Name: WANG Yuqi **Student ID**: [redacted]

Introduction

OS Components

Level 1: Kernel
- Process scheduling, memory allocation, syscall ...

Level 2: Syscalls
- *Interface* between user and applications and kernel
- File I/O: `open()`, `write()` ...
- Process Control: `exec()`, `exit()`, `fork()` ...
- Communication: `socket()`, `send()`, `recv()` ...

Level 3: Shell & System Programs
- Shell: CLI. Translates commands → syscalls
- System Programs: Apps built on syscalls. (compilers, disk formatter)

OS Functions

Process Management: Create, delete, suspend, resume, schedule
- *Multiprogramming*: Multiple programs loaded in RAM at the same time (switch when I/O bound)
- *Multitasking*: Rapidly switch between processes on time slice (illusion of simutaneous execution)

Memory Management: Register < Cache < RAM < 2nd Storage
- Tasks: Allocate / Deallocate, Protect unauthorize access, Optimization (paging, swapping, caching)

Storage (File) Management: Abstract storage into files / directories
- Task: Create / delete / open files. Enforce permissions (rwx). Backup, fragmentation management.

Device Management I/O Subsystem **&** Mass-Storage Management
- *IO Subsystem*: Buffering (temporary storage for data in transit), Caching (store frequently used data in RAM), Spooling (store data for later use), Device Drivers (interface between OS and device)
- *Mass-Storage*: Free-space management, Storage allocaiton, Disk scheduling (minimize head movements)

User Command Interface: CLI, GUI. Translate user input → syscall

Interrupts & System Calls

Interrupt Processing

Basic Idea: Major event → Signal to the CPU → Seize the CPU **Priority**:
- *Maskable Interrupt*: low priority. Ignored / handled later
- *Non-Maskable Interrupt* (**NMI**): high priority. Handled immediately
  - Higher priority NMI, can interrupt lower priority NMI
  - Otherwise, low priority NMI waits for prev NMI to finish

Procedure:
1. Non-maskable interrupt occurs
2. Save current CPU state (PC, Register) to stack
3. Look up an *interrupt vector table* (**IVT**) to find appropriate ISR
4. [User ⟶ Kernel Mode]: Jump to *Interrupt Service Routine* (**ISR**)
5. **[In Kernel Mode]:** Execute ISR
6. Restore CPU state from stack
7. Re-enable interrupts

Interrupt Types

[1] *Program Interrupt*
- Windows: `Ctrol-Alt-Del` → Secure Attention Sequence (SAS) → Hardware Interrupt. Major, unique system-level interrupt.
- Unix: `Ctrol-C` → Software Interrupt (User Trap). General purpose software termination
- Hardware Interrupts are reliable. BUT, impossible to catch
- Software Interrupts let program handle interrupt (cleanup resource)

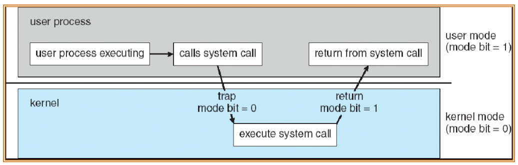


Figure 1: Dual Mode Operation: distinguished by *mode bit*

[2] *I/O Interrupt*

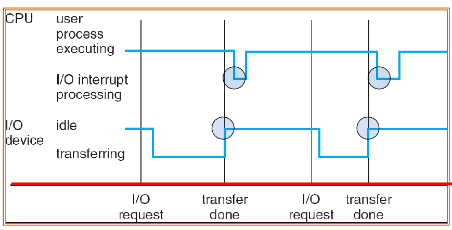


Figure 2: CPU interrupt needed 1 tick after transfer complete

Must know calculations:
- Sync: Total time = sum (e.g., 128ms for 20+5+3+100).
- Async: Total time ≈ longest task (100ms), more interrupts.

Device-Status Table:
- Managed by kernel.
- Each entry: contains type, address, state
- State updated after an interrupt (not function, idle, busy)

[3] *Timer Interrupt*
1. Interrupt occur after specific time period
2. Kills process / take back resource (e.g., unlock file)
3. In Unix: *clock routine*. Trigger by hardware per $\frac{1}{60}$ sec

System Calls

Goal: Abstract away hardware details. Provide interface for user.

Procedure

1. User process executes syscall
2. Look up *System Call Table*
3. CPU switch to kernel mode and execute syscall
4. Return status & results.

Types

Category	Functionalities	Windows	Unix
Process control	- Create process, terminate process - Load, execute - Get process attributes, set process attributes - Wait event, signal event - Allocate and free memory	CreateProcess(), ExitProcess(), WaitForSingleObject()	`fork()`, `exit()`, `wait()`
File management	- Create file, delete file - Open, close - Read, write, reposition - Get file attributes, set file attributes	CreateFile(), ReadFile(), WriteFile(), CloseHandle()	`open()`, `read()`, `write()`, `close()`
Device management	- Request device, release device - Read, write, reposition - Get device attributes, set device attributes - Logically attach or detach devices	SetConsoleMode(), ReadConsole(), WriteConsole()	`ioctl()`, `read()`, `write()`
Information maintenance	- Get time or date, set time or date - Get system data, set system data - Get process, file, or device attributes - Set process, file, or device attributes	GetCurrentProcessID(), SetTimer(), Sleep()	`getpid()`, `alarm()`, `sleep()`
Communications	- Create, delete communication connection - Send, receive messages - Transfer status information - Attach or detach remote devices	CreatePipe(), CreateFileMapping(), MapViewOfFile()	`pipe()`, `shm_open()`, `mmap()`
Protection	- Get file permissions - Set file permissions	SetFileSecurity(), InitializeSecurityDescriptor(), SetSecurityDescriptorGroup()	`chmod()`, `umask()`, `chown()`

System Programs

1. **File Management**
   - Functions: Create, delete, copy, rename, print, list, and manipulate files/directories.
   - Example: `ls` command in UNIX lists directory contents.
2. **Status Information**
   - Functions: Retrieve system data (e.g., date, time, memory, disk space, user count).
   - Complex versions: Performance monitoring, logging, debugging.
   - Output: Terminal, GUI, or files.
   - Example: `top` command in Linux displays system resource usage.
3. **File Modification**
   - Tools: text editors, search command, text transformation utilities.
   - Example: `vim` for editing files, `grep` for searching file contents.
4. **Programming-Language Support**
   - Tools: Compilers, assemblers, debuggers, interpreters.
   - Example: GCC for C/C++, Python interpreter.
5. **Program Loading and Execution**
   - Tools: Loaders (absolute, relocatable), linkage editors, overlay loaders, debugging systems.
   - Example: `gdb` for debugging C programs.
6. **Communications**
   - Functions: Facilitate virtual connections between processes, users, and systems.
   - Examples: Email, remote login (`ssh`), file transfer (`scp`).
7. **Background Services (Daemons)**
   - Functions: Long-running processes for essential tasks (e.g., network connections, process scheduling, error monitoring).
   - Examples: Network daemons, print servers.
   - Note: Daemons often run at boot time and continue until the system shuts down.

