Operating System Laboratory

Overview

These 20 practical experiments of Computer Operating System progress from fundamental UNIX/Linux interfaces (permissions, files, directories, text scanning) to core process control (fork/exec/wait, signals, pipelines), then to IPC and synchronization (shared memory + semaphore, threads, producer-consumer). The latter half focuses on classic OS simulations used in standard labs: CPU scheduling, deadlocks, memory allocation, paging/translation, page replacement, file allocation, and disk scheduling. This mirrors common coverage found across multiple university OS lab manuals.

Experiment 1: UNIX Permission and umask Calculator

Experiment 2: POSIX File Copy with open/read/write

Experiment 3: Directory Listing and Metadata Report (ls + stat subset)

Experiment 4: grep-lite: Deterministic Text Pattern Search

Experiment 5: Process Spawner and Exit-Status Reporter (fork/exec/wait)

Experiment 6: Signal-Based Timeout Supervisor (sigaction + alarm + kill)

Experiment 7: Pipe-Based Filter Chain (pipe + dup2)

Experiment 8: Shared Memory Counter IPC (shm_open + mmap + sem_open)

Experiment 9: Threaded Deterministic Reducer (pthreads + mutex)

Experiment 10: Bounded Buffer Producer-Consumer with Semaphores (deterministic summary)

Experiment 11: CPU Scheduling Simulator I (FCFS and Non-preemptive SJF)

Experiment 12: CPU Scheduling Simulator II (Round Robin)

Experiment 13: Priority Scheduling Simulator (Non-preemptive with Aging)

Experiment 14: Deadlock Avoidance using Banker's Algorithm

Experiment 15: Deadlock Detection via Wait-For Graph Cycle

Experiment 16: Contiguous Memory Allocation Simulator (First/Best/Worst Fit)

Experiment 17: Paging Address Translation with Optional TLB

Experiment 18: Page Replacement Simulator (FIFO, LRU, OPT)

Experiment 19: File Allocation Strategy Simulator (Contiguous, Linked, Indexed)

Experiment 20: Disk Scheduling Simulator (FCFS, SSTF, SCAN, C-SCAN)

Experiment 1: UNIX Permission and umask Calculator

a) Learning Outcomes
- Convert between octal permission modes and symbolic rwx strings.
- Apply a umask to a requested mode to compute the effective created permission.
- Validate command-line inputs and report errors deterministically.

b) Problem Statement
- Implement a CLI tool named `permcalc`.
- Inputs: `--mode <octal>` (required) and optional `--umask <octal>`.
- `<octal>` must be exactly 4 digits from 0000 to 0777 (leading zero required).
- Compute: `effective_mode = mode & (~umask)` (bitwise), limited to 0777.
- Output exactly two lines on success: (1) `OK: EFFECTIVE <octal>` (2) `OK: SYMBOLIC <rwxrwxrwx>`.
- On any error, output exactly one line using the standard error format and exit non-zero.

c) Context (if applicable)
This task models how `chmod` and process `umask` influence permissions of newly created files/directories.

d) Implementation Requirements: N/A
N/A

e) Behavior & Output Specification (include exact success/failure messages and error format rules)
- Standard error format: `ERROR: {CODE}: {MESSAGE}`.
- Success messages are exactly the `OK:` lines specified above.
- Error codes: `E_USAGE` (missing/extra args), `E_OCTAL` (bad octal), `E_RANGE` (out of 0000-0777).
- No additional whitespace; uppercase keywords exactly as shown.

f) Test Suite (6–10; Valid/Invalid; rationales; edge cases)
Valid

- `--mode 0644 --umask 0022` - Basic file mode with common umask

- `--mode 0777 --umask 0027` - Typical directory request; group/other masked
- `--mode 0000 --umask 0000` - All permissions disabled

Invalid

- `--umask 0022` - Missing required --mode
- `--mode 644 --umask 0022` - Mode not 4-digit octal
- `--mode 0888 --umask 0000` - Digits outside octal range

g) Sample I/O Transcript (2–5; fenced code blocks; non-interactive)

$ permcalc --mode 0644 --umask 0022

OK: EFFECTIVE 0644

OK: SYMBOLIC rw-r--r--

$ permcalc --mode 644 --umask 0022

ERROR: E_OCTAL: mode must be 4-digit octal (0000-0777)

h) Build & Run Instructions: N/A
N/A

i) Hints & Common Pitfalls
- Remember: umask clears bits (AND with bitwise-NOT of umask).
- Treat inputs as base-8; reject anything not exactly four octal digits.
- Mask the result with 0777 to avoid leaking higher bits.
- When generating `rwxrwxrwx`, map each triad consistently (r=4,w=2,x=1).

Experiment 2: POSIX File Copy with open/read/write

a) Learning Outcomes
- Use low-level file descriptor I/O (`open`, `read`, `write`, `close`) correctly.
- Handle partial reads/writes and propagate system-call failures deterministically.
- Produce a verifiable copy summary (bytes and checksum) for testing.

b) Problem Statement
- Implement a CLI tool named `fdcopy`.
- Inputs: `--src <path>` and `--dst <path>` (required), optional `--buf <N>` (1..1048576), optional `--force`.
- `--src -` means read from STDIN.
- Copy the exact byte stream from `src` to `dst` using only file-descriptor I/O.
- Compute CRC32 of bytes copied (IEEE 802.3) and total bytes copied.
- On success output exactly two lines: `OK: COPIED <bytes> BYTES` and `OK: CRC32 <8-hex>`.
- If `dst` exists, fail unless `--force` is provided.
- On any error, output exactly one line using the standard error format and exit non-zero.

c) Context (if applicable)
This is a foundational OS lab exercise for system-call based file I/O and robust error handling.

d) Implementation Requirements: N/A
N/A

e) Behavior & Output Specification (include exact success/failure messages and error format rules)
- Standard error format: `ERROR: {CODE}: {MESSAGE}`.
- Error codes: `E_USAGE`, `E_OPEN_SRC`, `E_OPEN_DST`, `E_EXISTS`, `E_READ`, `E_WRITE`, `E_CLOSE`, `E_RANGE`.
- CRC32 output must be lowercase hex, zero-padded to 8 characters.
- If `--src -` is used, the tool must not attempt to seek; it must stream until EOF.
- Do not print file contents; only the two `OK:` lines or one `ERROR:` line.

f) Test Suite (6–10; Valid/Invalid; rationales; edge cases)
Valid

- `printf 'abc' | fdcopy --src - --dst out/q02.bin` - Known input, small size (3 bytes)
- `printf '' | fdcopy --src - --dst out/empty.bin` - Empty input edge case (0 bytes)
- `printf '123456789' | fdcopy --src - --dst out/nine.bin --buf 1` - Forces many small writes

Invalid

- `fdcopy --dst out/x` - Missing required --src
- `fdcopy --src - --dst out/q02.bin` (run twice without `--force`) - Destination exists
- `fdcopy --src /no/such/file --dst out/x` - Source open failure

g) Sample I/O Transcript (2–5; fenced code blocks; non-interactive)

$ printf 'abc' | fdcopy --src - --dst out/q02.bin

OK: COPIED 3 BYTES

OK: CRC32 352441c2

$ fdcopy --src - --dst out/q02.bin

ERROR: E_EXISTS: destination already exists (use --force)

h) Build & Run Instructions: N/A
N/A

i) Hints & Common Pitfalls

- `write()` may write fewer bytes than requested; loop until the buffer is fully written.
- Do not assume `read()` returns the full buffer size; 0 means EOF.
- Reject buffer sizes outside the allowed range to keep tests consistent.
- Use `O_CREAT|O_EXCL` for safe destination creation; map the EEXIST case to `E_EXISTS`.

