Report Submission

Student ID: 23701011

Student Full Name: Nilanjana Das Jui

Academic Semester: 5th Semester

Session: 2022-2023

Batch: 23

Email: nilanjana.csecu@gmail.com

Department name: Computer Science and Engineering

Faculty Name: Engineering

Hall: Bijoy 24

University name: University of Chittagong

Course Code: CSE - 512

Course Title: Operating System Lab

Course Credits: 2

Date of Submission: February 05, 2026

Contents

Experiment 1: UNIX Permission and umask Calculator

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:
  - `OK: EFFECTIVE <octal>`
  - `OK: SYMBOLIC <rwxrwxrwx>`
- On any error, output exactly one line using the standard error format and exit non-zero.

Solution Implementation

```python
import sys
import re

# Error codes
E_USAGE = "E_USAGE"
E_OCTAL = "E_OCTAL"
E_RANGE = "E_RANGE"

def error_exit(code, message):
    """Print error message and exit with non-zero status."""
    print(f"ERROR: {code}: {message}", file=sys.stderr)
    sys.exit(1)

def parse_arguments(args):
```

```
15      """Parse command line arguments."""
16      mode = None
17      umask = "0000"  # Default umask
18
19      i = 0
20      while i < len(args):
21          if args[i] == "--mode":
22              if i + 1 >= len(args):
23                  error_exit(E_USAGE, "missing value for --mode")
24              mode = args[i + 1]
25              i += 2
26          elif args[i] == "--umask":
27              if i + 1 >= len(args):
28                  error_exit(E_USAGE, "missing value for --umask")
29              umask = args[i + 1]
30              i += 2
31          else:
32              error_exit(E_USAGE, f"unrecognized argument: {args[i]}")
33
34      if mode is None:
35          error_exit(E_USAGE, "missing required --mode")
36
37      return mode, umask
38
39  def validate_octal(value, name):
40      """Validate that value is exactly 4 octal digits (0000-0777)."""
41      # Check length
42      if len(value) != 4:
43          return False, f"{name} must be exactly 4 digits"
44
45      # Check each character is octal digit
46      if not re.match(r'^[0-7]{4}$', value):
47          return False, f"{name} must be 4-digit octal (0000-0777)"
48
49      # Convert to integer and check range
50      try:
51          int_val = int(value, 8)
52          if int_val < 0 or int_val > 0o777:
```

```
            return False, f"{name} must be in range 0000-0777"
    except ValueError:
        return False, f"{name} must be 4-digit octal (0000-0777)"

    return True, None

def octal_to_symbolic(octal_str):
    """Convert 4-digit octal string to symbolic rwxrwxrwx format."""
    # We only care about last 3 digits (permissions)
    # First digit is special permissions (sticky bit, setuid, setgid)
    perm_str = octal_str[-3:] if len(octal_str) == 4 else octal_str

    symbolic = ""
    for digit in perm_str:
        oct_val = int(digit)

        # Build rwx for this digit
        r = 'r' if oct_val & 4 else '-'
        w = 'w' if oct_val & 2 else '-'
        x = 'x' if oct_val & 1 else '-'

        symbolic += r + w + x

    return symbolic

def calculate_effective(mode_str, umask_str):
    """Calculate effective mode = mode & (~umask)."""
    mode_int = int(mode_str, 8)
    umask_int = int(umask_str, 8)

    # Apply umask: effective = mode AND (NOT umask)
    # Mask with 0o777 to ensure we only get 9 permission bits
    effective_int = mode_int & (~umask_int) & 0o777

    # Format back to 4-digit octal with leading zeros
    return f"{effective_int:04o}"

def main():
```

```
    # Parse arguments (skip program name)
    mode_str, umask_str = parse_arguments(sys.argv[1:])

    # Validate inputs
    valid, msg = validate_octal(mode_str, "mode")
    if not valid:
        error_exit(E_OCTAL, msg)

    valid, msg = validate_octal(umask_str, "umask")
    if not valid:
        error_exit(E_OCTAL, msg)

    # Calculate effective mode
    effective_str = calculate_effective(mode_str, umask_str)

    # Generate symbolic representation
    symbolic_str = octal_to_symbolic(effective_str)

    # Output results
    print(f"OK: EFFECTIVE {effective_str}")
    print(f"OK: SYMBOLIC {symbolic_str}")

    sys.exit(0)

if __name__ == "__main__":
    main()
```

Sample Input 1

```
$ python3 permcalc.py --mode 0644 --umask 0022
```

Sample Output 1

```
OK: EFFECTIVE 0644
OK: SYMBOLIC rw-r--r--
```

Sample Input 2

```
$ python3 permcalc.py --mode 0744 --umask 0022
```

Sample Output 2

```
OK: EFFECTIVE 0744
OK: SYMBOLIC rwxr--r--
```

Sample Input 3

```
$ python3 permcalc.py --mode 0844 --umask 0020
```

Sample Output 3

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

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

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`
  - `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.