Parameter Passing

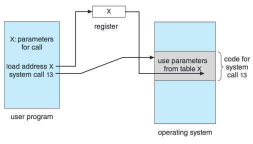


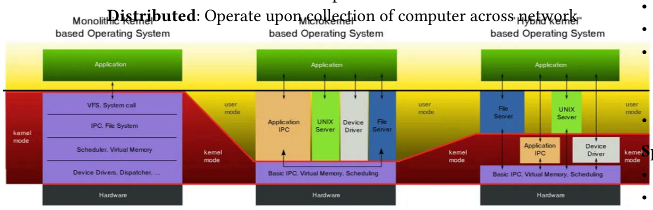
Figure 2.7 Passing of parameters as a table.

1. **Registers**: fast, but limit space.
2. **Memory Block/Table**:
   - Put param in mem block
   - Put address of param mem blocks in registers
3. **Stack**: for variable-len data.

OS Types & Structure

Structure Type	Description	Examples	Pros	Cons
Monolithic	Single address space containing all OS functionality	Traditional UNIX, Linux base	High performance, efficient syscalls	Complex maintenance, tightly coupled
Layered	Hierarchical layers with defined interfaces; Layer N uses only layers 0 to N-1	-	Simple debugging, clear separation	Performance overhead, layer definition challenges
Microkernel	Minimal kernel (process/memory management + IPC); Services run as user processes	Mach (Darwin kernel component)	Extensible, reliable, secure	IPC overhead, complex coordination
Modular	Core kernel + Loadable Kernel Modules (LKMs); Dynamic loading/unloading of services	Linux modules, kernel extensions	Flexible, efficient communication	Potential reliability issues

Real-time: Processes have completion deadlines must be met
Distributed: Operate upon collection of computer across network



Type	Definition	Example of Use
Batch OS	Data or programs are collected grouped and processed at a later date.	Payroll, stock control and billing systems.
Interactive OS	Allows the user and the computer to be in direct two-way communication	Select from a menu at ATM.
Real-time OS	Inputs immediately affect the outputs. Timing is critical	Control of nuclear power plants, air traffic control systems.
Network OS	Allow a computer on a network to serve requests from other computers for data and provide access to other resources such as printer and file systems.	Manage simultaneous access by multiple users
Multiuser OS	Handle many people running their programmes on the computer at the same time	A number of terminals communicating with a central computer which allocates processing time to each terminal in turn.
Multiprogramming OS	Ability to run many programmes apparently at the same time.	Mainframe systems. Each job is allocated a small amount of processing time (*time slice*) in turn.
Multitasking OS	The ability to hold several programmes in RAM at one time but the user switches between them.	Usually uses GUI's. Facilitates import and export of data.

I Hate Shell Script

`ls` usage
- `ls -l`: list in long format (detailed, verbose)
- `ls -a`: list all (include hidden files)
- `ls -R`: list subdirectories recursively
- `ls [hello]`: match single-character file `h`, `e`, `l`, **or** `o`
- `ls [hello]*.c`: files starting with `h`, `e`, `l`, `o` and ending with `.c`
- `ls hello[1-3]*`: file starting with `hello` followed by `1`, `2`, or `3`

Pipe: `(ls; cal) | wc` - (ls; cal) creates a subshell; within this subshell, `ls` and `cal` run in two subprocesses (so 3 processes in total on the left hand side). Then, output piped to `wc` which runs in another

Unfamiliar Commands:
- `echo something\`: \ allow continue typing command on new line
- `echo $$`, `$$` evaluates to PID of current shell
- `wc`: outputs line count, word count (by space separate), **bytes** count, file name (if called with file name as parameter, eg. `wc test.txt`)
- `source`: execute a script or batch file
- `history`: list recent cmds; `!n` run command n; `!!` run last command
- `| more`: pipe output to `more` to display one screen at a time
- `export`: make variable available to subshells, but not parent shell
- `` `cmd` `` or `$(cmd)`: command substitution. ` has undefined behavior
- Exponential in shell is **, not ^
- `ps -ef | more` will display detailed information about all processes

Special Parameters
- **`$*`**: All positional parameters as a single string
- **`$@`**: All positional parameters as separate strings
- **`$1, $2, ...`**: Positional params (arguments to script), start from 1
- **`$#`**: Number of positional parameters (arguments)
- **`$?`**: Exit status of the most recently executed command
- **`$$`**: Process ID of the shell
- **`$0`**: Name of the shell or shell script
- **`${!#}`**: evaluates to the last argument

Exist Status: 0 for success, non-zero for failure, −1=255, 256=0
Access Privilages: [dir or file] [user] [group] [other] `-rwx r-- r--`
- `chmod g+r f`: give group read access to file
- `chmod o-r f`: remove other's read access to file
- `chmod u+x f`: give user execute access to file
- `chmod -R g+rw folder`: recursively give group read/write access to folder and all its content within.
- `chmod 777 f`: give all access to file
- Each digit represents user, group, other. E.g., $6 = (110)_2 =$ rw-
- **First character (before `rwx`, eg `drwx-rwx-rwx`)**:
  - `p`: FIFO (named pipe)
  - `-`: Regular file
  - `d`: Directory
  - `l`: Symbolic link
  - `s`: Socket
  - `c`: Character device
  - `b`: Block device

`$PATH` separated by `:` || To add current dir: `PATH="$PATH:."`

Regex:
- *`^` means beginning of line;*
- *`$` means end of line;*
- *`.` means any character;*
- *`[abc]` any one of a, b, c;*
- *`[^abc]` anything except a, b, c;*
- *use `\` for escape.*

`^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$`

grep Commands: `grep poly *.c` list all C files containing `poly`;
`grep -i poly *.c` same above, but case insensitive;
`grep -l -i '^poly' *.c` print only **names** of file with lines beginning with poly
`grep -h -i '^poly' *.c` print only lines beginning with poly

- Interpreted: PHP, Ruby, Perl, Bash, Lua, Tcl, Basic, HTML, JS, Py
- Compiled: COBOL, Fortran, Pascal, Ada, Lisp

Scripting Language:
- Advantages: fast to write, small size, partial execution, directly access OS service and execute commands, directly process output from other programs;
- Disadvantage: slow execution, weak data structure / typing, odd syntax error-prone, occasionally buggy

Random Concepts
- Run commands together: `cmd1; cmd2; cmd3`
- Run cmd if previous success: `cmd1 && cmd2`
- Global variables by convention UPPERCASE, local lowercase
- For local variables use `local var=value`
- **DO NOT** add space before and after `=`
- **DO NOT** create a script called `test`, interfer with built-in `test` cmd
- **`let`** keyword
  - `let x=a+b` is equivalent to `x=$((a+b))`
  - but `x=a+b`, x is the literal string `a+b`
- **changes to the list** inside the loop will ***not*** affect the loop count