Experiment 3: Directory Listing and Metadata Report (ls + stat subset)

a) Learning Outcomes

- Traverse directories using `opendir`, `readdir`, and `closedir`.
- Collect file metadata using `lstat/stat` and classify entry types.
- Generate stable, testable output ordering and formatting.

b) Problem Statement

- Implement a CLI tool named `dirreport`.
- Input: `--path <dir>` (required), optional `--sort name|size` (default `name`).
- For each direct child entry (non-recursive), output one line: `ENTRY <type> <size> <name>`.
- `<type>` must be one of: `F` (regular file), `D` (directory), `L` (symlink), `O` (other).
- After listing, output a summary line: `OK: TOTAL <n> FILES <f> DIRS <d> LINKS <l> OTHER <o>`.
- If `--sort size`, sort by size ascending then name lexicographically.
- On error output one `ERROR:` line and exit non-zero.

c) Context (if applicable)

Students practice directory APIs and metadata, which underpin shells, file browsers, and backup tools.

d) Implementation Requirements: N/A
N/A

e) Behavior & Output Specification (include exact success/failure messages and error format rules)

- Standard error format: `ERROR: {CODE}: {MESSAGE}`.
- Error codes: `E_USAGE`, `E_NOTDIR`, `E_OPEN_DIR`, `E_READ_DIR`, `E_STAT`.
- Output must be deterministic: do not rely on filesystem enumeration order.
- Names must be printed exactly as returned by directory entries (no extra quoting).

f) Test Suite (6–10; Valid/Invalid; rationales; edge cases)
Valid

- `--path fixtures/q03/basic --sort name` - Deterministic name ordering

- `--path fixtures/q03/basic --sort size` - Size ordering with tie-break by name
- `--path fixtures/q03/onlydirs` - All entries are directories

Invalid

- `--path fixtures/q03/missing` - Non-existent path
- `--path fixtures/q03/file.txt` - Path exists but is not a directory
- `--path /root --sort name` - Permission denied reading directory

g) Sample I/O Transcript (2–5; fenced code blocks; non-interactive)

$ dirreport --path fixtures/q03/basic --sort name

ENTRY D 4096 subdir

ENTRY F 12 notes.txt

ENTRY L 0 link_to_notes

OK: TOTAL 3 FILES 1 DIRS 1 LINKS 1 OTHER 0

$ dirreport --path fixtures/q03/file.txt

ERROR: E_NOTDIR: path is not a directory

h) Build & Run Instructions: N/A
N/A

i) Hints & Common Pitfalls
- Use `lstat()` if you need to classify symlinks without following them.
- Exclude `.` and `..` explicitly.
- Do not print absolute paths; tests compare only the base names.
- Sort using a stable comparison to keep output identical across runs.

Experiment 4: grep-lite: Deterministic Text Pattern Search

a) Learning Outcomes
- Implement buffered file reading and line scanning.
- Handle multiple input files and aggregate match counts.
- Produce deterministic outputs suitable for automated grading.

b) Problem Statement
- Implement a CLI tool named `greplite`.
- Inputs: `--pattern <ASCII>` (required), `--files <f1,f2,...>` (required).
- Match is literal substring (case-sensitive) within each line.
- For each match, output a line: `MATCH <file>:<line_no>:<line>` where `<line>` is the original line without trailing newline.
- After all files, output: `OK: MATCHES <k> FILES <n>`.
- If a file cannot be opened, treat as error (do not partially succeed).
- On any error, output one `ERROR:` line and exit non-zero.

c) Context (if applicable)
This emulates a small subset of `grep` and reinforces text processing using OS I/O primitives.

d) Implementation Requirements: N/A
N/A

e) Behavior & Output Specification (include exact success/failure messages and error format rules)
- Standard error format: `ERROR: {CODE}: {MESSAGE}`.
- Error codes: `E_USAGE`, `E_EMPTY_PATTERN`, `E_OPEN`, `E_READ`.
- Line numbers start at 1.
- Output order is file order given in `--files`, then line order.

f) Test Suite (6–10; Valid/Invalid; rationales; edge cases)
Valid

- `--pattern main --files fixtures/q04/a.c` - Single file, multiple matches
- `--pattern TODO --files fixtures/q04/a.c,fixtures/q04/b.c` - Multiple files

- `--pattern xyz --files fixtures/q04/a.c` - No matches still succeeds

Invalid

- `--pattern " --files fixtures/q04/a.c` - Empty pattern rejected
- `--pattern main` - Missing --files
- `--pattern main --files fixtures/q04/missing.c` - Missing file

g) Sample I/O Transcript (2–5; fenced code blocks; non-interactive)

```
$ greplite --pattern TODO --files fixtures/q04/a.c,fixtures/q04/b.c

MATCH fixtures/q04/a.c:3:// TODO: refactor

MATCH fixtures/q04/b.c:1:// TODO: add tests

OK: MATCHES 2 FILES 2
```

```
$ greplite --pattern " --files fixtures/q04/a.c

ERROR: E_EMPTY_PATTERN: pattern must be non-empty
```

h) Build & Run Instructions: N/A
N/A

i) Hints & Common Pitfalls
- Define clearly whether you accept empty pattern; the spec requires rejecting it.
- Be careful to preserve the original line content (excluding only the final newline).
- Do not emit extra summary lines or blank lines.
- If you split the comma-separated file list, reject empty segments (e.g., trailing comma).

Experiment 5: Process Spawner and Exit-Status Reporter (fork/exec/wait)

a) Learning Outcomes
- Create child processes with `fork()` and replace images with `exec*()`.
- Collect termination information using `waitpid()` and interpret status codes.
- Produce consistent reporting for normal exits vs signal terminations.

b) Problem Statement
- Implement a CLI tool named `spawnwait`.
- Inputs: `--cmd <program>` and optional `--args <a1,a2,...>` and optional `--repeat <k>` (default 1).
- Spawn `k` children sequentially (next starts after previous terminates).
- For each child, print one line: `CHILD <i> PID <pid> START` then one line on completion:
- Normal exit: `CHILD <i> PID <pid> EXIT <code>`
- Signal termination: `CHILD <i> PID <pid> SIG <signum>`
- After all, print: `OK: COMPLETED <k>`.
- On any error (fork/exec/wait failures), print one `ERROR:` line and exit non-zero.

c) Context (if applicable)
This is a core process-management lab aligned with standard OS lab manuals emphasizing fork/exec/wait.

d) Implementation Requirements: N/A
N/A

e) Behavior & Output Specification (include exact success/failure messages and error format rules)
- Standard error format: `ERROR: {CODE}: {MESSAGE}`.
- Error codes: `E_USAGE`, `E_FORK`, `E_EXEC`, `E_WAIT`, `E_RANGE`.
- Child index `i` starts at 1.
- The tool must not print timing data or other non-deterministic fields beyond PID (PID varies; accept as variable in grading).

