (rotational delay)
2. move the head to the right track (seek)
3. actually transfer the data (transfer)

Calculating average rotational delay
If a disk spins at 7200 RPM, what is the average rotational delay?

7200 RPM = 7200 $rev\ min$

7200 $rev * min$ = 120 $rev\ min$ 60$sec\ sec$

1sec $*$1000ms=8.3ms 120 $rev\ sec\ rev$

so about 4.2 ms per 1/2 revolution

Given: access time is 10 ms, transfer rate is 100 MB/s We need to read 3.2 MB.

1. How long to do a sequential read of 3.2MB?

■ time: access time + transfer time ~ 42 ms
■ overall rate: 3.2 MB/0.042 s ~ 76 MB/sec

2. How long to do 100 random reads of 32KB each?

■ time: 100 * (access time + transfer time) ~ 1030 ms
■ overall rate: 3.2 MB/1.03 s ~ 3.2 MB/sec

**Overall rate is about 24x better for sequential workload

Disk scheduling (I/O scheduling)

The disk scheduler decides the order in which disk requests should be processed.

Some scheduling methodologies:

1. Shortest Seek Time First (SSTF)
2. Elevator Scheduling

Shortest Seek Time First- put requests closest to the current track at front of the queue.

Linus Elevator
■ performs merging and sorting (replaced in 2.6)

Deadline I/O Scheduler
■ gives up elevator approach if old requests exist

Anticipatory I/O Scheduler
■ waits a few ms after a seek for more read requests

(Linux 2.6 default)
Completely Fair Queuing I/O Scheduler■ one queue for each process
■ designed for multimedia workloads

Noop I/O Scheduler
■ maintains request queue in FIFO order

Language Processing

Lexical Analysis

expr := NUM | ID | expr + expr
Derive ID + NUM

$$\begin{aligned}&\text{expr} \rightarrow \\ &\qquad \text{expr + expr} \rightarrow \\ &\qquad\qquad \text{ID + expr} \rightarrow \\ &\qquad\qquad \text{ID + NUM}\end{aligned}$$

NUM: one or more digits
ID: A letter followed by zero or more digits/letters

1. Break inputs into “tokens”
2. Parse the tokens
    a. When “tokenize”, store the values of some tokens for later use.

Put API for Lexical Analyzer

Use **match()** when you know what the next token must be.

Predictive Parsing

Recursive descent parsing : "top down method of syntax analysis in which we execute a set of recursive procedures to process the input"
Predictive parsing: "a form of recursive descent parsing in which the lookahead symbol determines the procedure selected for each non terminal"

To prevent recursion, BNF transforms the expressions into something equivalent so expr can be used.

General Rule
A::= A $\boldsymbol{\alpha}$ | β
To

A::= β R
R::= **α** R | “ “

Ex: rewrite to eliminate left recursion

```
expr ::= expr + expr
        | var
        To
expr ::= var expr1
expr1 ::= + expr expr1 | “ “
```

How to deal with empty productions ie “ “

```
Make it default case
        default;
        ;
```

Write a BNF grammar that creates “a”, “(a)”, “((a))”...

expr ::= a | (expr)

Write a BNF grammar that creates only “1 + 1” and “1 + x”.

```
expr ::= 1 + expr1

expr1 ::= 1 | x
```

Rewrite to eliminate left recursion. Assume var is a terminal value.

expr ::= expr + expr | var

```
expr ::= var expr1

expr1 ::= + var expr1 | “”
```

Direct Memory Access (DMA) - uses a special hardware device that transfers data between devices and memory without CPU help.

Parse trees

Rules

1. Root is labeled with the start symbol

2. The symbols in one of its productions become child nodes
3. Continue until all leaf nodes are terminal symbols

Ex: expr = num
| var
| expr + expr   Derive “x+1” using a parse tree

Solution

```
       expr
      /  |  \
   expr + expr
      |      |
  var      num
    x        1
```

A grammar is ambiguous is, for some string that can be derived from the grammar, there is more than one parse tree.

Ex: expr ::= var (expr)   apply left factoring
| var

Solution

```
expr ::= var expr1
expr1 :: = ( expr ) | “ “
```