Special Option Variable
- **`history`**: enable command history, default = on
- **`noclobber`**: prevent overwrite file when I/O redirection, default = off

- **ignoreeof**: stop shell from exiting when EOF is sent, default = off
- **allexport**: *automatically* export all modified variables, default = off
- To turn on/off, use `set -o/+o variable`:
  - `set -o noclobber` (turn on noclobber feature)
  - `set +o noclobber` (turn off noclobber feature)

Built-in Variables
- `HOME`: Full pathname of home dir; default destination for plain `cd`
- `PATH`: list of directories for executables and scripts, `:` separated
- `PS1`: Primary prompt string; default 's-v$ ' (system-version then $).
- `PS2`: Secondary prompt string for line continuation; default '> '.
- `PWD`: Current working directory (updated by `cd`).
- `HOSTNAME`: Current machine name.
- `UID`: Internal Unix user ID. `PPID`: Parent process ID.
- `HISTSIZE`: Max history lines; default 500.

Magic Numbers
- `#!` is *magic number*. Look at the path following `#!` and start that p as an interpreter.
- *Big Endian*: Stores the most significant byte first (same order as readable representation).
- *Little Endian*: Stores the least significant byte first (reverse order to human-readable

representation).
- `od -c -x file | more` to see magic number of file
  - `od` means octal dump, `-c` means char,
  - `-x` means hex in little-endian, `-t x1` means hex in big-endian

```
[ammoham@apollo] od -c -x lab1.pdf | more
0000000   %   P   D   F   -   1   .   4  \r  \n   % 265 265 265 265  \
            5025    4644    312d    342e    0a0d    b525    b5b5    0db
0000020  \n   1       0       o   b   j  \r  \n   <   <   /   T   y
            310a    3020    6f20    6a62    0a0d    3c3c    542f    707
```

Array
- `declare -a arr`: create an empty array
- `arr=(1 2 3 4 "str")`: create array
- `${arr[2]}`: access element (index start from 0)
- `${arr[@]}`: access all elements, `${#arr[@]}`: length of array
- `${arr[@]:1:2}`: slice array from index 1 to 2
- **@ vs. ***: completely the same unquoted. Different when quoted.
  - `"${arr[@]}"`: five strings || `for i in "${arr[@]}"`, output 5 lines.
  - `"${arr[*]}"`: one string || `for i in "${arr[*]}"`, output 1 line.
- **single quote '**
  - no variable expansion: `echo '$HOME'` prints `$HOME`
  - no command substitution: `echo '$(ls)'` prints `$(ls)`
  - no escape sequence: `echo '\n'` prints `\n`,
  - no special character interpretation: `echo '!'` prints `!`
- **double quote "**
  - Variable expansion: `echo "$HOME"` prints `/home/user`
  - Command substitution: `echo "$(ls)"` prints output of directory
  - Arithmetic expansion: `echo "$((2+3))"` prints 5
  - Escape sequence: `echo "\n"` prints newline
  - **no quotes**:
    - `${arr[@]}` or `${arr[*]}` behave the same
    - `${#arr[@]}` or `${#arr[*]}` behave the same
    - Same as `expand(' '.join())`
      - `expand(arr)` is variable expansion on each element of `arr`
  - **with quotes + [@]**: `"${arr[@]}"`: preserve list as is
  - **with quotes + [*]**: `"${arr[*]}"`: equivalent to `' '.join(arr)`
- `declare -a arr`: declare array || `declare -p arr`: print array

Conditionals
- `if [ condition ]; then ... fi`.
- `[]`, POSIX-compliant. Variables must be quoted to handle space.
- `[[]]`, bash-specific.
  - Allows pattern matching (`[[ $var == pattern* ]]`),
  - regex (`[[ $var =~ ^[0-9]+$ ]]`)
- Logic operators (`[]` and `[[]]` both supports):
  - `!`, `-a` (&&), `-o` (||). `-eq`(==), `-ne`, `-gt`, `-ge`, `-lt`, `-le`

Case-Statement

```
case $code in
  comp1*) echo "level 1" ;;
  comp2*) echo "level 2" ;;
  comp[34]*) echo "senior" ;;
  comp*) echo "invalid code" ;;
  *) echo "bruh what?" ;;
esac
```

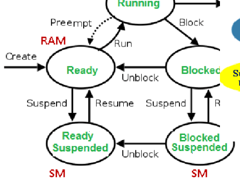
- Each pattern ends with)
- Pattern: constant or wildcards
- Case sensitive
- `;;` no fall through.
- `;;&` fall through

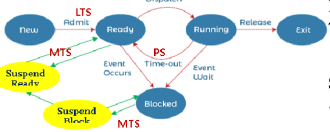
For-loops: `for var in "${list[@]}"; do ... done`
While-loops: `while [ condition ]; do ... done`
Until-loops: `until [ condition ]; do ... done`

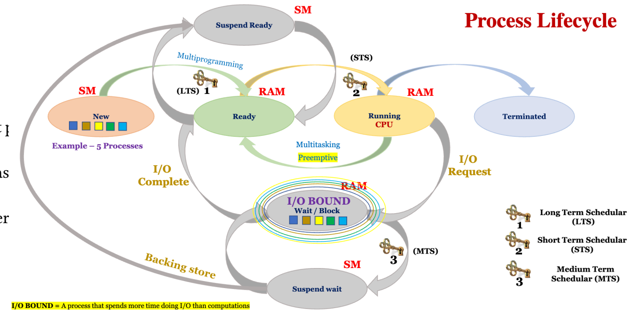
File Testing: `-e` exists, `-d` is dir, `-f` is plain file, `-r` readable, `-w` writable, `-x` executable, `-z` empty

Numeric Calculation: `result=$((expression))`

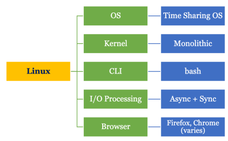
Input/Output
- `read var`: read input from user, store in var
- `read -p "Prompt: " var`: print prompt, then read input
- `read -a arr`: read elements into an array "arr"
- Redirection **(Sequential!)**
  - `<` and `>`: redirect input and output.
  - `script < input.txt > output.txt`
  - `>>`: append to file, `&>>` same as `>>` but also **append error**
- Pipes **(Concurrent!)**
  - `p1 | p2` ≡ `p1 > tmp; p2 < tmp`
  - `p1 | p2 | p3` ≡ `p1 > tmp1; p2 < tmp1 > tmp2; p3 < tmp2`
  - `p1 < in | p2 > out` ≡ `p1 < in > tmp; p2 < tmp > out`

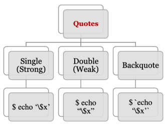


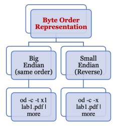




System call	Feature
int execl(const char *path, const char *arg0, ...);	Execute a program at pathname **path**, last argument must be **NULL**.
int execlp(const char *file, const char *arg0, ...);	Execute a program named by **file**, last argument must be **NULL**.
int execle(const char *path, const char *arg0, ..., char *const envp[]);	Same as execl, but access environment variables via ***envp[]**.
int execv(const char *path, char *const argv[]);	Same as execl, but arguments are stored in ***argv[]** instead.
int execvp(const char *file, char *const argv[]);	Same as execlp, but arguments are stored in ***argv[]** instead.
int execve(const char *path, char *const argv[], char *const envp[]);	Same as execv, but access environment variables via ***envp[]**.