f) Test Suite (6–10; Valid/Invalid; rationales; edge cases)
Valid

- `--cmd /bin/true --repeat 2` - Repeated normal exit
- `--cmd /bin/sh --args -c,exit\ 7` - Non-zero exit code
- `--cmd /bin/sh --args -c,kill\ -9\ $$` - Signal termination

Invalid

- `--repeat 0 --cmd /bin/true` - Repeat must be >=1
- `--cmd /no/such/program` - Exec failure
- `--args a,b,c` - Missing --cmd

g) Sample I/O Transcript (2–5; fenced code blocks; non-interactive)

```
$ spawnwait --cmd /bin/sh --args -c,exit\ 7 --repeat 1

CHILD 1 PID 12345 START

CHILD 1 PID 12345 EXIT 7

OK: COMPLETED 1
```

```
$ spawnwait --cmd /no/such/program --repeat 1

ERROR: E_EXEC: cannot exec program
```

h) Build & Run Instructions: N/A
N/A

i) Hints & Common Pitfalls

- Interpret `waitpid()` status using standard macros (exit vs signal).
- When `exec` fails in the child, ensure the child exits with a known code so the parent can detect it.
- Validate `--repeat` early to avoid partial execution.
- Do not run children concurrently; sequential spawning keeps output deterministic.

Experiment 6: Signal-Based Timeout Supervisor (sigaction + alarm + kill)

a) Learning Outcomes
- Install signal handlers using `sigaction()` with well-defined semantics.
- Implement a watchdog that terminates a child after a timeout.
- Report outcomes deterministically with clear success/failure messages.

b) Problem Statement
- Implement a CLI tool named `timeoutwrap`.
- Inputs: `--seconds <t>` (required, integer 1..60) and `--cmd <program>` with optional `--args <a1,a2,...>`.
- Behavior: fork a child that execs the command. Parent arms an alarm for `t` seconds.
- If the child exits before timeout, cancel alarm and print: `OK: EXIT <code>`.
- If timeout occurs first, send SIGKILL to the child, wait for it, and print: `OK: TIMEOUT KILLED`.
- On any error, print one `ERROR:` line and exit non-zero.

c) Context (if applicable)
This models how service managers enforce time limits and introduces signal handling in a controlled setting.

d) Implementation Requirements: N/A
N/A

e) Behavior & Output Specification (include exact success/failure messages and error format rules)
- Standard error format: `ERROR: {CODE}: {MESSAGE}`.
- Error codes: `E_USAGE`, `E_RANGE`, `E_FORK`, `E_EXEC`, `E_WAIT`, `E_SIGNAL`.
- Only one `OK:` line on success; never print timestamps.
- If the command is terminated by a signal before timeout, print: `OK: SIG <signum>`.

f) Test Suite (6–10; Valid/Invalid; rationales; edge cases)
Valid

- `--seconds 2 --cmd /bin/sh --args -c,sleep\ 1;\ exit\ 0` - Completes before timeout

- `--seconds 1 --cmd /bin/sh --args -c,sleep\ 5` - Times out
- `--seconds 2 --cmd /bin/sh --args -c,kill\ -2\ $$` - Child self-terminates by signal

Invalid

- `--seconds 0 --cmd /bin/true` - Seconds below range
- `--seconds 61 --cmd /bin/true` - Seconds above range
- `--seconds 2` - Missing --cmd

g) Sample I/O Transcript (2–5; fenced code blocks; non-interactive)

$ timeoutwrap --seconds 1 --cmd /bin/sh --args -c,sleep\ 5

OK: TIMEOUT KILLED

$ timeoutwrap --seconds 0 --cmd /bin/true

ERROR: E_RANGE: seconds must be in 1..60

h) Build & Run Instructions: N/A
N/A

i) Hints & Common Pitfalls
- Prefer `sigaction()` over `signal()` for predictable behavior.
- Avoid race conditions: handle the case where the child exits just as the alarm fires.
- Always `waitpid()` after killing to prevent zombies.
- Validate numeric inputs strictly to keep grading deterministic.

Experiment 7: Pipe-Based Filter Chain (pipe + dup2)

a) Learning Outcomes
- Create anonymous pipes and connect them to standard streams via `dup2()`.
- Launch a multi-process pipeline with `fork()` and `exec()`.
- Detect and report failures in any stage deterministically.

b) Problem Statement
- Implement a CLI tool named `pipechain`.
- Inputs: `--producer <cmd1>` `--filter <cmd2>` `--consumer <cmd3>` (all required).
- Each `<cmd>` is a single shell-free command path with optional comma-separated args via `--producer-args`, `--filter-args`, `--consumer-args`.
- Run the equivalent of: `cmd1 | cmd2 | cmd3` using two pipes and three child processes.
- Parent waits for all children; if all exit 0, print: `OK: PIPELINE SUCCESS`.
- If any stage exits non-zero, print: `ERROR: E_STAGE: stage <name> exit <code>` (exactly one line).
- If a stage is terminated by a signal, print: `ERROR: E_STAGE: stage <name> sig <signum>`.

c) Context (if applicable)
Pipelines are a canonical OS abstraction that combine processes with IPC via pipes.

d) Implementation Requirements: N/A
N/A

e) Behavior & Output Specification (include exact success/failure messages and error format rules)
- Success output is exactly `OK: PIPELINE SUCCESS`.
- On failure, output exactly one `ERROR:` line as specified (note: this question uses `E_STAGE` with embedded details).
- Stage names are exactly: `producer`, `filter`, `consumer`.
- Deterministic output rule: redirect STDOUT and STDERR of all three stages to /dev/null so only pipechain's single status line is printed.

f) Test Suite (6–10; Valid/Invalid; rationales; edge cases)
Valid

- `--producer /bin/echo --producer-args hello --filter /usr/bin/tr --filter-args a-z,A-Z --consumer /usr/bin/wc --consumer-args -c` - Standard 3-stage pipeline
- `--producer /bin/echo --producer-args a --filter /bin/cat --consumer /usr/bin/wc --consumer-args -l` - Deterministic 1-line output through a no-op filter
- `--producer /bin/true --filter /bin/cat --consumer /bin/true` - All stages succeed without output

Invalid

- `--producer /no/such --filter /bin/cat --consumer /bin/true` - Producer exec failure
- `--producer /bin/sh --producer-args -c,exit\ 2 --filter /bin/cat --consumer /bin/true` - Non-zero producer exit
- `--producer /bin/true --filter /bin/cat` - Missing consumer

g) Sample I/O Transcript (2–5; fenced code blocks; non-interactive)

$ pipechain --producer /bin/true --filter /bin/cat --consumer /bin/true

OK: PIPELINE SUCCESS

$ pipechain --producer /bin/sh --producer-args -c,exit\ 2 --filter /bin/cat --consumer /bin/true

ERROR: E_STAGE: stage producer exit 2

h) Build & Run Instructions: N/A
N/A

i) Hints & Common Pitfalls
- Close unused pipe ends in each process to avoid deadlocks due to open writers.
- Connect stdin/stdout with `dup2()` before `exec`.
- Wait for all children and report the first failing stage in fixed order: producer, filter, consumer.
- Avoid invoking a shell; parse args explicitly to keep behavior deterministic.