Solution Implementation

```
1
2  import sys
3  import os
4  import zlib
5
6  # Error codes
7  E_USAGE = "E_USAGE"
8  E_OPEN_SRC = "E_OPEN_SRC"
9  E_OPEN_DST = "E_OPEN_DST"
10 E_EXISTS = "E_EXISTS"
11 E_READ = "E_READ"
```

```
E_WRITE = "E_WRITE"
E_CLOSE = "E_CLOSE"
E_RANGE = "E_RANGE"

def error_exit(code, message):
    """Print error message and exit with non-zero status."""
    print(f"ERROR: {code}: {message}", file=sys.stderr)
    sys.exit(1)

def parse_arguments(args):
    """Parse command line arguments."""
    src = None
    dst = None
    buffer_size = 4096  # Default buffer size
    force = False

    i = 0
    while i < len(args):
        if args[i] == "--src":
            if i + 1 >= len(args):
                error_exit(E_USAGE, "missing value for --src")
            src = args[i + 1]
            i += 2
        elif args[i] == "--dst":
            if i + 1 >= len(args):
                error_exit(E_USAGE, "missing value for --dst")
            dst = args[i + 1]
            i += 2
        elif args[i] == "--buf":
            if i + 1 >= len(args):
                error_exit(E_USAGE, "missing value for --buf")
            try:
                buffer_size = int(args[i + 1])
            except ValueError:
                error_exit(E_USAGE, "buffer size must be an integer")
            i += 2
        elif args[i] == "--force":
            force = True
```

```
            i += 1
        else:
            error_exit(E_USAGE, f"unrecognized argument: {args[i]}")

    if src is None:
        error_exit(E_USAGE, "missing required --src")
    if dst is None:
        error_exit(E_USAGE, "missing required --dst")

    # Validate buffer size
    if buffer_size < 1 or buffer_size > 1048576:  # 1..1048576 bytes
        error_exit(E_RANGE, "buffer size must be 1..1048576 bytes")

    return src, dst, buffer_size, force

def open_source(src_path):
    """Open source file or stdin."""
    if src_path == "-":
        # Use stdin (file descriptor 0)
        return 0  # STDIN_FILENO
    else:
        try:
            # O_RDONLY: read only, no create
            fd = os.open(src_path, os.O_RDONLY)
            return fd
        except OSError as e:
            error_exit(E_OPEN_SRC, f"cannot open source: {e.strerror}")

def open_destination(dst_path, force):
    """Open destination file with appropriate flags."""
    # Default flags: write only, create, exclusive (fail if exists)
    flags = os.O_WRONLY | os.O_CREAT

    if not force:
        flags |= os.O_EXCL  # Fail if file exists

    # Default permissions: rw-r--r-- (0644)
    mode = 0o644
```

```
88
89      try:
90          fd = os.open(dst_path, flags, mode)
91          return fd
92      except OSError as e:
93          if e.errno == 17:  # EEXIST: File exists
94              error_exit(E_EXISTS, "destination already exists (use --
                    force)")
95          else:
96              error_exit(E_OPEN_DST, f"cannot open destination: {e.
                    strerror}")
97
98  def copy_file(fd_src, fd_dst, buffer_size):
99      """
100     Copy from fd_src to fd_dst using buffer_size chunks.
101     Returns (bytes_copied, crc32_value)
102     """
103     bytes_copied = 0
104     crc32 = 0
105
106     while True:
107         # Read chunk from source
108         try:
109             chunk = os.read(fd_src, buffer_size)
110         except OSError as e:
111             error_exit(E_READ, f"read error: {e.strerror}")
112
113         # EOF reached
114         if not chunk:
115             break
116
117         # Write chunk to destination (handle partial writes)
118         bytes_written = 0
119         while bytes_written < len(chunk):
120             try:
121                 written = os.write(fd_dst, chunk[bytes_written:])
122             except OSError as e:
123                 error_exit(E_WRITE, f"write error: {e.strerror}")
```

```
124
125             if written == 0:
126                 error_exit(E_WRITE, "write returned 0 bytes")
127
128             bytes_written += written
129
130         # Update CRC32 and byte count
131         crc32 = zlib.crc32(chunk, crc32)
132         bytes_copied += len(chunk)
133
134     return bytes_copied, crc32
135
136 def close_file(fd, fd_name):
137     """Close file descriptor with error handling."""
138     if fd > 0:  # Don't close stdin (fd=0) if we didn't open it
139         try:
140             os.close(fd)
141         except OSError as e:
142             error_exit(E_CLOSE, f"cannot close {fd_name}: {e.strerror}"
                )
143
144 def main():
145     # Parse arguments
146     src_path, dst_path, buffer_size, force = parse_arguments(sys.argv
        [1:])
147
148     # Open source
149     fd_src = open_source(src_path)
150
151     # Open destination
152     fd_dst = open_destination(dst_path, force)
153
154     try:
155         # Copy data
156         bytes_copied, crc32_value = copy_file(fd_src, fd_dst,
            buffer_size)
157
158         # Ensure CRC32 is 32-bit unsigned
```

```
159         crc32_value = crc32_value & 0xffffffff
160
161         # Output results
162         print(f"OK: COPIED {bytes_copied} BYTES")
163         print(f"OK: CRC32 {crc32_value:08x}")
164
165     finally:
166         # Always close files
167         close_file(fd_src, "source")
168         close_file(fd_dst, "destination")
169
170     sys.exit(0)
171
172 if __name__ == "__main__":
173     main()
```

Sample Input 1

```
$ printf "abc" | python3 fdcopy.py --src - --dst out/test.bin
```

Sample Output 1

```
ERROR: E_OPEN_DST: cannot open destination: No such file or
directory
```

Sample Input 2

```
$ mkdir -p out
$ printf "abc" | python3 fdcopy.py --src - --dst out/test.bin
```

Sample Output 2

```
OK: COPIED 3 BYTES
OK: CRC32 352441c2
```

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

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 exactly one `ERROR:` line and exit non-zero.

Solution Implementation

```
1
2  import sys
3  import os
4  import stat
5
6  # Error codes
7  E_USAGE = "E_USAGE"
8  E_NOTDIR = "E_NOTDIR"
9  E_OPEN_DIR = "E_OPEN_DIR"
10 E_READ_DIR = "E_READ_DIR"
11 E_STAT = "E_STAT"
12
13 def error_exit(code, message):
```

```
    """Print error message and exit with non-zero status."""
    print(f"ERROR: {code}: {message}", file=sys.stderr)
    sys.exit(1)

def parse_arguments(args):
    """Parse command line arguments."""
    path = None
    sort_by = "name"  # default

    i = 0
    while i < len(args):
        if args[i] == "--path":
            if i + 1 >= len(args):
                error_exit(E_USAGE, "missing value for --path")
            path = args[i + 1]
            i += 2
        elif args[i] == "--sort":
            if i + 1 >= len(args):
                error_exit(E_USAGE, "missing value for --sort")
            sort_by = args[i + 1]
            if sort_by not in ["name", "size"]:
                error_exit(E_USAGE, "sort must be 'name' or 'size'")
            i += 2
        else:
            error_exit(E_USAGE, f"unrecognized argument: {args[i]}")

    if path is None:
        error_exit(E_USAGE, "missing required --path")

    return path, sort_by

def get_entry_type(mode):
    """Determine entry type character from stat mode."""
    if stat.S_ISREG(mode):
        return 'F'  # Regular file
    elif stat.S_ISDIR(mode):
        return 'D'  # Directory
    elif stat.S_ISLNK(mode):
```

```
52          return 'L'  # Symbolic link
53      else:
54          return 'O'  # Other (device, pipe, socket, etc.)
55
56  def list_directory(path, sort_by):
57      """
58      List directory entries and return list of (name, type, size) tuples
            .
59      """
60      entries = []
61
62      # Check if path exists and is a directory
63      try:
64          if not os.path.exists(path):
65              error_exit(E_NOTDIR, "path does not exist")
66
67          if not os.path.isdir(path):
68              error_exit(E_NOTDIR, "path is not a directory")
69      except OSError as e:
70          error_exit(E_NOTDIR, f"cannot access path: {e.strerror}")
71
72      # Open and read directory
73      try:
74          # Using scandir() which is more efficient than listdir()
75          with os.scandir(path) as it:
76              for entry in it:
77                  name = entry.name
78
79                  # Skip . and ..
80                  if name in ['.', '..']:
81                      continue
82
83                  # Get file info using lstat() (doesn't follow symlinks)
84                  try:
85                      stat_info = entry.stat(follow_symlinks=False)
86                  except OSError as e:
87                      error_exit(E_STAT, f"cannot stat '{name}': {e.
                            strerror}")
```

```
88
89                  # Determine type
90                  entry_type = get_entry_type(stat_info.st_mode)
91
92                  # Get size
93                  size = stat_info.st_size
94
95                  entries.append((name, entry_type, size))
96
97      except PermissionError as e:
98          error_exit(E_OPEN_DIR, f"permission denied: {e.strerror}")
99      except OSError as e:
100         error_exit(E_OPEN_DIR, f"cannot open directory: {e.strerror}")
101
102     # Sort entries
103     if sort_by == "name":
104         entries.sort(key=lambda x: x[0])  # Sort by name
105     elif sort_by == "size":
106         # Sort by size, then by name for ties
107         entries.sort(key=lambda x: (x[2], x[0]))
108
109     return entries
110
111 def main():
112     # Parse arguments
113     path, sort_by = parse_arguments(sys.argv[1:])
114
115     # Get directory entries
116     entries = list_directory(path, sort_by)
117
118     # Counters for summary
119     total = len(entries)
120     files = 0
121     dirs = 0
122     links = 0
123     other = 0
124
125     # Output each entry
```

```
    for name, entry_type, size in entries:
        print(f"ENTRY {entry_type} {size} {name}")

        # Update counters
        if entry_type == 'F':
            files += 1
        elif entry_type == 'D':
            dirs += 1
        elif entry_type == 'L':
            links += 1
        elif entry_type == 'O':
            other += 1

    # Output summary
    print(f"OK: TOTAL {total} FILES {files} DIRS {dirs} LINKS {links}
       OTHER {other}")

    sys.exit(0)

if __name__ == "__main__":
    main()
```