Process Management

- **LTS** aka *Job Scheduler*: which process to enter *ready queue*
  - Controls the degree of multiprogramming
  - No long-term scheduler in Unix and Windows
- **STS** aka *CPU Scheduler*: which process to allocate CPU next
- **MTS** when too many processes compete CPU, remove some from ready queue; When few process in CPU, add them back
  - *MTS* can cause ***Thrashing***: when CPU spends more time swapping than executing
- Cooperative Process: share variable, mem, code, resource (CPU, I/O)
  - need synchronization
- Independent Process: no shared resource

Cooperative Process
- **Producer-consumer** Streaming, Web Server
- **Reader-writer** Example: Banking Systems
- **Dinning Philosopher**
  - Cooperating process that need to share limited resources

Process Control Block (PCB)
- Process State: New, Ready, Running, Waiting, Terminated
- Program Counter: address of next instruction
- CPU Registers: accumulators, index registers, stack pointers
- CPU Scheduling Information: priority, scheduling queue
- Memory Management Information: limit of memory boundary
- Accounting Information: CPU used, process ID
- I/O Status Information: list of open files, devices

`exec()`: replace current process with new process:
- **Same**: PID, file descriptor, cwd, user / group ID, resource limit
- **Overwritten**: mem, stack, data, heap, program counter, registers

`fork()`: create new process (child) from existing process (parent)
- **Parent**: return child PID, child PID > 0, error return −1
- **Child**: return 0. Call `getppid()` to get parent PID

`wait()`: parent blocked until **one of** the childs terminates

Interprocess Communication (IPC)

General Concepts
Indirect Communication:
- Messages are directed and received from *mailboxes* aka *ports*

Direct Communication:
- Process must name each other directly
  - `send(P, msg) receive(Q, msg)`
- Links established automatically by OS. Each link associate with **exactly one pair** of processes.

Blocking (*Synchronous*):
- Sender blocked until message received by receiver
- Receiver blocked until message is available from the sender

Non-blocking (*Asynchronous*):
- Sender sends message and always continue
- Receiver receives valid message or `null` if message not available

Buffering (*Message Queues*) – sender sent, recv haven't read yet
- Zero Capacity: No msg buffered. Sender must wait for receiver, v.v.
- Bounded Capacity **(common)**: Fixed size queue. Sender wait if full.
- Unbounded Capacity: Infinite size queue. Sender never wait.

Process: Top-level execution container. Separate Memory Space
Threads: Runs *inside* a process. Share Memory Space

Unix-specific IPC
Shared Memory Mechanism (faster than message passing):
- **Method 1**: Special Syscalls to create / remove / write / read from shared kernel memory space. || **Method 2**: Use `Pthreads` library, whose threads naturally share memory space due to same process

Message Passing Mechanism: *pipes* or *sockets*
- `write` one way. `talk` two way.
- Pipes: *Unnamed pipes* OR *Named pipes*
  - Unidirectional, FIFO, unstructured data stream
  - Fixed size Buffer allocated in kernel.
  - ***Unnamed Pipes***:
    - Create by syscall `pipe()`. Kernel buffer allocated.
    - Two file descriptors (fd) as a 2-element array
      - `fd[0]` read end, `fd[1]` write end
      - Process will read from `fd[0]` and write to `fd[1]`
      - e.g., `read(fd[0], buf, count)`, `write(fd[1], buf, count)`
    - **Usage**:
      1. First create `fd[]` then `fork()`.
      2. Now, both child and parent have access to `fd[]`
      3. `close(fd[0-or-1])` in child and parent (removes their access, does not actually destroy the file descriptors)
  - ***Named Pipes***
    - *named pipe* is an actual file created by `mkfifo()`
      - `mkfifo()` return 0 on success, −1 on error
    - After creation, any process knowing the name can `open()` it
    - They can then communicate via `read()` and `write()`
    - pipe won't delete when processes end. Must use `unlink(pipe)` at the end of process to remove it (good practice)
  - **Choices**
    - *Unnamed pipes* when only parent-child communication, safe
    - *Named pipes* when unrelated processes or across sessions

Syscalls
Below uses `#include<unistd.h>`
- `ssize_t write(int fd, const void *buf, size_t count);`
  - `fd`: file descriptor, `buf`: data to be written, `count`: number of bytes to write. If `count` > content in `buf`, write random shit at end.
- `ssize_t read(int fd, const void *buf, size_t count);`
  - `fd`: file descriptor, `buf`: where data saved, `count`: # of bytes to read

Below uses `#include<sys/types.h>` `#include<sys/stat.h>`
- `int mkfifo(const char *pathname, mode_t mode)`
  - `mode = 0777`, means `prwxrwxrwx`
- `int open(const char *pathname, int flags)`
  - Also needs `#include <fcntl.h>`
  - flag for reading `O_RDONLY`, for writing `O_WRONGLY`, for r+w `O_RDWR`
- `int open(const char *pathname, int flags, mode_t mode)`
  - `mode` is the mode to open the file – reading or writing
  - Also needs `#include <fcntl.h>`

Communication Topology

Cables
- **Star Topology**: $n-1$ cables. Bottleneck at parent. Few pipes. 2 hops needed at max.
- **Ring Topology**: $n+1$ cables. Many hops. Few pipes.
- **Linear Topology**: $n+1$ cables. Many hops. Few pipes
- **Fully Connected**(fc): $\frac{n(n-1)}{2}$ cables. Many pipes. Single hop
- **Tree Topology**: Few pipes. Many hops (log?)
- **Mesh Topology**: $\frac{n(n-1)}{2}$ cables. Follow some structure (e.g., grid). Less pipes then fc. Well established non-neighbor routing. Flexible.
- **Hypercube Topology**: Fewer pipes than fc. Well established non-neighbor routing. Flexible.