Experiment 8: Shared Memory Counter IPC (shm_open + mmap + sem_open)

a) Learning Outcomes
- Create and map a POSIX shared memory object.
- Synchronize cross-process updates using a named semaphore.
- Validate that concurrent increments produce an exact final value.

b) Problem Statement
- Implement a CLI tool named `shmcounter`.
- Inputs: `--procs <p>` (2..16), `--iters <n>` (1..100000), `--name <id>` (alphanumeric, 1..16).
- Create shared memory object `/shm_<id>` containing a 64-bit signed integer counter initialized to 0.
- Create named semaphore `/sem_<id>` initialized to 1.
- Fork `p` child processes; each performs `n` increments of the shared counter with semaphore protection.
- After all children exit, output exactly: `OK: FINAL <value>` where `<value>=p*n`.
- Always unlink shared memory and semaphore before exit (success or failure).
- On error output one `ERROR:` line and exit non-zero.

c) Context (if applicable)
This is a standard IPC lab theme (shared memory + synchronization) found in many OS lab manuals.

d) Implementation Requirements: N/A
N/A

e) Behavior & Output Specification (include exact success/failure messages and error format rules)
- Standard error format: `ERROR: {CODE}: {MESSAGE}`.
- Error codes: `E_USAGE`, `E_RANGE`, `E_SHM`, `E_MMAP`, `E_SEM`, `E_FORK`, `E_WAIT`.
- No per-process logging; only the single final `OK:` line on success.
- `<value>` must be printed as a base-10 integer.

f) Test Suite (6–10; Valid/Invalid; rationales; edge cases)
Valid

- `--procs 2 --iters 1000 --name t1` - Small concurrency
- `--procs 8 --iters 1 --name t2` - Many processes, minimal work
- `--procs 16 --iters 100000 --name t3` - Upper-range stress test

Invalid

- `--procs 1 --iters 10 --name t1` - Processes below range
- `--procs 2 --iters 0 --name t1` - Iters below range
- `--procs 2 --iters 10 --name 'bad-name'` - Name must be alphanumeric only

g) Sample I/O Transcript (2–5; fenced code blocks; non-interactive)

$ shmcounter --procs 4 --iters 250 --name demo

OK: FINAL 1000

$ shmcounter --procs 1 --iters 10 --name demo

ERROR: E_RANGE: procs must be in 2..16

h) Build & Run Instructions: N/A
N/A

i) Hints & Common Pitfalls
- Use `ftruncate()` to size the shared memory before `mmap()`.
- Initialize the counter only once in the parent before forking.
- Ensure cleanup on all exit paths (including after partial creation).
- Print only the final value to avoid non-deterministic interleaving.

Experiment 9: Threaded Deterministic Reducer (pthreads + mutex)

a) Learning Outcomes
- Create and join POSIX threads with a fixed work partition.
- Protect shared aggregation with a mutex to avoid data races.
- Produce deterministic results independent of thread scheduling.

b) Problem Statement
- Implement a CLI tool named `thrsum`.
- Inputs: `--threads <t>` (1..32) and `--n <N>` (1..1000000).
- Compute the sum of integers 1..N using `t` threads.
- Work partition must be deterministic: thread i handles a contiguous block of the range.
- Each thread computes a local sum and then adds to a shared total under a mutex.
- Output exactly one line: `OK: SUM <value>`.
- On error output one `ERROR:` line and exit non-zero.

c) Context (if applicable)
This introduces threads and synchronization without requiring non-deterministic interleaved logging.

d) Implementation Requirements: N/A
N/A

e) Behavior & Output Specification (include exact success/failure messages and error format rules)
- Standard error format: `ERROR: {CODE}: {MESSAGE}`.
- Error codes: `E_USAGE`, `E_RANGE`, `E_THREAD`, `E_MUTEX`.
- Sum must be computed using 64-bit arithmetic to avoid overflow for N up to 1,000,000.
- No additional lines or timing output.

f) Test Suite (6–10; Valid/Invalid; rationales; edge cases)
Valid

- `--threads 1 --n 10` - Single-thread baseline
- `--threads 4 --n 100` - Multi-thread correct aggregation

- `--threads 32 --n 1000000` - Upper-range stress

Invalid

- `--threads 0 --n 10` - Threads below range
- `--threads 2 --n 0` - N below range
- `--threads 3` - Missing N

g) Sample I/O Transcript (2–5; fenced code blocks; non-interactive)

```
$ thrsum --threads 4 --n 10

OK: SUM 55
```

```
$ thrsum --threads 0 --n 10

ERROR: E_RANGE: threads must be in 1..32
```

h) Build & Run Instructions: N/A
N/A

i) Hints & Common Pitfalls
- Partition [1..N] carefully so every integer is included exactly once.
- Use `long long`/`int64_t` for totals.
- Lock only during the shared update; keep most work thread-local.
- Validate inputs before creating threads to avoid partial execution.

Experiment 10: Bounded Buffer Producer-Consumer with Semaphores (deterministic summary)

a) Learning Outcomes
- Implement a bounded buffer using semaphores and a mutex.
- Coordinate multiple producer and consumer threads safely.
- Demonstrate correctness via deterministic invariants and summary output.

b) Problem Statement
- Implement a CLI tool named `pcbuf`.
- Inputs: `--buf <B>` (1..1024), `--producers <p>` (1..16), `--consumers <c>` (1..16), `--items <m>` (1..100000).
- Total items produced must equal `m` and total items consumed must equal `m`.
- Each produced item is the integer sequence 1..m (assigned in increasing order by a protected counter).
- Consumers compute the sum of consumed values; after all threads join, output exactly:
- `OK: PRODUCED <m>`
- `OK: CONSUMED <m>`
- `OK: SUM <S>` where `S = m*(m+1)/2`.
- On error, output one `ERROR:` line and exit non-zero.

c) Context (if applicable)
Producer-consumer with semaphores is a canonical OS synchronization lab exercise.

d) Implementation Requirements: N/A
N/A

e) Behavior & Output Specification (include exact success/failure messages and error format rules)
- Standard error format: `ERROR: {CODE}: {MESSAGE}`.
- Error codes: `E_USAGE`, `E_RANGE`, `E_THREAD`, `E_SEM`, `E_MUTEX`.
- Do not print per-item logs; only the three final `OK:` lines.
- If any invariant fails internally, print `ERROR: E_INVARIANT: produced/consumed mismatch` and exit non-zero.

f) Test Suite (6–10; Valid/Invalid; rationales; edge cases)
Valid

- `--buf 4 --producers 1 --consumers 1 --items 10` - Small baseline
- `--buf 8 --producers 2 --consumers 2 --items 100` - Multiple producers/consumers
- `--buf 1 --producers 4 --consumers 4 --items 1000` - Buffer size 1 stresses synchronization

Invalid

- `--buf 0 --producers 1 --consumers 1 --items 10` - Buffer below range
- `--buf 8 --producers 0 --consumers 1 --items 10` - Producers below range
- `--buf 8 --producers 1 --consumers 1 --items 0` - Items below range

g) Sample I/O Transcript (2–5; fenced code blocks; non-interactive)

$ pcbuf --buf 4 --producers 2 --consumers 2 --items 10

OK: PRODUCED 10

OK: CONSUMED 10

OK: SUM 55

$ pcbuf --buf 0 --producers 1 --consumers 1 --items 10

ERROR: E_RANGE: buf must be in 1..1024

h) Build & Run Instructions: N/A
N/A

i) Hints & Common Pitfalls
- Use `empty` and `full` counting semaphores plus a mutex for buffer access.
- Assign item IDs under a mutex to ensure exactly 1..m are produced.
- Keep output deterministic by printing only the final summary.
- Be careful to terminate consumers: use a shared consumed count or sentinel strategy.