Sample Input 1: Directory does not exist

```
$ python3 dirreport.py --path testdir --sort size
```

Sample Output 1

```
ERROR: E_NOTDIR: path does not exist
```

Commands to create test directory and files

```
cd LAB/
mkdir testdir
echo "Hello" > testdir/file1.txt
echo "World" > testdir/file2.txt
mkdir testdir/subdir
ln -s file1.txt testdir/link1
```

Sample Input 2: List directory sorted by size after creation

```
$ python3 dirreport.py --path testdir --sort size
```

Sample Output 2

```
ENTRY F 6 file1.txt
ENTRY F 6 file2.txt
ENTRY L 9 link1
ENTRY D 4096 subdir
OK: TOTAL 4 FILES 2 DIRS 1 LINKS 1 OTHER 0
```

Sample Input 3: Directory **anotherdir**

```
$ mkdir anotherdir
$ echo "Data" > anotherdir/data1.txt
$ echo "MoreData" > anotherdir/data2.txt
$ mkdir anotherdir/subfolder
$ ln -s data1.txt anotherdir/linkA
$ python3 dirreport.py --path anotherdir --sort size
```

Sample Output 3

```
ENTRY F 5 data1.txt
ENTRY F 9 data2.txt
ENTRY L 6 linkA
ENTRY D 4096 subfolder
OK: TOTAL 4 FILES 2 DIRS 1 LINKS 1 OTHER 0
```

Sample Input 4: Directory **missingdir** (does not exist)

```
$ python3 dirreport.py --path missingdir
```

Sample Output 4

```
ERROR: E_NOTDIR: path does not exist
```

Experiment 4: grep-lite: Deterministic Text Pattern Search

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 processing all files, output exactly one summary line: `OK: MATCHES <k> FILES <n>` (`<k>` = total matches, `<n>` = number of files processed).
- If any file cannot be opened, treat it as an error (do not partially succeed).
- On any error, output exactly one `ERROR:` line and exit non-zero.

Solution Implementation

```
import sys

# Error codes
E_USAGE = "E_USAGE"
E_EMPTY_PATTERN = "E_EMPTY_PATTERN"
E_OPEN = "E_OPEN"
E_READ = "E_READ"

def error_exit(code, message):
    """Print error message and exit with non-zero status."""
    print(f"ERROR: {code}: {message}", file=sys.stderr)
    sys.exit(1)

```

```
14 def parse_arguments(args):
15     """Parse command line arguments."""
16     pattern = None
17     files_str = None
18 
19     i = 0
20     while i < len(args):
21         if args[i] == "--pattern":
22             if i + 1 >= len(args):
23                 error_exit(E_USAGE, "missing value for --pattern")
24             pattern = args[i + 1]
25             i += 2
26         elif args[i] == "--files":
27             if i + 1 >= len(args):
28                 error_exit(E_USAGE, "missing value for --files")
29             files_str = args[i + 1]
30             i += 2
31         else:
32             error_exit(E_USAGE, f"unrecognized argument: {args[i]}")
33 
34     if pattern is None:
35         error_exit(E_USAGE, "missing required --pattern")
36     if files_str is None:
37         error_exit(E_USAGE, "missing required --files")
38 
39     # Validate pattern
40     if pattern == "":
41         error_exit(E_EMPTY_PATTERN, "pattern must be non-empty")
42 
43     # Parse file list
44     if files_str.endswith(','):
45         error_exit(E_USAGE, "invalid file list: trailing comma")
46 
47     files = [f.strip() for f in files_str.split(',') if f.strip()]
48 
49     if not files:
50         error_exit(E_USAGE, "must provide at least one file")
51 
```

```
52      # Check for empty filenames in the middle
53      if '' in [f.strip() for f in files_str.split(',')]:
54          error_exit(E_USAGE, "file list contains empty entries")
55
56      return pattern, files
57
58  def search_files(pattern, files):
59      """Search for pattern in files and return results."""
60      matches = []
61      total_matches = 0
62      files_processed = 0
63
64      # Try to open all files first (all-or-nothing)
65      file_handles = []
66      try:
67          for filename in files:
68              try:
69                  f = open(filename, 'r')
70                  file_handles.append((filename, f))
71              except OSError as e:
72                  # Close any files we successfully opened
73                  for _, f in file_handles:
74                      f.close()
75                  error_exit(E_OPEN, f"cannot open '{filename}': {e.
                       strerror}")
76      except Exception as e:
77          # Catch any other unexpected errors
78          for _, f in file_handles:
79              f.close()
80          error_exit(E_OPEN, f"unexpected error opening files: {e}")
81
82      # Now search each file
83      for filename, f in file_handles:
84          try:
85              line_number = 0
86              file_match_count = 0
87
88              for line in f:
```

```
89                  line_number += 1
90                  # Remove trailing newline but keep other whitespace
91                  line_content = line.rstrip('\n')
92
93                  # Check for pattern (case-sensitive substring)
94                  if pattern in line_content:
95                      matches.append(f"MATCH {filename}:{line_number}:{
                            line_content}")
96                      total_matches += 1
97                      file_match_count += 1
98
99              files_processed += 1
100
101         except OSError as e:
102             # Close all files before exiting
103             for _, f in file_handles:
104                 f.close()
105             error_exit(E_READ, f"error reading '{filename}': {e.
                    strerror}")
106         finally:
107             f.close()
108
109     return matches, total_matches, files_processed
110
111 def main():
112     # Parse arguments
113     pattern, files = parse_arguments(sys.argv[1:])
114
115     # Search files
116     matches, total_matches, files_processed = search_files(pattern,
            files)
117
118     # Output matches
119     for match in matches:
120         print(match)
121
122     # Output summary
123     print(f"OK: MATCHES {total_matches} FILES {files_processed}")
```

```
124
125     sys.exit(0)
126
127 if __name__ == "__main__":
128     main()
```

Commands to create test files

```
cd "LAB"
echo "abc def" > test1.txt
echo "hello" > test2.txt
```

Sample Input 1: Files not yet created

```
$ python3 4.greplite.py --pattern "xyz" --files "test1.txt,
test2.txt"
```

Sample Output 1: Files missing

```
ERROR: E_OPEN: cannot open 'test1.txt': No such file or
directory
```

Sample Input 2: After creating test files

```
$ python3 4.greplite.py --pattern "xyz" --files "test1.txt,
test2.txt"
```

Sample Output 2: No matches

```
OK: MATCHES 0 FILES 2
```

Sample Input 3: Pattern ”hello”

```
$ python3 4.greplite.py --pattern "hello" --files "test1.txt,
test2.txt"
```

Sample Output 3: One match

```
MATCH test2.txt:1:hello
OK: MATCHES 1 FILES 2
```