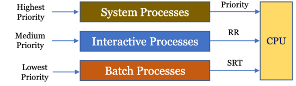
Number of pipes
- **Star**: 2(n-1) pipes. Bottleneck at parent node. Single-point-of-failure
- **Ring**: 2n pipes. No particular bottleneck. Distributed communication
- **Linear**: 2(n-1) pipes. Node failure separates the network. • **Fully connected**: n(n-1) pipes. No bottleneck. Each node has 2(n-1) connections
- **Tree**: 2(n-1) pipes. Bottleneck spread throughout hierarchy. • **Mesh**: Between 2(n-1) and n(n-1) pipes. Each node connects to 4 neighbors. • **Hypercube**: Between 2(n-1) and n(n-1) pipes. Each node connected to $\log_2(n)$ neighbors

CPU Scheduling

Basic Considerations
- **CPU utilization**: % of CPU time used for real work.
- **Throughput**: # of processes completing execution per time unit.
- **Turnaround time**: Time to finish process since arrival.
- **Waiting time**: Time a process spends waiting in ready queue
- **Response time**: Time from request submission until first response.
- **Completion Time**: The time when completed.

Scheduling Algorithms
- **Preemptive**: Priority Scheduling, Round Robin (RR), Shortest Remaining Time (SRT)
- **Non-preemptive**: First Come First Serve (FCFS), Shortest Job First (SJF) Multi Level Queues

Non-preemptive:
- **First Come First Serve (FCFS)**: First process in, first process out.
- **Shortest Job First (SJF)**: Shortest job first.
- **Multi Level Queue**: Ready queue devided into multiple queues.
  - Each queue has own scheudling algoirhtm
  - Scheduling **between queues**.
    - Fixed Priority Scheduling: each queue has own priority
    - Time Slicing: each queue gets a time slice (like RR)



Preemptive:

- **Shortest Remaining Time (SRT)**: Preemptive version of SJF.
- **Round Robin (RR)**:
  - Predefined *time quantum* (e.g., $q = 10$ms).
- **Priority Scheduling**:
  - *Priority Number* assigned to each process. Higher priority first.
  - SJF is priority scheduling where priority is the time to completion.
  - FCFS is priority scheduling where priority is the time of arrival.

Strengths & Weaknesses

- SJF / SRT: Optimal for minimizing waiting time, good for batch jobs.
  - SJF proven to have lower TAT, **optimal** for non-preemptive
  - SRT proven to have lower TAT, **optimal** for preemptive
- RR: fair to all process, good for interactive jobs.
- FCFS: simple, no starvation since all will eventually execute. But *Convoy Effect*, where short process stuck behind long process.

Common Scheduling Issues & Solutions

Issue 1: Starvation

- **Problem:** Low-priority processes may never execute due to continuous arrival of high-priority processes
- **Solution:** Implement *aging* mechanism - processes gradually increase in priority the longer they wait

Issue 2: High Overhead

- **Problem:** Complex scheduling algorithms (e.g., multilevel feedback queues) create significant system overhead
- **Solution:** Optimize queue management with efficient data structures and algorithms

Issue 3: Priority Inversion

- **Problem:** Higher priority process waits while lower priority process holds a needed resource
- **Solution:** Implement *priority inheritance* - lower priority process temporarily inherits the higher priority

Issue 4: Unpredictability

- **Problem:** Execution order can be unpredictable, problematic for real-time systems requiring timing guarantees
- **Solution:** Use deterministic scheduling algorithms or provide explicit timing guarantees for real-time requirements

Issue 5: Parameter Tuning

- **Problem:** Selecting optimal values (time quantum, priority levels) significantly impacts performance
- **Solution:** Systematic testing and performance analysis to determine optimal parameters

Memory Management

- **Logical Address**: virtual address, generated by CPU
- **Physical Address**: absolute address. address sent to memory units.

Binding: logical → physical addr translation

- ***Compile-time***: Phys addr hardcoded. Recompile if program moved.
- ***Load-time***: *Relocatable code* (relative addr). Phys addr calculated when loaded to RAM; address fixed afterwards.
- ***Execution-time (Run-time)***: Bind delayed till run. Process can move during execution. Logical addr used; MMU calc phys addr dynamically. Needs HW support (MMU).

Memory Management Unit (MMU)

- *Relocation Register* (RR): added to every logical addr to get base.
- *Limit Register* (LR): offset limit from the base physical memory

Contiguous Mem. Allocation: Entire process in single block of mem

- Multiple fixed partition: multiprogramming w/ fixed num. of tasks
- Mult. variable partition: multprogramming w/ varying num. of tasks
  - Process arrives → finds **hole** large enough to fit it
  - First-fit: find first hole that's big enough.
  - Best-fit: find smallest hole big enough (little leftovers, not useful)
  - Worst-fit: find largest hole (large but useful leftovers)
  - **First-fit** and **Best-fit** are *normally* better
  - ***External Fragmentation***: space exist, but not contiguous (fixed with *compaction* - stop all process and group memory together)
  - ***Internal Fragmentation***: waste within partition (hole slightly larger than process, and managing this overhead is not worth it)

Non-Cont. Mem. Alloc: multiple blocks (*paging* or *segmentation*)

- ***Paging***: physical mem divided into fixed-size **frames**. Logical mem divided into same fixed-size **pages**. Page table maps page → frames
  - logical addr. = page number + offset
  - *Page number*: index of page table. *Offset*: offset within page.
  - <u>Paging suffers Internal Fragmentation</u>
  - <u>Paging allows **sharing** a program across multiple processes</u>
  - **Implementation**:
    - *page-table base register* (PTBR): points to page table of a process in the memory. **Fast context switch**: just change PTBR.
    - But needs <u>two mem. accesses</u>: 1. get page table, 2. get the data.
    - **Solution**: *Translation Look-aside Buffer* (TLB) - store recently translated logical page numbers.
      - cache access time c, memory access time m, TLB hit ratio h
      - Translation time = $h \times c + (1-h) \times (c+m)$
      - Effective access time = $c + (2-h) \times m$
  - **Hierarchical Paging** (otherwise, page table too large):
    - Example, 32bit machine - 20bit for page number, 12bit for offset.
    - Page the page table: 10 bit page number, 10 bit offset
  - **Memory Protection**:
    - *Valid-Invalid bit*: additional bit to each page table entry
    - *Page-table length register (PTLR)*: check validity of logical addr.
- ***Segmentation***: logical memory space is different-sized segments. Each segment table entry contains *base*, *limit*, *valid bit*
  - Segment-table base register(STBR): point to segment table base in memory. *Segment-table length register (STLR)*: number of segment
  - <u>Segmentation suffers External Fragmentation</u>
  - <u>Segmentation also allows sharing. Put same entry in each table</u>

Virtual Memory Techniques

- ***Overlay***: break program into stages, and load them sequentially
  - swapping but on different **stages** instead of parts of program
  - Memory has a common area for common modules
  - When program > physical memory (eg embedded systems)
- ***Vritual memory***: store process on disk, load on demand.
  - Uses **memory map** (basically page table but for virtual mem)
  - More genereal puposed

Checking memory usage

- `a.out &` runs `a.out` in background. <u>**allow multiple instance**</u>
- Then `ps -l` lists current running process in this terminal.
- Columns: `VSZ` total size. `RSS` mem usage (`VSZ` ≥ `RSS`)