Experiment 11: CPU Scheduling Simulator I (FCFS and Non-preemptive SJF)

a) Learning Outcomes
- Parse a process set with arrival and burst times.
- Simulate FCFS and non-preemptive SJF scheduling deterministically.
- Compute waiting time and turnaround time correctly.

b) Problem Statement
- Implement a CLI tool named `schedsim1`.
- Input is provided via stdin as CSV with header: `pid,arrival,burst` (pid is a string without commas).
- Simulate FCFS and non-preemptive SJF (choose shortest burst among arrived; tie-break by arrival then pid).
- For each algorithm, output exactly:
- `ALG <name>` on its own line (name is `FCFS` or `SJF`).
- One Gantt line: `GANTT <pid1>@<t0>-<t1> <pid2>@<t1>-<t2> ...`.
- Summary line: `OK: AVG_WAIT <w> AVG_TAT <t>` with averages as decimals rounded to 2 places.
- On any parse/validation error, output one `ERROR:` line and exit non-zero.

c) Context (if applicable)
CPU scheduling simulations are standard OS lab themes and appear frequently in university lab manuals.

d) Implementation Requirements: N/A
N/A

e) Behavior & Output Specification (include exact success/failure messages and error format rules)
- Standard error format: `ERROR: {CODE}: {MESSAGE}`.
- Error codes: `E_INPUT` (CSV malformed), `E_RANGE` (negative times), `E_DUPPID`.
- Time is integer; CPU is idle if no job has arrived: represent idle as `IDLE@t0-t1` in the Gantt line.
- All numeric outputs use base-10; averages formatted with exactly 2 digits after decimal.

f) Test Suite (6–10; Valid/Invalid; rationales; edge cases)

Valid

- `3 processes, distinct arrivals` - Basic scheduling
- `Includes idle gap` - Tests IDLE segments
- `Tie bursts and arrivals` - Tests tie-breaking

Invalid

- `Missing header` - Reject unknown format
- `Negative burst` - Range validation
- `Duplicate pid` - Uniqueness required

g) Sample I/O Transcript (2–5; fenced code blocks; non-interactive)

```
$ printf 'pid,arrival,burst

P1,0,5

P2,2,2

P3,4,1

' | schedsim1

ALG FCFS

GANTT P1@0-5 P2@5-7 P3@7-8

OK: AVG_WAIT 2.00 AVG_TAT 4.67

ALG SJF

GANTT P1@0-5 P3@5-6 P2@6-8

OK: AVG_WAIT 1.67 AVG_TAT 4.33
```

```
$ printf 'pid,arrival,burst

P1,0,-1

' | schedsim1
```

ERROR: E_RANGE: arrival and burst must be non-negative; burst must be > 0

h) Build & Run Instructions: N/A
N/A

i) Hints & Common Pitfalls
- Treat burst time 0 as invalid to avoid zero-length intervals.
- SJF is non-preemptive here: once scheduled, a process runs to completion.
- Define tie-break rules explicitly and apply them consistently.
- Round averages after computing in floating point; format with two decimals.

Experiment 12: CPU Scheduling Simulator II (Round Robin)

a) Learning Outcomes
- Simulate preemptive Round Robin with a fixed quantum.
- Maintain a ready queue deterministically with clear enqueue rules.
- Compute average waiting/turnaround time for RR.

b) Problem Statement
- Implement a CLI tool named `schedsim2`.
- Input via stdin as CSV header: `pid,arrival,burst`.
- Arguments: `--q <quantum>` (integer 1..1000).
- Simulate Round Robin with these rules:
- Newly arrived processes are enqueued at the end at their arrival time.
- When a time slice ends and the running process is not finished, it is enqueued at the end.
- If CPU becomes idle, time jumps to next arrival.
- Output exactly:
- `ALG RR`
- `GANTT ...` using segments `<pid>@<t0>-<t1>` (include `IDLE` segments if any).
- `OK: AVG_WAIT <w> AVG_TAT <t>` with 2 decimal places.
- On error, output one `ERROR:` line and exit non-zero.

c) Context (if applicable)
RR is widely taught and practiced in OS labs for understanding preemption and ready-queue behavior.

d) Implementation Requirements: N/A
N/A

e) Behavior & Output Specification (include exact success/failure messages and error format rules)
- Standard error format: `ERROR: {CODE}: {MESSAGE}`.
- Error codes: `E_INPUT`, `E_RANGE`, `E_DUPPID`.
- Tie-breaking for multiple arrivals at same time: order by pid lexicographically.
- Do not compress adjacent segments of the same pid; keep time slices explicit.

f) Test Suite (6–10; Valid/Invalid; rationales; edge cases)

Valid

- `Quantum smaller than bursts` - Forces multiple quanta per process
- `Quantum larger than all bursts` - Reduces to FCFS-like behavior
- `Simultaneous arrivals` - Tests tie-breaking on arrival

Invalid

- `Quantum 0` - Range check
- `Malformed CSV` - Parser check
- `Burst 0` - Invalid burst

g) Sample I/O Transcript (2–5; fenced code blocks; non-interactive)

```
$ printf 'pid,arrival,burst
P1,0,5
P2,0,3
' | schedsim2 --q 2
ALG RR
GANTT P1@0-2 P2@2-4 P1@4-6 P2@6-7 P1@7-8
OK: AVG_WAIT 3.00 AVG_TAT 7.00
```

```
$ printf 'pid,arrival,burst
P1,0,5
' | schedsim2 --q 0
ERROR: E_RANGE: quantum must be in 1..1000
```

h) Build & Run Instructions: N/A
N/A

i) Hints & Common Pitfalls
- Be explicit about queue update order at time boundaries (arrivals vs re-queue).
- Use a consistent tie-breaker for arrivals at the same time.
- Compute waiting time via `turnaround - burst` if you compute completion times correctly.
- Avoid floating drift; format with two decimals deterministically.

Experiment 13: Priority Scheduling Simulator (Non-preemptive with Aging)

a) Learning Outcomes
- Simulate non-preemptive priority scheduling with arrival times.
- Apply aging to prevent starvation in a testable manner.
- Compute scheduling metrics and provide a deterministic Gantt chart.

b) Problem Statement
- Implement a CLI tool named `schedprio`.
- Input via stdin as CSV header: `pid,arrival,burst,priority` where smaller `priority` means higher priority.
- Non-preemptive execution: once started, a job runs to completion.
- Aging rule: for every 1 unit of waiting time in the ready queue, effective priority decreases by 1 (down to minimum 0).
- At each dispatch decision, select the ready process with lowest effective priority; tie-break by arrival then pid.
- Output exactly:
- `ALG PRIO_AGING`
- `GANTT ...`
- `OK: AVG_WAIT <w> AVG_TAT <t>` (2 decimals).
- On error, output one `ERROR:` line and exit non-zero.

c) Context (if applicable)
This extends basic priority scheduling with a simple, measurable anti-starvation mechanism.