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

Problem Statement

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

Solution Implementation

```
import sys
import os

def error_exit(code, message):
    print(f"ERROR: {code}: {message}", file=sys.stderr)
    sys.exit(1)

def main():
    # 1. Parse Arguments
    args = sys.argv[1:]
    cmd = None
```

```
    cmd_args = []
    repeat = 1

    i = 0
    while i < len(args):
        if args[i] == "--cmd":
            if i + 1 < len(args):
                cmd = args[i+1]
                i += 2
            else:
                error_exit("E_USAGE", "missing value for --cmd")
        elif args[i] == "--args":
            if i + 1 < len(args):
                # Split comma-separated arguments
                cmd_args = args[i+1].split(',')
                i += 2
            else:
                error_exit("E_USAGE", "missing value for --args")
        elif args[i] == "--repeat":
            if i + 1 < len(args):
                try:
                    repeat = int(args[i+1])
                    if repeat < 1: raise ValueError
                except ValueError:
                    error_exit("E_RANGE", "repeat must be >= 1")
                i += 2
            else:
                error_exit("E_USAGE", "missing value for --repeat")
        else:
            error_exit("E_USAGE", f"unrecognized argument: {args[i]}")

    if cmd is None:
        error_exit("E_USAGE", "missing required --cmd")

    # 2. Sequential Spawning Loop
    for k in range(1, repeat + 1):
        try:
            # Step 1: Fork
```

```
50              pid = os.fork()
51          except OSError:
52              error_exit("E_FORK", "failed to fork process")
53
54          if pid == 0:
55              # --- CHILD PROCESS ---
56              try:
57                  # Step 2: Exec
58                  # os.execvp(file, args_list) - first arg must be the
                        program name
59                  os.execvp(cmd, [cmd] + cmd_args)
60              except OSError:
61                  # If exec fails, the child must exit immediately
62                  # We exit with a special code so parent knows it's an
                        exec failure
63                  os._exit(127)
64          else:
65              # --- PARENT PROCESS ---
66              print(f"CHILD {k} PID {pid} START")
67
68              # Step 3: Wait for termination
69              try:
70                  # waitpid(pid, options)
71                  _, status = os.waitpid(pid, 0)
72
73                  # Interpret status
74                  if os.WIFEXITED(status):
75                      exit_code = os.WEXITSTATUS(status)
76                      # Check if the exit was actually an exec failure
77                      if exit_code == 127:
78                          error_exit("E_EXEC", "cannot exec program")
79                      print(f"CHILD {k} PID {pid} EXIT {exit_code}")
80
81                  elif os.WIFSIGNALED(status):
82                      signum = os.WTERMSIG(status)
83                      print(f"CHILD {k} PID {pid} SIG {signum}")
84
85              except OSError:
```

```
                error_exit("E_WAIT", "waitpid failed")

    print(f"OK: COMPLETED {repeat}")

if __name__ == "__main__":
    main()
```

Sample Input 1

```
$ python3 spawnwait.py --cmd /bin/true --repeat 3
```

Sample Output 1

```
CHILD 1 PID 34879 START
CHILD 1 PID 34879 EXIT 0
CHILD 2 PID 34880 START
CHILD 2 PID 34880 EXIT 0
CHILD 3 PID 34881 START
CHILD 3 PID 34881 EXIT 0
OK: COMPLETED 3
```

Sample Input 2: Invalid command

```
$ python3 spawnwait.py --cmd /bin/sh --args -c,exit\ 7
```

Sample Output 2

```
CHILD 1 PID 34941 START
CHILD 1 PID 34941 EXIT 7
OK: COMPLETED 1
```

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

Problem Statement

- Implement a CLI tool named `timeoutwrap`.
- Inputs: `--seconds <t>` (required, integer 1..60), `--cmd <program>` (required), optional `--args <a1,a2,...>`.
- The parent process must fork a child that executes the given command.
- The parent arms an alarm for `t` seconds.
- If the child exits before timeout, cancel the alarm and output: `OK: EXIT <code>`.
- If timeout occurs first, send `SIGKILL` to the child, wait for it, and output: `OK: TIMEOUT KILLED`.
- On any error (fork, exec, signal, wait), output exactly one `ERROR:` line and exit non-zero.

Solution Implementation

```
import sys
import os
import signal
import time

# Error codes
E_USAGE = "E_USAGE"
E_RANGE = "E_RANGE"
E_FORK = "E_FORK"
E_EXEC = "E_EXEC"
E_WAIT = "E_WAIT"
E_SIGNAL = "E_SIGNAL"
```

```
13
14 def error_exit(code, message):
15     print(f"ERROR: {code}: {message}", file=sys.stderr)
16     sys.exit(1)
17
18 # Global variable to track the child PID
19 child_pid = -1
20
21 def alarm_handler(signum, frame):
22     """This function runs when the alarm timer reaches zero."""
23     global child_pid
24     if child_pid > 0:
25         try:
26             # Send SIGKILL to the child
27             os.kill(child_pid, signal.SIGKILL)
28         except OSError:
29             pass # Child might have just finished
30
31 def main():
32     global child_pid
33
34     # 1. Parse Arguments
35     args = sys.argv[1:]
36     seconds = None
37     cmd = None
38     cmd_args = []
39
40     i = 0
41     while i < len(args):
42         if args[i] == "--seconds":
43             if i + 1 < len(args):
44                 try:
45                     seconds = int(args[i+1])
46                     if not (1 <= seconds <= 60): raise ValueError
47                 except ValueError:
48                     error_exit(E_RANGE, "seconds must be in 1..60")
49                 i += 2
50             else:
```

```
                error_exit(E_USAGE, "missing value for --seconds")
        elif args[i] == "--cmd":
            if i + 1 < len(args):
                cmd = args[i+1]
                i += 2
            else:
                error_exit(E_USAGE, "missing value for --cmd")
        elif args[i] == "--args":
            if i + 1 < len(args):
                cmd_args = args[i+1].split(',')
                i += 2
            else:
                error_exit(E_USAGE, "missing value for --args")
        else:
            error_exit(E_USAGE, f"unrecognized argument: {args[i]}")

    if seconds is None or cmd is None:
        error_exit(E_USAGE, "--seconds and --cmd are required")

    # 2. Setup Signal Handler
    # signal.signal is the Python equivalent to sigaction()
    signal.signal(signal.SIGALRM, alarm_handler)

    # 3. Fork and Exec
    try:
        child_pid = os.fork()
    except OSError:
        error_exit(E_FORK, "fork failed")

    if child_pid == 0:
        # --- CHILD PROCESS ---
        try:
            os.execvp(cmd, [cmd] + cmd_args)
        except OSError:
            os._exit(127) # Exec failure
    else:
        # --- PARENT PROCESS ---
        # Arm the alarm
```

```
89          signal.alarm(seconds)
90
91          try:
92              # Wait for the child to change state
93              pid, status = os.waitpid(child_pid, 0)
94
95              # Cancel the alarm because the child finished
96              signal.alarm(0)
97
98              # Analyze how the child died
99              if os.WIFEXITED(status):
100                 code = os.WEXITSTATUS(status)
101                 if code == 127:
102                     error_exit(E_EXEC, "cannot exec program")
103                 print(f"OK: EXIT {code}")
104
105             elif os.WIFSIGNALED(status):
106                 sig = os.WTERMSIG(status)
107                 if sig == signal.SIGKILL:
108                     # This means our alarm_handler killed it
109                     print("OK: TIMEOUT KILLED")
110                 else:
111                     # Child died by some other signal (e.g. SIGINT)
112                     print(f"OK: SIG {sig}")
113
114         except OSError:
115             error_exit(E_WAIT, "waitpid failed")
116
117 if __name__ == "__main__":
118     main()
```

Sample Input 1: Command completes before timeout

```
$ python3 timeoutwrap.py --seconds 2 --cmd /bin/sh --args -c,
sleep\ 1
```

Sample Output 1

```
OK: EXIT 0
```

Sample Input 2: Command exceeds timeout

```
$ python3 timeoutwrap.py --seconds 1 --cmd /bin/sh --args -c,
sleep\ 7
```

Sample Output 2

```
OK: TIMEOUT KILLED
```

Sample Input 3

```
$ python3 timeoutwrap.py --seconds 70 --cmd /bin/true
```

Sample Output 3

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

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

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 arguments 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 exactly one line: `ERROR: E_STAGE: stage <name> exit <code>`.
- If a stage is terminated by a signal, print exactly one line: `ERROR: E_STAGE: stage <name> sig <signum>`.