Methods:

- *Demand paging*: load page only when it's needed.
  - Need for a page indicated by PC or MAR.
  - Via lazy process ***swapper***. Swapper that swaps pages = ***pager***.
  - Valid-invalid bit: indicate whether page is in memory
  - **Procedure**: 1. check valid-invalid bit. 2. if invalid, trigger a *trap* called *page fault*. 3. Run *page fault handler*, that ask *pager* to swap
    - Note: valid-invalid bit set to invalid for all entries initially.
    - <u>Page fault handling</u>: 1. get empty frame from free frame list. 2. load page from disk. 3. update page table to point to new frame. 4. set valid-invalid bit to valid. 5. Restart page fault instruction.
  - **Performance**:
    - *page fault rate* = $0 \le f \le 1$,
    - *memory access time* = m,
    - *page fault service time* = s.
    - *Effective Access Time (EAT)*: *expected memory access time*

    page fault service time = page fault overhead + time to swap page in + restart overhead

    $$\text{EAT} = (1-f) \times m + f \times s$$
- *Anticipatory paging*: predict what page will be needed next.

Page Replacement

- *Belady's Anomaly*: page fault rate ↑ if number of page frames ↑
- *Reference String*: page number sequence
- **FIFO**: Always replace oldest page. Suffer from *Belady's Anomaly*.
- **Optimal**: Replace page that will be used furthest in the future.
- **LRU**: Replace page that has not been used for the longest time.
- **Allocation**:
  - *Local page replacement*: process only replaces its own frames
  - *Global page replacement*: process can replace other's frames
  - *Fixed allocation* (<u>**only *local page replacement* allowed**</u>):
    - Equal allocation: each process gets same number of frames ($\frac{F}{n}$)
    - Proportional allocation: each get frames proportion to its size
  - *Variable allocation*: number of frames vary over time. Give processes with too many page faults more frames.

Thrashing: when number of available frames < total size of active pages of the processes in the ready queue.

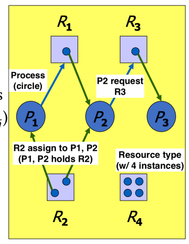
- Solution: **reduce** the degree of multiprogramming.
- *Working-Set*: total number of unique pages reference in a period Δ
- When working-set size > number of frames, thrashing occurs
- ***Page Fault Frequency (PFF) algorithm***: define acceptable range. If rate too low, takeaway some frames. If rate too high, add frames.

Deadlock

- **Deadlock**: set of blocked resources. cannot proceed. **Conditions:**
  - *Mutual Exclusion*: only one process can use a resource at a time
  - *Hold and wait*: process holds ≥ 1 resource, waits another resource
  - *No preemption*: resource only released voluntarily by process
  - *Circular wait*: hold-and-wait processes forms a cycle.
- **Livelock**: can be resolved by chance.

Resource Allocation Graph (RAG)

- $P = \{P_1, P_2, \dots P_n\}$, set of all processes
- $R = \{R_1, R_2, \dots R_m\}$, set of all resources
- **Request Edge**: P_i requests R_j ($P_i \to R_j$)
- **Assignment Edge**: $R_j \to P_i$
- If a req can fulfill, <u>***request edge becomes assignment edge***</u>



Check for deadlock (RULES):

- If no cycle, <u>*definitely no deadlock*</u>
- If cycle, only one instance per resource type, <u>*deadlock*</u>
- If cycle, multiple instances per resource type, <u>*might deadlock*</u>

Deadlock Handling

- *Ostrich approach (Deadlock ignorance)*: ignore the deadlock
- *Deadlock avoidance (Banker's algorithm)*: don't enter deadlock state
- *Deadlock prevention*: ensure never enter deadlock state
- *Deadlock detection*: detect deadlock after happen; recover from it

Deadlock Prevention:

- Ensure when process request resource, it is not holding other resources (<u>in-degree/out-degree cannot be non-zero at the same time</u>)
- Make preemption possible (e.g., by using time quantum)
- Prevent circular wait:
  - Order all resource types with a pre-defined number
  - P_i can only request R_j if j larger than the number of any resource currently held by P_i. Else P_i have to release larger resource first
  - (<u>All edge point from small number R to large. No reverse edges</u>)

Deadlock Avoidance (*Banker's algorithm*):

- *Safe State*: system is safe if exist certain resource allocation order that can avoid deadlock. This order is called ***Safe Sequence***
- $\{P_1, P_2, \dots, P_n\}$ is safe sequence if: for each P_i, its *need* can be satisfied by current available resource + resources held by all $P_{j<i}$

Deadlock Detection (***Wait-for Graph***)

- Detect deadlock by *deadlock detection algorithm*
- Execute *deadlock recovery scheme* to recover from deadlock
  - *Process termination*: kill all deadlocked processes / kill one victim process at a time until deadlock eliminated (consider: priority progress, etc.) Might ***starvation***: some process always victim
  - *Process rollback*: return to safe state and retry

Banker's Algorithm

- Avail $[m]$ # of resources m currently available
- Alloc$_i[m]$ # of resources m currently allocated to process i
- Max$_i[m]$ # of resources m that process i will at most need
- Need$_i[m]$ Max$_i[m]$ − Alloc$_i[m]$ (how much more i needs)

Intuition:

1. Simulate allocate resource to a P_k
2. Assume P_k run in background (async)
3. **If** (synchronized) execution sequence exists then **safe** Else **unsafe**

```
Work ← Avail (array of length m)
Finish ← [False, ...] (array of length n)
while not done do:
    find i s.t. ¬ Finish[i] and Need_i ≤ Work
    if no such i exists, break
    Work ← Work + Alloc_i
    Finish[i] ← True
if Finish[i] = False ∀i then return Safe
else return Unsafe
```

Find all safe sequence: requires drawing search (DAG) graph.

Wait-for Graph

RAG except simplify **FROM** $P_i \to R_j$ and $R_j \to P_i$ **TO** $P_i \to P_j$ Periodically check for cycles in the *Wait-for Graph*

Synchronization

Critical Section (CS) Problem:

- *Critical Section*: code segment where shared data is accessed
- Solution: [*Entry Section*] + [*Critical Section*] + [*Exit Section*]
- **Rules**:
  1. *Mutual Exclusion*: only one process in their CS at a time
  2. *Progress*: When no one in CS, and one process wants to enter CS decision must be made within finite time. (**no hanging request!**)
  3. *Bounded Waiting*: Between request and grant for A, limit the number of times B enters CS. (**fairness / no starvation!**)
- **Solutions**:
  - **Lock Variable**: shared boolean

    ```
    while (lock == true) wait();
    lock = true; Enter_CS(); lock = false;
    ```