d) Implementation Requirements: N/A
N/A

e) Behavior & Output Specification (include exact success/failure messages and error format rules)
- Standard error format: `ERROR: {CODE}: {MESSAGE}`.
- Error codes: `E_INPUT`, `E_RANGE`, `E_DUPPID`.
- Priority must be integer 0..99; burst must be > 0; arrival >= 0.
- Represent idle gaps as `IDLE@t0-t1` in Gantt.

f) Test Suite (6–10; Valid/Invalid; rationales; edge cases)

Valid

- `Higher priority arrives later` - Checks dispatch decisions
- `Starvation scenario mitigated by aging` - Validates aging rule
- `Idle gap before first arrival` - IDLE formatting

Invalid

- `Priority -1` - Range validation
- `Non-integer priority` - Input validation
- `Missing column` - CSV schema validation

g) Sample I/O Transcript (2–5; fenced code blocks; non-interactive)

```
$ printf 'pid,arrival,burst,priority

A,0,4,5

B,1,2,0

' | schedprio

ALG PRIO_AGING

GANTT A@0-4 B@4-6

OK: AVG_WAIT 1.50 AVG_TAT 4.50
```

```
$ printf 'pid,arrival,burst,priority

A,0,4,-1

' | schedprio

ERROR: E_RANGE: priority must be in 0..99
```

h) Build & Run Instructions: N/A
N/A

i) Hints & Common Pitfalls

- Define when aging is applied (at dispatch points) and apply it consistently.
- Keep priorities bounded at 0 to avoid negative values.
- Avoid per-time-unit simulation unless needed; compute waiting time analytically if possible.
- Clearly separate base priority from effective priority in your calculations.

Experiment 14: Deadlock Avoidance using Banker's Algorithm

a) Learning Outcomes
- Model resource allocation with Allocation, Max, and Available vectors.
- Apply Banker's safety algorithm to find a safe sequence.
- Report SAFE/UNSAFE deterministically with precise formatting.

b) Problem Statement
- Implement a CLI tool named `banker`.
- Input via stdin in this exact format:
- First line: `P R` (integers: processes, resource types)
- Next P lines: Allocation matrix (R ints each)
- Next P lines: Max matrix (R ints each)
- Last line: Available vector (R ints).
- Validate that Allocation <= Max for every entry.
- Run the safety check; if safe, output exactly:
- `OK: SAFE`
- `OK: SEQ <p0> <p1> ... <p(P-1)>` (process indices 0..P-1).
- If unsafe, output exactly one line: `OK: UNSAFE`.
- On error, output one `ERROR:` line and exit non-zero.

c) Context (if applicable)
Banker's algorithm is a standard lab experiment for deadlock avoidance and resource-allocation reasoning.

d) Implementation Requirements: N/A
N/A

e) Behavior & Output Specification (include exact success/failure messages and error format rules)
- Standard error format: `ERROR: {CODE}: {MESSAGE}`.
- Error codes: `E_INPUT` (wrong counts/format), `E_RANGE` (negative), `E_INVALID` (alloc>max).
- If multiple safe sequences exist, output the lexicographically smallest sequence of process indices.

- No intermediate printing; only the specified `OK:` lines.

f) Test Suite (6–10; Valid/Invalid; rationales; edge cases)

Valid

- `Small safe instance (P=3,R=2)` - Produces SAFE + sequence
- `Instance with multiple safe sequences` - Tests lexicographically smallest requirement
- `Edge: P=1` - Trivial safe/unsafe behavior

Invalid

- `Allocation greater than Max` - Reject invalid matrices
- `Negative entry` - Range check
- `Wrong number of columns` - Schema validation

g) Sample I/O Transcript (2–5; fenced code blocks; non-interactive)

$ printf '3 2

1 0

0 1

1 1

2 0

1 2

1 1

1 1

' | banker

OK: SAFE

OK: SEQ 1 0 2

$ printf '2 1

1

0

0

0

0

' | banker

ERROR: E_INVALID: allocation must be <= max

h) Build & Run Instructions: N/A
N/A

i) Hints & Common Pitfalls
- Compute Need = Max - Allocation and verify it is non-negative.
- To guarantee lexicographically smallest sequence, always choose the smallest-index runnable process at each step.
- Be strict about counts: P lines and R integers per line.
- Keep all arithmetic in integers; no floats required.

Experiment 15: Deadlock Detection via Wait-For Graph Cycle

a) Learning Outcomes
- Model process waiting relationships as a directed graph.
- Detect cycles deterministically using DFS or Kahn-style methods.
- Report deadlock presence and one canonical cycle.

b) Problem Statement
- Implement a CLI tool named `wfgcheck`.
- Input via stdin as:
- First line: `P E` (process count, edge count)
- Next E lines: `u v` meaning u waits for v (directed u->v), processes are 0..P-1.
- Output exactly one of:
- No deadlock: `OK: DEADLOCK NO`
- Deadlock: two lines `OK: DEADLOCK YES` and `OK: CYCLE <p0> <p1> ... <pk> <p0>`.
- If multiple cycles exist, output the cycle with the smallest starting node; if ties, smallest lexicographic sequence.
- On error, output one `ERROR:` line and exit non-zero.

c) Context (if applicable)
This models OS deadlock detection as used with single-instance resources and wait-for graphs.

d) Implementation Requirements: N/A
N/A

e) Behavior & Output Specification (include exact success/failure messages and error format rules)
- Standard error format: `ERROR: {CODE}: {MESSAGE}`.
- Error codes: `E_INPUT`, `E_RANGE`.
- Self-loop (u->u) counts as a deadlock cycle of length 1.
- Do not print adjacency lists or debugging output.

f) Test Suite (6–10; Valid/Invalid; rationales; edge cases)
Valid

- `Acyclic graph` - Should report DEADLOCK NO
- `Simple 2-node cycle` - Detect and print canonical cycle
- `Self-loop` - Cycle length 1

Invalid

- `Edge uses out-of-range node` - Range check
- `Negative P/E` - Invalid counts
- `Malformed line` - Input validation

g) Sample I/O Transcript (2–5; fenced code blocks; non-interactive)

$ printf '3 2

0 1

1 2

' | wfgcheck

OK: DEADLOCK NO

$ printf '3 3

0 1

1 2

2 1

' | wfgcheck

OK: DEADLOCK YES

OK: CYCLE 1 2 1

h) Build & Run Instructions: N/A
N/A

i) Hints & Common Pitfalls

- Canonical cycle reporting requires consistent tie-breaking; decide it up-front.
- For deterministic output, sort adjacency lists by node number.
- Handle self-loops explicitly.
- Validate that E matches the number of edge lines actually provided.

Experiment 16: Contiguous Memory Allocation Simulator (First/Best/Worst Fit)

a) Learning Outcomes
- Model memory as a list of free holes/blocks.
- Apply first-fit, best-fit, and worst-fit placement policies.
- Report allocation results and failures deterministically.

b) Problem Statement
- Implement a CLI tool named `memfit`.
- Input via stdin in this exact format:
- Line 1: `B` number of blocks
- Line 2: B integers: block sizes
- Line 3: `P` number of processes
- Line 4: P integers: process sizes.
- Simulate three algorithms independently using the original block list each time: FIRST_FIT, BEST_FIT, WORST_FIT.
- For each algorithm output:
- `ALG <name>`
- One line per process i (0-based): `PROC <i> SIZE <s> -> BLOCK <j>` or `PROC <i> SIZE <s> -> FAIL`.
- Summary line: `OK: ALLOCATED <k>/<P>`.
- On error, output one `ERROR:` line and exit non-zero.

c) Context (if applicable)
This aligns with common OS lab exercises on contiguous allocation strategies.

d) Implementation Requirements: N/A
N/A

e) Behavior & Output Specification (include exact success/failure messages and error format rules)
- Standard error format: `ERROR: {CODE}: {MESSAGE}`.
- Error codes: `E_INPUT`, `E_RANGE`.
- If multiple blocks qualify, tie-break by smallest block index.