Solution Implementation

```
import sys
import os

def error_exit(msg):
    print(f"ERROR: {msg}")
    sys.exit(1)

def main():
    # 1. Argument Parsing
    args_raw = sys.argv[1:]
    params = {}
    i = 0
```

```
    while i < len(args_raw):
        key = args_raw[i].lstrip('-')
        if i + 1 < len(args_raw):
            params[key] = args_raw[i+1]
            i += 2
        else:
            error_exit("E_USAGE: Missing value for argument")

    # Required check
    for req in ['producer', 'filter', 'consumer']:
        if req not in params:
            error_exit("E_USAGE: Missing required stage")

    # Prepare command lists
    stages = [
        ("producer", params['producer'], params.get('producer-args', '').split(',') if params.get('producer-args') else []),
        ("filter", params['filter'], params.get('filter-args', '').split(',') if params.get('filter-args') else []),
        ("consumer", params['consumer'], params.get('consumer-args', '').split(',') if params.get('consumer-args') else [])
    ]

    # 2. Create Pipes
    # pipe1: Producer -> Filter | pipe2: Filter -> Consumer
    p1_read, p1_write = os.pipe()
    p2_read, p2_write = os.pipe()

    pids = {}

    # 3. Spawn Stages
    for name, cmd, cmd_args in stages:
        try:
            pid = os.fork()
            if pid == 0:  # CHILD
                # Redirect STDERR and STDOUT to /dev/null as per spec
                devnull = os.open(os.devnull, os.O_WRONLY)
                os.dup2(devnull, sys.stderr.fileno())
```

```
48
49                  if name == "producer":
50                      os.dup2(p1_write, sys.stdout.fileno())
51                  elif name == "filter":
52                      os.dup2(p1_read, sys.stdin.fileno())
53                      os.dup2(p2_write, sys.stdout.fileno())
54                  elif name == "consumer":
55                      os.dup2(p2_read, sys.stdin.fileno())
56                      os.dup2(devnull, sys.stdout.fileno()) # Consumer
                            output to devnull
57
58                  # Close all pipe fds in child
59                  for fd in [p1_read, p1_write, p2_read, p2_write,
                        devnull]:
60                      os.close(fd)
61
62                  os.execvp(cmd, [cmd] + cmd_args)
63                  os._exit(127) # Exec failed
64
65              pids[name] = pid
66          except OSError:
67              error_exit("E_FORK: Fork failed")
68
69      # 4. PARENT: Close all pipe ends
70      for fd in [p1_read, p1_write, p2_read, p2_write]:
71          os.close(fd)
72
73      # 5. Wait and Report (Check in fixed order: producer, filter,
            consumer)
74      final_error = None
75      for name, _, _ in stages:
76          _, status = os.waitpid(pids[name], 0)
77
78          if final_error is None: # Only capture the first error found in
                order
79              if os.WIFEXITED(status):
80                  code = os.WEXITSTATUS(status)
81                  if code != 0:
```

```
                    # code 127 is our custom exec failure
                    msg = "cannot exec" if code == 127 else f"exit {
                        code}"
                    final_error = f"E_STAGE: stage {name} {msg}"
            elif os.WIFSIGNALED(status):
                signum = os.WTERMSIG(status)
                final_error = f"E_STAGE: stage {name} sig {signum}"

    if final_error:
        print(f"ERROR: {final_error}")
        sys.exit(1)
    else:
        print("OK: PIPELINE SUCCESS")

if __name__ == "__main__":
    main()
```

Sample Input 1

```
$ python3 pipechain.py --producer /bin/echo --producer-args
world --filter /usr/bin/tr --filter-args a-z,A-Z --consumer
/usr/bin/wc --consumer-args -c
```

Sample Output 1

```
OK: PIPELINE SUCCESS
```

Sample Input 2: Producer fails

```
$ python3 pipechain.py --producer /bin/sh --producer-args
-c,exit\ 4 --filter /bin/cat --consumer /bin/true
```

Sample Output 2

```
ERROR: E_STAGE: stage producer exit 4
```

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

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.

Solution Implementation

```
import sys
import os
import argparse
import mmap
import struct
import posix_ipc # Note: 'pip install posix_ipc' is usually required for POSIX semaphores
import time

# Error codes
```

```
E_USAGE = "E_USAGE"
E_RANGE = "E_RANGE"
E_SHM   = "E_SHM"
E_MMAP  = "E_MMAP"
E_SEM   = "E_SEM"
E_FORK  = "E_FORK"
E_WAIT  = "E_WAIT"

def error_exit(code, message):
    print(f"ERROR: {code}: {message}")
    sys.exit(1)

def main():
    # 1. Parse and Validate Arguments
    parser = argparse.ArgumentParser(add_help=False)
    parser.add_argument('--procs', type=int)
    parser.add_argument('--iters', type=int)
    parser.add_argument('--name')
    args, _ = parser.parse_known_args()

    if args.procs is None or args.iters is None or args.name is None:
        error_exit(E_USAGE, "missing required arguments")

    if not (2 <= args.procs <= 16):
        error_exit(E_RANGE, "procs must be in 2..16")
    if not (1 <= args.iters <= 100000):
        error_exit(E_RANGE, "iters must be in 1..100000")
    if not args.name.isalnum():
        error_exit(E_RANGE, "name must be alphanumeric only")

    shm_name = f"/shm_{args.name}"
    sem_name = f"/sem_{args.name}"

    shm = None
    sem = None

    try:
        # 2. Create and Map Shared Memory
```

```
48          try:
49              # Create a 8-byte shared memory object (for 64-bit int)
50              shm = posix_ipc.SharedMemory(shm_name, flags=posix_ipc.
                    O_CREAT | posix_ipc.O_TRUNC, size=8)
51              map_file = mmap.mmap(shm.fd, 8)
52              # Initialize counter to 0
53              map_file.seek(0)
54              map_file.write(struct.pack('q', 0))
55          except Exception as e:
56              error_exit(E_SHM, f"could not create shm: {e}")
57
58          # 3. Create Semaphore
59          try:
60              sem = posix_ipc.Semaphore(sem_name, flags=posix_ipc.O_CREAT
                    | posix_ipc.O_TRUNC, initial_value=1)
61          except Exception as e:
62              error_exit(E_SEM, f"could not create semaphore: {e}")
63
64          # 4. Fork Processes
65          pids = []
66          for i in range(args.procs):
67              try:
68                  pid = os.fork()
69                  if pid == 0: # CHILD
70                      child_shm = posix_ipc.SharedMemory(shm_name)
71                      child_map = mmap.mmap(child_shm.fd, 8)
72                      child_sem = posix_ipc.Semaphore(sem_name)
73
74                      for _ in range(args.iters):
75                          child_sem.acquire()
76                          # Read 64-bit int, increment, and write back
77                          child_map.seek(0)
78                          val = struct.unpack('q', child_map.read(8))[0]
79                          child_map.seek(0)
80                          child_map.write(struct.pack('q', val + 1))
81                          child_sem.release()
82
83                      child_map.close()
```

```
                    os._exit(0)
                else:
                    pids.append(pid)
            except OSError:
                error_exit(E_FORK, "fork failed")

        # 5. Parent Wait
        for pid in pids:
            try:
                os.waitpid(pid, 0)
            except OSError:
                error_exit(E_WAIT, "waitpid failed")

        # 6. Read Final Value
        map_file.seek(0)
        final_val = struct.unpack('q', map_file.read(8))[0]
        print(f"OK: FINAL {final_val}")

    finally:
        # 7. Cleanup (Unlink)
        if shm:
            shm.close_fd()
            try: posix_ipc.unlink_shared_memory(shm_name)
            except: pass
        if sem:
            sem.close()
            try: posix_ipc.unlink_semaphore(sem_name)
            except: pass

if __name__ == "__main__":
    main()
```