    Problem:
    1. Process P_0 checks lock; it's false.
    2. Context switch **before** P_0 can set lock = true.
    3. Process P_1 checks lock; it's still false.
    4. P_1 sets lock = true and enters its critical section.
    5. Context switch back to P_0 (already passed while). Sets lock = true (even though it's already true) and also enters its CS.
    6. Now both P_0 and P_1 are in their CS simultaneously!!!
  - **Test-and-Set (TSL) Lock**: use special hardware instruction, TSL. This instruction is atomic. <u>It is a single, **indivisible** operation</u>

    ```
    while(TestAndSet(&lock) == true) wait();
    Enter_CS(); lock = false;
    ```
  - **Turn Variable**: two processes P_0 and P_1. A shared integer variable **turn** to indicate <u>whose turn it is to enter the critical section</u>

    ```
    while(turn != self) wait();
    Enter_CS(); turn = other;
    ```

Solution	*Mutual Excl.*	*Progress*	*Bounded Wait*
Lock Variable	Fails	Fails	Fails
Test and Set	Satisfied	Satisfied	Fails
Turn Variable	Satisfied	Fails	Satisfied

Peterson's Algorithm

Peterson's Algorithm (flag helps satisfy *Progress*)

```
turn ← 0 // indicate whose turn to enter CS
flag[0] ← flag[1] ← false // whether process ready to enter CS
while true do:
    flag[i] ← true
    turn ← j // Use j to indicate other processes (j ≠ i)
    while flag[j] = true and turn = j do wait() end
    [Critical Section]
    flag[i] ← false
    Remainder Section ...
```

Pros: Builds on top of **Turn Variable** and satisfies *Progress*
Cons: while-loop waste CPU time. The thing it's waiting starved.

Semaphores - Ultimate solution for critical section problem

```
while (true) {
  P(S);
  Enter_CS();
  V(S);
  Remainder_Section();
}
```

S is a semaphore, shared by all processes
P(S): decrease S by 1, if negative move current process to waiting queue
V(S): increase S by 1, if exist processes in waiting queue, wake up one of them

Intuition:

- **S** is a counter of available resources
- **P(S)** check if any resource available. If true, take one (**S** -= 1). Otherwise (**S** ≤ 0), this process sent to waiting queue
- **V(S)**: called when process release a resource (**S** += 1). If there is any process in waiting queue, wake up one of them.
- Both **P(S)** and **V(S)** are <u>atomic operations</u>. Can't context switch.

– **P(S)**, aka: `wait()`, `down()`, `acquire()`
– **V(S)**, aka: `signal()`, `up()`, `release()`

Other Concepts

- *Counting Semaphore*: $[-\infty, +\infty]$, though typically non-negative.
  – For resource that has multiple instances
- *Binary Semaphore* (**Mutex**): $[0, 1]$, Init. to 1, alter between 0 and 1.
  – For resource that has only one instance.

Producer-Consumer Solution (using Semaphores)

```
// producer
while (true)
{
  down(empty);
  down(mutex);
  produce();
  up(mutex);
  up(full);
}
```

```
// consumer
while (true)
{
  down(full);
  down(mutex);
  consume();
  up(mutex);
  up(empty);
}
```

empty: # of empty buffer slots
full: # of occupied slots
mutex: ensure only one party access the buffer at a time

If `wait(mutex)` placed before `wait(empty)`, producer can lock buffer and do nothing (if buffer is full) cause **deadlock**

File System, Secondary Storage

Access Methods

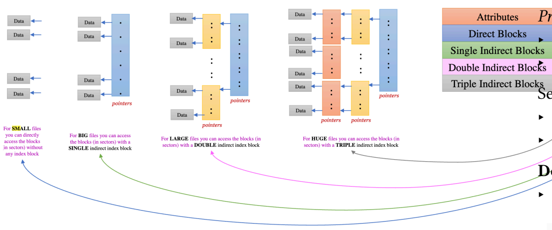
- **Sequential Access**: Data accessed in order, beginning to end
  – **read next**: read next data, advance pointer (`read fp; fp++`)
  – **write next**: write next data, advance pointer (`write fp; fp++`)
  – **reset**: put file pointer back to the beginning (`fp=0`)
  – **skip forward/back**: fp forward/backward without reading.
- **Direct Access**: aka Relative access. Files are fixed-length *records*. Direct access like arrays, but based on *record number*
  – **Read** n: return n-th data item / block
  – **Write** n: update n-th data item / block
- **Indexed Access**: Use a separate *index file* (aka *direct file* or *index file*) contain pointers to data blocks of data file. Can direct access

Allocation Methods

Methods should 1) Efficient disk util. 2) fast access time

- **Contiguous Allocation**: Each file stored as a contiguous block
  – **Pro**: easy to implement, fast access time (sequential / direct)
  – **Con**: Disk fragmented, difficult to grow file
- **Linked List Allocation**: File has many blocks, linked by pointer
  – **Pro**: No external fragmentation. File size can increase
  – **Con**: Large seek time, direct access hard, pointer overhead
- **Indexed Allocation**: index file points to each file's first block
  – **Pro**: No external fragmentation. Supports direct access
  – **Cons**: Pointer (memory) overhead. Multi-level index (if file big)

UNIX I-Node Structure



Disk Scheduling Algorithm

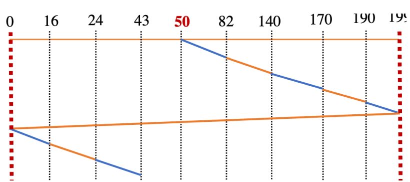
- Goal: Minimize seek time
- Seek time = time taken to reach desired track

FCFS (First Come First Serve)
SSTF (Shortest Seek Time First)

- Greedily choose the closest request from current head position.
- Issue: starvation problem. Overhead calculating seek time.

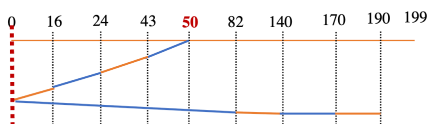
C-SCAN (Circular SCAN)

1. Move from current to the largest track (read along the way)
2. Then immediately back to smallest track (**don't read**)
3. Read from smallest track to last requested track (read along way)



SCAN (assume larger value indicator, otherwise vice versa)

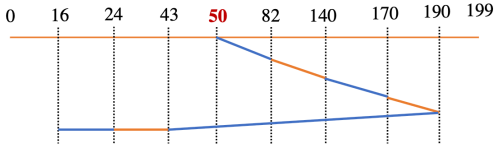
- Minimize seek time using *Elevator algorithm*
  1. Move head to the largest track (while reading along the way)
  2. Then back to the smallest **requested** track (read along way)
- **SCAN better than C-SCAN**



LOOK (assume direction towards large value, if not then vice versa)

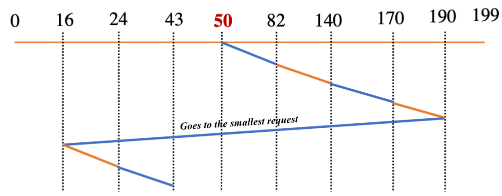
1. Move from current to largest **requested** track (read along way)
2. Move back to smallest **requested** track (read along the way)