- Blocks shrink after allocation (remaining size stays available in the same block index).

f) Test Suite (6–10; Valid/Invalid; rationales; edge cases)

Valid

- `Classic example with mixed sizes` - Demonstrates different placements
- `All processes fit` - No FAIL outputs
- `No process fits` - All FAIL outputs

Invalid

- `Negative block size` - Range check
- `B=0` - Invalid counts
- `Non-integer token` - Parser strictness

g) Sample I/O Transcript (2–5; fenced code blocks; non-interactive)

```
$ printf '3

10 20 5

4

6 8 21 5

' | memfit

ALG FIRST_FIT

PROC 0 SIZE 6 -> BLOCK 0

PROC 1 SIZE 8 -> BLOCK 1

PROC 2 SIZE 21 -> FAIL

PROC 3 SIZE 5 -> BLOCK 1

OK: ALLOCATED 3/4

ALG BEST_FIT

PROC 0 SIZE 6 -> BLOCK 0

PROC 1 SIZE 8 -> BLOCK 1
```

PROC 2 SIZE 21 -> FAIL

PROC 3 SIZE 5 -> BLOCK 2

OK: ALLOCATED 3/4

ALG WORST_FIT

PROC 0 SIZE 6 -> BLOCK 1

PROC 1 SIZE 8 -> BLOCK 1

PROC 2 SIZE 21 -> FAIL

PROC 3 SIZE 5 -> BLOCK 0

OK: ALLOCATED 3/4

$ printf '0

1

5

' | memfit

ERROR: E_RANGE: B and P must be positive

h) Build & Run Instructions: N/A
N/A

i) Hints & Common Pitfalls
- Make a fresh copy of the block list for each algorithm.
- Tie-breakers matter for deterministic grading; specify and implement them.
- Shrinking blocks should not change their index.
- Reject zero-sized blocks or processes unless explicitly allowed (spec assumes positive sizes).

Experiment 17: Paging Address Translation with Optional TLB

a) Learning Outcomes
- Translate virtual addresses using page number and offset.
- Detect and report page faults deterministically.
- Model a small direct-mapped TLB and count hits/misses.

b) Problem Statement
- Implement a CLI tool named `pagetrans`.
- Arguments: `--pagesize <S>` (power of 2, 256..65536), `--tlb <K>` (0..64).
- Input via stdin format:
- Line 1: `N` number of page table entries
- Next N lines: `vpn pfn valid` (valid is 0 or 1)
- Next line: `Q` number of queries
- Next Q lines: `vaddr` (unsigned decimal).
- For each query, output exactly one line:
- If valid mapping: `OK: VA <vaddr> -> PA <paddr> (TLB HIT|TLB MISS)` when K>0, else omit TLB part.
- If page fault: `OK: VA <vaddr> -> PAGEFAULT`.
- After all queries, if K>0 output: `OK: TLB_HITS <h> TLB_MISSES <m>`.
- On error, output one `ERROR:` line and exit non-zero.

c) Context (if applicable)
This reinforces basic paging translation and illustrates the role of the TLB cache.

d) Implementation Requirements: N/A
N/A

e) Behavior & Output Specification (include exact success/failure messages and error format rules)
- Standard error format: `ERROR: {CODE}: {MESSAGE}`.
- Error codes: `E_INPUT`, `E_RANGE`.
- TLB is direct-mapped with index `vpn % K`; store (vpn,pfn) entries for valid translations only.
- If K=0, do not print any TLB HIT/MISS text or final TLB counters.

f) Test Suite (6–10; Valid/Invalid; rationales; edge cases)

Valid

- `K=0, all pages valid` - No TLB text
- `K=4, repeated address` - Shows HIT after first MISS
- `Includes invalid entry` - Page fault output

Invalid

- `Page size not power of 2` - Reject invalid pagesize
- `Valid flag not 0/1` - Input validation
- `Negative vaddr token` - Range check

g) Sample I/O Transcript (2–5; fenced code blocks; non-interactive)

```
$ printf '2

0 5 1

1 9 0

3

0

300

700

' | pagetrans --pagesize 256 --tlb 0

OK: VA 0 -> PA 1280

OK: VA 300 -> PA 1580

OK: VA 700 -> PAGEFAULT
```

```
$ printf '1

0 1 1
```

2

10

10

' | pagetrans --pagesize 256 --tlb 4

OK: VA 10 -> PA 266 (TLB MISS)

OK: VA 10 -> PA 266 (TLB HIT)

OK: TLB_HITS 1 TLB_MISSES 1

h) Build & Run Instructions: N/A
N/A

i) Hints & Common Pitfalls
- Compute vpn = vaddr // pagesize and offset = vaddr % pagesize.
- Validate that page table contains at most one entry per vpn (or define overwrite behavior if duplicates appear; prefer reject).
- Keep all arithmetic unsigned where possible.
- Ensure TLB updates occur only on successful translations.

Experiment 18: Page Replacement Simulator (FIFO, LRU, OPT)

a) Learning Outcomes
- Simulate classic page replacement strategies over a reference string.
- Track frame contents deterministically and count page faults.
- Provide reproducible traces suitable for grading.

b) Problem Statement
- Implement a CLI tool named `pagerepl`.
- Arguments: `--frames <F>` (1..64).
- Input via stdin as: first line `L` (length), second line `L` integers (page numbers >=0).
- Simulate FIFO, LRU, and OPT (Belady optimal) separately.
- For each algorithm output:
- `ALG <name>`
- `OK: FAULTS <k>`
- `OK: FINAL <f0> <f1> ... <f(F-1)>` where empty frames are printed as `-1`.
- On error, output one `ERROR:` line and exit non-zero.

c) Context (if applicable)
Page replacement is a central OS memory-management topic frequently used as a lab simulation exercise.

d) Implementation Requirements: N/A
N/A

e) Behavior & Output Specification (include exact success/failure messages and error format rules)
- Standard error format: `ERROR: {CODE}: {MESSAGE}`.
- Error codes: `E_INPUT`, `E_RANGE`.
- LRU tie-break: if multiple candidates are equally least-recently-used, evict the lowest frame index.
- OPT tie-break: if multiple pages are never used again, evict the lowest frame index.

f) Test Suite (6–10; Valid/Invalid; rationales; edge cases)

Valid

- `Small reference string` - Manual verification
- `All same page` - Only first access faults
- `Frames=1` - Degenerate case

Invalid

- `Frames 0` - Range check
- `Negative page number` - Invalid reference
- `Length mismatch` - Input validation

g) Sample I/O Transcript (2–5; fenced code blocks; non-interactive)

$ printf '12

1 2 3 2 4 1 2 5 2 1 2 3

' | pagerepl --frames 3

ALG FIFO

OK: FAULTS 9

OK: FINAL 2 1 3

ALG LRU

OK: FAULTS 8

OK: FINAL 2 1 3

ALG OPT

OK: FAULTS 7

OK: FINAL 2 1 3

$ printf '2

1 -1

' | pagerepl --frames 2

ERROR: E_RANGE: page numbers must be >= 0

h) Build & Run Instructions: N/A
N/A

i) Hints & Common Pitfalls
- Represent empty frames as -1 exactly as specified.
- For OPT, you may need look-ahead; ensure tie-breakers are deterministic.
- Keep per-algorithm state isolated; do not reuse mutated structures across algorithms.
- Validate that L matches the number of integers on the reference line.