Sample Input 1

```
$ python3 shmcounter.py --procs 16 --iters 100000 --name t
```

Sample Output 1

```
OK: FINAL 1600000
```

Sample Input 2: Invalid Arguments

```
$ python3 shmcounter.py --procs 2 --iters 0 --name t
```

Sample Output 2

```
ERROR: E_RANGE: iters must be in 1..100000
```

Experiment 9: Threaded Deterministic Reducer (pthreads + mutex)

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.

Solution Implementation

```python
import threading
import sys

def error(code, message):
    print(f"ERROR: {code}: {message}")
    sys.exit(1)

def worker(start, end, total, lock):
    local_sum = 0
    for i in range(start, end + 1):
        local_sum += i

    # protect shared total
    with lock:
        total[0] += local_sum
```

```
16
17 def main():
18     # very simple argument parsing
19     if "--threads" not in sys.argv or "--n" not in sys.argv:
20         error("E_USAGE", "missing required arguments")
21
22     try:
23         t = int(sys.argv[sys.argv.index("--threads") + 1])
24         n = int(sys.argv[sys.argv.index("--n") + 1])
25     except (ValueError, IndexError):
26         error("E_USAGE", "invalid arguments")
27
28     if t < 1 or t > 32:
29         error("E_RANGE", "threads must be in 1..32")
30
31     if n < 1 or n > 1_000_000:
32         error("E_RANGE", "n must be in 1..1000000")
33
34     threads = []
35     lock = threading.Lock()
36     total = [0]  # mutable container for shared sum
37
38     chunk = n // t
39     remainder = n % t
40     start = 1
41
42     for i in range(t):
43         end = start + chunk - 1
44         if i < remainder:
45             end += 1
46
47         th = threading.Thread(
48             target=worker,
49             args=(start, end, total, lock)
50         )
51         threads.append(th)
52         th.start()
53
```

```
54          start = end + 1
55
56      for th in threads:
57          th.join()
58
59      print(f"OK: SUM {total[0]}")
60
61  if __name__ == "__main__":
62      main()
```

Sample Input 1

```
$ $ python3 threadsum.py --threads 4 --n 20
```

Sample Output 1

```
OK: SUM 210
```

Sample Input 2: Invalid Arguments

```
$ ./thrsum --threads 50 --n 500
```

Sample Output 2

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

Experiment 10: Bounded Buffer Producer-Consumer with Semaphores

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.

Solution Implementation

```
#!/usr/bin/env python3
import sys
import argparse
import threading
from queue import Queue
from threading import Semaphore, Lock

def parse_args():
    parser = argparse.ArgumentParser()
```

```
    parser.add_argument("--buf", type=int, required=True)
    parser.add_argument("--producers", type=int, required=True)
    parser.add_argument("--consumers", type=int, required=True)
    parser.add_argument("--items", type=int, required=True)
    return parser.parse_args()

def main():
    try:
        args = parse_args()
        B = args.buf
        P = args.producers
        C = args.consumers
        M = args.items

        if not (1 <= B <= 1024 and 1 <= P <= 16 and 1 <= C <= 16 and 1
           <= M <= 100000):
            print("ERROR: Invalid arguments")
            sys.exit(1)

        buffer = Queue(maxsize=B)
        empty_slots = Semaphore(B)
        filled_slots = Semaphore(0)
        counter_lock = Lock()
        consume_lock = Lock()

        produced_count = [0]
        consumed_count = [0]
        consumed_sum = [0]
        next_item = [1]  # shared counter for items 1..M

        def producer():
            nonlocal next_item
            while True:
                with counter_lock:
                    if next_item[0] > M:
                        break
                    item = next_item[0]
                    next_item[0] += 1
```

```
47                  empty_slots.acquire()
48                  buffer.put(item)
49                  filled_slots.release()
50                  with counter_lock:
51                      produced_count[0] += 1
52
53          def consumer():
54              while True:
55                  filled_slots.acquire()
56                  if consumed_count[0] >= M:
57                      filled_slots.release()
58                      break
59                  item = buffer.get()
60                  empty_slots.release()
61                  with consume_lock:
62                      consumed_count[0] += 1
63                      consumed_sum[0] += item
64
65          producers = [threading.Thread(target=producer) for _ in range(P
               )]
66          consumers = [threading.Thread(target=consumer) for _ in range(C
               )]
67
68          for th in producers + consumers:
69              th.start()
70          for th in producers + consumers:
71              th.join()
72
73          print(f"OK: PRODUCED {produced_count[0]}")
74          print(f"OK: CONSUMED {consumed_count[0]}")
75          print(f"OK: SUM {consumed_sum[0]}")
76
77      except Exception as e:
78          print("ERROR:", e)
79          sys.exit(1)
80
81  if __name__ == "__main__":
82      main()
```

Sample Input 1

```
$ python3 pcbuf.py --buf 10 --producers 2 --consumers 2
--items 100
```

Sample Output 1

```
OK: PRODUCED 100
OK: CONSUMED 100
OK: SUM 5050
```

Sample Input 2: Invalid Arguments

```
$ python3 pcbuf.py --buf 0 --producers 2 --consumers 2
--items 50
```

Sample Output 2

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

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

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:
  - FCFS: First-Come, First-Served.
  - SJF: Non-preemptive Shortest Job First; among arrived processes, choose the shortest burst, tie-break by arrival then pid.
- For each algorithm, output exactly:

```
ALG <name>
GANTT <pid1>@<t0>-<t1> <pid2>@<t1>-<t2> ...
OK: AVG_WAIT <w> AVG_TAT <t>
```

  Averages are rounded to 2 decimal places.
- On any parse or validation error, output one `ERROR:` line and exit non-zero.