- **LOOK better than SCAN**

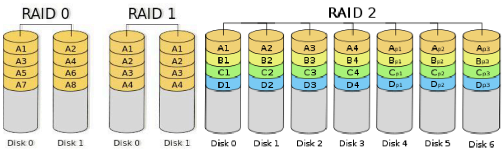


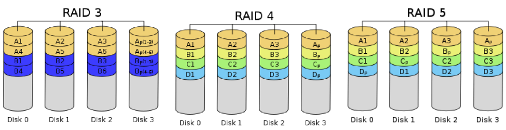
C-LOOK (Circular LOOK)

1. Move from current to largest **requested** track (read along way)
2. Immediately back to smallest **requested** track (don't read)
3. Then read to final requested track (read along way)



Redundant Arrays of Inexpensive Disks (RAID)





Level	Description	Util. (N=#disk)
RAID0	Stripe. No Fault tolerance	$100\%(N/N)$
RAID1	Mirror (store 2 copies)	$50\%(N/2)$
RAID2	Bit-lvl stripe + ECC	Low, $(N-P)N$
RAID3	Byte-lvl stripe. dedicate parity disk	$(N-1)/N$
RAID4	Block-lvl stripe. dedicate parity disk	$(N-1)/N$
RAID5	Block-lvl. distributed parity disk	$(N-1)/N$

P (ECC disks) must satisfy: $2^P \geq N + P + 1$

Protection

Principle of Least Privilege:

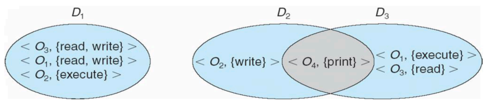
- Programs, user, systems given **just enough** to perform their task
- Limit damage if bug or get abused

Separate *policy* from *mechanism*:

- *Mechanism*: the actual implementation that enforces the *policy*
- *Policy*: customizable security rules realized through *mechanism*

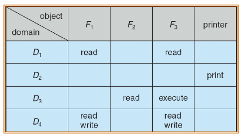
Domain specifies a set of object and their operations

- `Access-right = <object-name, right-set>`



Access Matrix

- Row represent *domains*
- Column represent *objects*
- Access(i, j) tells the set of operation Domain$_i$ can perform on Object$_j$



domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

- Entire table: stored as ***ordered triples***
  – Result: List of triples `<domain, object, right>`
- Each column called ***access control list***
  – Result: per-object list of ordered pairs: `<domain, right>`
- Each row called ***capability list***

```
fill() {
    local size=$1 content=$2 blob=()
    for ((i=0; i<size; i++)); do blob+=("$content"); done
    echo "${blob[@]}"
}

lookup() {
    local mode=$1 query=$2 index_only=$3 keys vals i results=()
    if [[ $mode == "name" ]]; then
        keys=("${emp_nms[@]}")
        vals=("${emp_ids[@]}")
    else
        keys=("${emp_ids[@]}")
        vals=("${emp_nms[@]}")
    fi
    for ((i=0; i<total_emps; i++)); do
        if [[ ${keys[i]} == $query ]]; then
            if (( index_only )); then results+=("$i");
            else results+=("${vals[i]}"); fi
        fi
    done
    echo "${results[@]}"
}

push_unique() {
    local i
    for i in ${queue[@]}; do
        (($i == $1)) && return
    done
    queue+=($1)
}
```

```
check() {
    [ -e "$1" ] && [ -f "$1" ] && [ -r "$1" ] || { echo "Invalid
File: $1"; exit 1; }
}

parse() {
    local flag=0
    for p in "${params[@]}"; do
        [[ $p == "employee" ]] && { flag=1; continue; }
        if (( flag )); then
            emps+=("$p")
        else
            [[ " ${dats[*]} " != *" $p "* ]] && dats+=("$p")
        fi
    done
    n_emps=${#emps[@]}
}

rd() {
    # read employee
    while IFS= read -r line || [[ -n "$line" ]]; do
        read id nm <<< "$line"; emp_ids+=("$id"); emp_nms+=("$nm"
    done < "employees.dat"
    total_emps=${#emp_ids[@]} # total number of employees

    # read data files
    for file in "${dats[@]}"; do
        check "$file" # check file validity
        local firstline=1 index=-1 date=0
        while IFS= read -r line || [[ -n "$line" ]]; do
            if (( firstline )); then
                firstline=0
                 local tmp1 dpt month year; read tmp1 dpt month
year <<< "$line"
                date=$(parse_date "$year" "$month")
                # find the index of the department
                local i
                for (( i=0; i<${#departments[@]}; i++ )); do
                    if [[ ${departments[i]} == $dpt ]]; then
index=$i; break; fi
                done
                # if department not found, add the department nam
                # then expand the attendance status array size
                # then initialize new slots with "N/A"
                if (( index == -1 )); then
                    departments+=("$dpt")
                    status+=($(fill "$total_emps" "N/A"))
                    last_modified+=($(fill "$total_emps" 0))
                    index=$((${#departments[@]} - 1))
                fi
            else
                local id attd; read id attd <<< "$line"
               local i=$(lookup "id" "$id" 1) # look up the index
of ID
                local j=$(( index * total_emps + i ))
                 if (( date >= ${last_modified[j]} )); then
last_modified[j]=$date; status[j]=$attd; fi
            fi
        done < "$file"
    done
}

# echo results
out() {
    # remove redundancy
    for e in "${emps[@]}"; do
          # combine indices from both name and id searches in
one array
        local found_indices=($(lookup "name" "$e" 1) $(lookup "id"
"$e" 1))
        if (( ${#found_indices[@]} == 0 )); then
            echo "Invalid Entry: $e"
            exit 1
        fi
        for index in "${found_indices[@]}"; do
            push_unique "$index"
        done
    done

    # loop through employees' indices
    for ei in "${queue[@]}"; do
        echo "Attendance Report for ${emp_ids[ei]} ${emp_nms[ei]}
        local total_present=0 total_absent=0 total_leave=0

        # loop through known departments
        for di in "${!departments[@]}"; do
            local j=$(( di * total_emps + ei ))
            local department_name=${departments[di]}
            local department_status=${status[j]}

            if [[ $department_status == "N/A" ]]; then
                            echo  "Department  ${department_name}:
${department_status}"
            else
                            echo  "Department  ${department_name}:
${department_status} 1"
            fi

            # increment counters based on status
          [[ $department_status == "Present" ]] && ((total_present
+))
            [[ $department_status == "Absent" ]] && ((total_absen
            [[ $department_status == "Leave" ]] && ((total_leave+
        done
        echo

        # output statistics
        echo "Total Present Days: $total_present"
        echo "Total Absent Days: $total_absent"
        echo "Total Leave Days: $total_leave"
        echo
    done
}

main() {
    parse
    rd
    out
}
```