Experiment 19: File Allocation Strategy Simulator (Contiguous, Linked, Indexed)

a) Learning Outcomes
- Model disk blocks and free space management at an abstract level.
- Apply three classic file allocation strategies.
- Report allocation maps and failures deterministically.

b) Problem Statement
- Implement a CLI tool named `filealloc`.
- Input via stdin:
- Line 1: `N` total blocks (1..10000)
- Line 2: `F` number of free blocks
- Line 3: F integers listing free block IDs (0..N-1, unique)
- Line 4: `M` number of files
- Next M lines: `name size` where size is number of blocks needed (1..N).
- Simulate three strategies independently using the original free list each time:
- CONTIGUOUS: allocate `size` consecutive blocks.
- LINKED: allocate any `size` blocks; represent as a chain.
- INDEXED: allocate 1 index block + `size` data blocks (total size+1 blocks).
- Output per algorithm:
- `ALG <name>`
- For each file: `FILE <name> -> <map>` or `FILE <name> -> FAIL`.
- Map formats:

- CONTIGUOUS: `START <b> LEN <size>`

- LINKED: `CHAIN <b1>-><b2>->...`

- INDEXED: `INDEX <i> DATA <b1>,<b2>,...`.

- On error, output one `ERROR:` line and exit non-zero.

c) Context (if applicable)
Many OS lab manuals include file allocation simulations as a standard file-system topic.

d) Implementation Requirements: N/A
N/A

e) Behavior & Output Specification (include exact success/failure messages and error format rules)

- Standard error format: `ERROR: {CODE}: {MESSAGE}`.
- Error codes: `E_INPUT`, `E_RANGE`, `E_DUPBLOCK`.
- Allocation policy for choosing blocks must be deterministic: always choose the smallest available block IDs; for contiguous, choose the smallest starting block that fits.
- If a file fails, it must not consume any blocks for that algorithm.

f) Test Suite (6–10; Valid/Invalid; rationales; edge cases)

Valid

- `Enough space for all files` - All allocations succeed
- `Contiguous fails but linked succeeds` - Demonstrates fragmentation effects
- `Indexed overhead makes file fail` - Tests index-block accounting

Invalid

- `Free list contains duplicates` - Reject invalid free list
- `Block ID out of range` - Range check
- `File size 0` - Invalid file size

g) Sample I/O Transcript (2–5; fenced code blocks; non-interactive)

$ printf '10

6

0 1 2 5 6 7

2

A 2

B 3

' | filealloc

ALG CONTIGUOUS

FILE A -> START 0 LEN 2

FILE B -> START 5 LEN 3

ALG LINKED

FILE A -> CHAIN 0->1

FILE B -> CHAIN 2->5->6

ALG INDEXED

FILE A -> INDEX 0 DATA 1,2

FILE B -> FAIL

$ printf '5

3

0 0 1

1

A 1

' | filealloc

ERROR: E_DUPBLOCK: free block list must contain unique IDs

h) Build & Run Instructions: N/A
N/A

i) Hints & Common Pitfalls
- Keep the original free list and simulate each strategy on a fresh copy.
- For contiguous allocation, scan for the smallest run of consecutive free blocks.
- For indexed allocation, remember the extra index block requirement.
- Do not partially allocate a file; allocation is all-or-nothing per file.

Experiment 20: Disk Scheduling Simulator (FCFS, SSTF, SCAN, C-SCAN)

a) Learning Outcomes
- Model disk head movement and request servicing order.
- Simulate classic disk scheduling algorithms deterministically.
- Compute total head movement and service sequence.

b) Problem Statement
- Implement a CLI tool named `disksched`.
- Arguments: `--max <C>` (max cylinder, integer >= 1), `--start <S>` (0..C), `--dir left|right` (for SCAN/C-SCAN).
- Input via stdin: first line `L` then second line `L` integers of requested cylinders (0..C).
- Simulate FCFS, SSTF, SCAN, and C-SCAN.
- For each algorithm output:
- `ALG <name>`
- `OK: ORDER <c1> <c2> ... <cL>` (service order, duplicates preserved),
- `OK: MOVES <m>` (total head movement as integer).
- On error, output one `ERROR:` line and exit non-zero.

c) Context (if applicable)
Disk scheduling is a standard OS storage-management lab topic alongside paging and deadlocks.

d) Implementation Requirements: N/A
N/A

e) Behavior & Output Specification (include exact success/failure messages and error format rules)
- Standard error format: `ERROR: {CODE}: {MESSAGE}`.
- Error codes: `E_INPUT`, `E_RANGE`, `E_DIR`.
- SSTF tie-break: if two requests are equally near, choose the smaller cylinder number.
- SCAN: move in the given direction, servicing requests in order, go to end cylinder (0 or C) then reverse once.
- C-SCAN: move in the given direction to end cylinder, jump to the other end (count the jump as movement), then continue.

f) Test Suite (6–10; Valid/Invalid; rationales; edge cases)

Valid

- `Mixed requests both sides of start` - Exercises all algorithms
- `All requests on one side` - SCAN/C-SCAN edge behavior
- `Duplicate requests` - Preserve duplicates in service order

Invalid

- `Request out of range` - Range check
- `Bad dir value` - Dir validation
- `L mismatch` - Input validation

g) Sample I/O Transcript (2–5; fenced code blocks; non-interactive)

```
$ printf '5

55 58 39 18 90

' | disksched --max 199 --start 50 --dir right

ALG FCFS

OK: ORDER 55 58 39 18 90

OK: MOVES 132

ALG SSTF

OK: ORDER 55 58 39 18 90

OK: MOVES 120

ALG SCAN

OK: ORDER 55 58 90 39 18

OK: MOVES 330

ALG C-SCAN

OK: ORDER 55 58 90 18 39

OK: MOVES 388
```

$ printf '1

200

' | disksched --max 199 --start 50 --dir left

ERROR: E_RANGE: request cylinders must be in 0..max

h) Build & Run Instructions: N/A
N/A

i) Hints & Common Pitfalls
- Define precisely whether SCAN/C-SCAN go to the end cylinder even if no request there; the spec requires going to end.
- Count head movement as absolute differences between consecutive head positions, including the C-SCAN jump.
- Keep duplicates as separate requests; do not deduplicate.
- Implement tie-breakers to keep SSTF deterministic.

Textbook & References
   • Avi Silberschatz, Peter B. Galvin, Greg Gagne. *Operating System Concepts* (9th ed. used in this course; 10th ed. also acceptable).
   • Andrew S. Tanenbaum, Herbert Bos. *Modern Operating Systems*.
   • Michael Kerrisk. *The Linux Programming Interface*; and the Linux man-pages project (man7.org).
   • Selected OS lab syllabi/manuals consulted to calibrate standard experiment themes (system calls, scheduling, memory, deadlocks, IPC, disk, file allocation).