Solution Implementation

```
1
2 import sys
3 from collections import namedtuple
4
5 Process = namedtuple("Process", ["pid", "arrival", "burst"])
6
7 def parse_input():
```

```
    lines = [line.strip() for line in sys.stdin if line.strip()]
    if not lines:
        print("ERROR: E_INPUT: empty input", file=sys.stderr)
        sys.exit(1)
    if lines[0] != "pid,arrival,burst":
        print("ERROR: E_INPUT: missing or incorrect header", file=sys.
           stderr)
        sys.exit(1)

    processes = []
    seen_pids = set()
    for i, line in enumerate(lines[1:], start=2):
        parts = line.split(",")
        if len(parts) != 3:
            print(f"ERROR: E_INPUT: line {i} malformed", file=sys.
               stderr)
            sys.exit(1)
        pid = parts[0].strip()
        if pid in seen_pids:
            print(f"ERROR: E_DUPPID: duplicate pid {pid}", file=sys.
               stderr)
            sys.exit(1)
        seen_pids.add(pid)
        try:
            arrival = int(parts[1].strip())
            burst = int(parts[2].strip())
        except ValueError:
            print(f"ERROR: E_INPUT: arrival and burst must be integers
               on line {i}", file=sys.stderr)
            sys.exit(1)
        if arrival < 0 or burst <= 0:
            print("ERROR: E_RANGE: arrival and burst must be non-
               negative; burst must be > 0", file=sys.stderr)
            sys.exit(1)
        processes.append(Process(pid, arrival, burst))
    return processes

def fcfs(processes):
```

```
    processes.sort(key=lambda p: p.arrival)
    time = 0
    gantt = []
    wait_times = []
    tat_times = []

    for p in processes:
        if time < p.arrival:
            gantt.append(f"IDLE@{time}-{p.arrival}")
            time = p.arrival
        start = time
        end = start + p.burst
        gantt.append(f"{p.pid}@{start}-{end}")
        wait_times.append(start - p.arrival)
        tat_times.append(end - p.arrival)
        time = end

    avg_wait = round(sum(wait_times)/len(wait_times), 2)
    avg_tat = round(sum(tat_times)/len(tat_times), 2)

    print("ALG FCFS")
    print("GANTT " + " ".join(gantt))
    print(f"OK: AVG_WAIT {avg_wait:.2f} AVG_TAT {avg_tat:.2f}")

def sjf(processes):
    time = 0
    gantt = []
    wait_times = []
    tat_times = []
    remaining = processes.copy()

    while remaining:
        available = [p for p in remaining if p.arrival <= time]
        if not available:
            next_arrival = min(remaining, key=lambda p: p.arrival)
            gantt.append(f"IDLE@{time}-{next_arrival.arrival}")
            time = next_arrival.arrival
            available = [p for p in remaining if p.arrival <= time]
```

```
        available.sort(key=lambda p: (p.burst, p.arrival, p.pid))
        p = available[0]
        start = time
        end = start + p.burst
        gantt.append(f"{p.pid}@{start}-{end}")
        wait_times.append(start - p.arrival)
        tat_times.append(end - p.arrival)
        time = end
        remaining.remove(p)

    avg_wait = round(sum(wait_times)/len(wait_times), 2)
    avg_tat = round(sum(tat_times)/len(tat_times), 2)

    print("ALG SJF")
    print("GANTT " + " ".join(gantt))
    print(f"OK: AVG_WAIT {avg_wait:.2f} AVG_TAT {avg_tat:.2f}")

def main():
    processes = parse_input()
    fcfs(processes)
    sjf(processes)

if __name__ == "__main__":
    main()
```

Sample Input

```
$ printf 'pid,arrival,burst
P1,0,5
P2,2,2
P3,4,1
' | python cpu_scheduling.py
```

Sample Output

```
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
```

Experiment 12: CPU Scheduling Simulator II (Round Robin)

Problem Statement

- Implement a CLI tool named `schedsim2`.
- Input is provided via `stdin` as CSV with header: `pid,arrival,burst`.
- Argument: `--q <quantum>` (integer 1..1000).
- Simulate preemptive Round Robin scheduling:
  - Newly arrived processes are enqueued at the end at their arrival time.
  - When a time slice ends and the running process is not finished, enqueue it at the end.
  - If CPU becomes idle, time jumps to the next process arrival.
- Output exactly:

```
ALG RR
GANTT <pid1>@<t0>-<t1> <pid2>@<t1>-<t2> ...
OK: AVG_WAIT <w> AVG_TAT <t>
```

  Include `IDLE` segments if CPU is idle.
- On error, output one `ERROR:` line and exit non-zero.

Solution Implementation

```
1
2 import sys
3 import csv
4 import argparse
5
```

```
def error(code, msg):
    print(f"ERROR: {code}: {msg}")
    sys.exit(1)

# Parse arguments
parser = argparse.ArgumentParser()
parser.add_argument("--q", type=int, required=True)
args = parser.parse_args()

quantum = args.q
if not (1 <= quantum <= 1000):
    error("E_RANGE", "quantum must be in 1..1000")

# Read CSV from stdin
try:
    reader = csv.DictReader(sys.stdin)
    processes = []
    pids_set = set()
    for row in reader:
        pid = row["pid"]
        if pid in pids_set:
            error("E_DUPPID", f"duplicate pid {pid}")
        pids_set.add(pid)
        try:
            arrival = int(row["arrival"])
            burst = int(row["burst"])
            if burst <= 0:
                error("E_RANGE", "burst must be positive")
        except:
            error("E_INPUT", "invalid arrival or burst")
        processes.append({
            "pid": pid,
            "arrival": arrival,
            "burst": burst,
            "remaining": burst,
            "completion": 0
        })
except Exception:
```

```
    error("E_INPUT", "malformed CSV")

# Sort by arrival time, then PID lexicographically
processes.sort(key=lambda x: (x["arrival"], x["pid"]))

time = 0
ready_queue = []
gantt = []

# Keep track of processes left
remaining_processes = processes.copy()

while remaining_processes or ready_queue:
    # Add new arrivals at current time
    arrivals = [p for p in remaining_processes if p["arrival"] <= time]
    arrivals.sort(key=lambda x: x["pid"])  # tie-breaker
    for p in arrivals:
        ready_queue.append(p)
        remaining_processes.remove(p)

    if not ready_queue:
        if remaining_processes:
            next_arrival = min(remaining_processes, key=lambda x: x["
               arrival"])
            gantt.append(f"IDLE@{time}-{next_arrival['arrival']}")
            time = next_arrival["arrival"]
            continue
        else:
            break

    # Pick first process in ready queue
    current = ready_queue.pop(0)
    run_time = min(current["remaining"], quantum)
    gantt.append(f"{current['pid']}@{time}-{time + run_time}")
    time += run_time
    current["remaining"] -= run_time

    # Check if current finished
```

```
    if current["remaining"] == 0:
        current["completion"] = time
    else:
        # Requeue unfinished process
        ready_queue.append(current)

# Compute metrics
total_wt = 0
total_tat = 0
for p in processes:
    tat = p["completion"] - p["arrival"]
    wt = tat - p["burst"]
    total_wt += wt
    total_tat += tat

avg_wt = total_wt / len(processes)
avg_tat = total_tat / len(processes)

# Output
print("ALG RR")
print("GANTT", " ".join(gantt))
print(f"OK: AVG_WAIT {avg_wt:.2f} AVG_TAT {avg_tat:.2f}")
```

Sample Input

```
printf 'pid,arrival,burst
    P1,0,5
    P2,1,3
    P3,2,8
    P4,3,6' | python3 cpu_scheduling_round_robin.py --q 2
```

Sample Output

```
ALG RR
GANTT P1@0-2 P1@2-4 P2@4-6 P3@6-8 P1@8-9
P4@9-11 P2@11-12 P3@12-14 P4@14-16 P3@16-18 P4@18-20 P3@20-22
OK: AVG_WAIT 8.75 AVG_TAT 14.25
```

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

Problem Statement

- Implement a CLI tool named `schedprio`.
- Input is provided via `stdin` as CSV with header: `pid,arrival,burst,priority`.
- Smaller `priority` means higher priority.
- Scheduling rules:
  - Non-preemptive: once a process starts, it runs to completion.
  - Aging: for every unit of waiting in the ready queue, effective priority decreases by 1 (minimum 0).
  - At dispatch, select ready process with lowest effective priority; tie-break by arrival then pid.
- Output exactly:

```
ALG PRIO_AGING
GANTT <pid1>@<t0>-<t1> <pid2>@<t1>-<t2> ...
OK: AVG_WAIT <w> AVG_TAT <t>
```

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

Solution Implementation

```
#!/usr/bin/env python3
import sys

def error(code, msg):
```

```
    print(f"ERROR: {code}: {msg}")
    sys.exit(1)

def read_processes_from_stdin():
    lines = [line.strip() for line in sys.stdin if line.strip()]
    if not lines:
        error("E_INPUT", "no input provided")
    # Skip header if present
    if lines[0].lower().startswith("pid"):
        lines = lines[1:]
    processes = []
    seen_pid = set()
    for i, line in enumerate(lines):
        parts = line.split(",")
        if len(parts) != 4:
            error("E_INPUT", f"invalid input format in line {i+2}")
        pid = parts[0].strip()
        if pid in seen_pid:
            error("E_DUPPID", f"duplicate pid '{pid}'")
        seen_pid.add(pid)
        try:
            arrival = int(parts[1])
            burst = int(parts[2])
            priority = int(parts[3])
        except:
            error("E_INPUT", f"non-integer field in line {i+2}")
        if arrival < 0:
            error("E_RANGE", f"arrival must be >=0 in line {i+2}")
        if burst <= 0:
            error("E_RANGE", f"burst must be >0 in line {i+2}")
        if not (0 <= priority <= 99):
            error("E_RANGE", f"priority must be in 0..99 in line {i+2}"
                )
        processes.append({
            "pid": pid,
            "arrival": arrival,
            "burst": burst,
            "priority": priority
```

```
        })
    return processes

def schedule(processes):
    time = 0
    ready = []
    todo = processes[:]
    completed = {}
    gantt = []

    while todo or ready:
        for p in todo[:]:
            if p["arrival"] <= time:
                ready.append(p)
                todo.remove(p)

        if not ready:
            next_time = min(p["arrival"] for p in todo)
            gantt.append(f"IDLE@{time}-{next_time}")
            time = next_time
            continue

        # Aging at dispatch
        for p in ready:
            waiting = time - p["arrival"]
            p["eff_prio"] = max(0, p["priority"] - waiting)

        ready.sort(key=lambda x: (x["eff_prio"], x["arrival"], x["pid"
           ]))
        cur = ready.pop(0)

        start = time
        time += cur["burst"]
        gantt.append(f"{cur['pid']}@{start}-{time}")
        completed[cur["pid"]] = time

    # Metrics
    total_wait = 0
```

```
    total_tat = 0
    n = len(processes)

    for p in processes:
        ct = completed[p["pid"]]
        tat = ct - p["arrival"]
        wt = tat - p["burst"]
        total_wait += wt
        total_tat += tat

    return gantt, total_wait/n, total_tat/n

def main():
    processes = read_processes_from_stdin()
    gantt, avg_wait, avg_tat = schedule(processes)
    print("\nALG PRIO_AGING")
    print("GANTT", " ".join(gantt))
    print(f"OK: AVG_WAIT {avg_wait:.2f} AVG_TAT {avg_tat:.2f}")

if __name__ == "__main__":
    main()
```

Sample Input

```
printf 'pid,arrival,burst,priority
     A,0,6,7
     B,1,11,12
     ' | python3 priority_scheduling_cli.py
```

Sample Output

```
ALG PRIO_AGING
GANTT A@0-6 B@6-17
OK: AVG_WAIT 2.50 AVG_TAT 11.00
```

Experiment 14: Deadlock Avoidance using Banker's Algorithm

Problem Statement

- Implement a CLI tool named `banker`.
- Input is provided via `stdin` in the exact format:
  - First line: `P R` (number of processes and resource types).
  - Next `P` lines: Allocation matrix (`R` integers per line).
  - Next `P` lines: Max matrix (`R` integers per line).
  - Last line: Available vector (`R` integers).
- Validate that `Allocation[i][j] <= Max[i][j]` for all entries.
- Apply Banker's safety algorithm.
- Output exactly:

```
OK: SAFE
OK: SEQ <p0> <p1> ... <p(P-1)>
```

- If the system is unsafe, output exactly:

```
OK: UNSAFE
```

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

Solution Implementation

```
#!/usr/bin/env python3
import sys
```

```
3
4  def error(msg):
5      print(f"ERROR: {msg}")
6      sys.exit(1)
7
8  def read_input():
9      lines = [line.strip() for line in sys.stdin if line.strip()]
10     if len(lines) < 3:
11         error("Not enough input lines")
12     try:
13         P, R = map(int, lines[0].split())
14     except:
15         error("First line must be 'P R' with integers")
16
17     expected_lines = 1 + P*2 + 1
18     if len(lines) != expected_lines:
19         error(f"Expected {expected_lines} lines for {P} processes and {
               R} resources, got {len(lines)}")
20
21     # Allocation matrix
22     allocation = []
23     for i in range(P):
24         try:
25             row = list(map(int, lines[1 + i].split()))
26         except:
27             error(f"Invalid integers in Allocation row {i}")
28         if len(row) != R:
29             error(f"Allocation row {i} must have {R} entries")
30         allocation.append(row)
31
32     # Max matrix
33     max_need = []
34     for i in range(P):
35         try:
36             row = list(map(int, lines[1 + P + i].split()))
37         except:
38             error(f"Invalid integers in Max row {i}")
39         if len(row) != R:
```

```
            error(f"Max row {i} must have {R} entries")
        max_need.append(row)

    # Available vector
    try:
        available = list(map(int, lines[-1].split()))
    except:
        error("Invalid integers in Available vector")
    if len(available) != R:
        error(f"Available vector must have {R} entries")

    # Basic validation
    for i in range(P):
        for j in range(R):
            if allocation[i][j] > max_need[i][j]:
                error(f"Allocation cannot exceed Max for process {i},
                    resource {j}")
            if allocation[i][j] < 0 or max_need[i][j] < 0 or available[
                j] < 0:
                error("Resource values must be non-negative")

    return allocation, max_need, available

def bankers_algorithm(allocation, max_need, available):
    n = len(allocation)
    m = len(available)
    need = [[max_need[i][j] - allocation[i][j] for j in range(m)] for i
        in range(n)]
    finish = [False]*n
    safe_sequence = []
    work = available[:]

    while len(safe_sequence) < n:
        found = False
        for i in range(n):
            if not finish[i] and all(need[i][j] <= work[j] for j in
                range(m)):
                for j in range(m):
```

```
                    work[j] += allocation[i][j]
                finish[i] = True
                safe_sequence.append(i)
                found = True
        if not found:
            print("OK: UNSAFE")
            return

    print("OK: SAFE")
    print("OK: SEQ", *safe_sequence)

def main():
    allocation, max_need, available = read_input()
    bankers_algorithm(allocation, max_need, available)

if __name__ == "__main__":
    main()
```

Sample Input

```
printf '5 3
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
3 3 2
' | python3 deadlock_banker_algo.py
```

Sample Output

```
OK: SAFE
OK: SEQ 1 3 4 0 2
```

Experiment 15: Deadlock Detection via Wait-For Graph Cycle

Problem Statement

- Implement a CLI tool named `wfgcheck`.
- Input is provided via `stdin` in the following format:
  - First line: `P E` (number of processes and edges).
  - Next `E` lines: `u v` meaning process `u` waits for `v` (directed edge `u -> v`).
  - Processes are numbered `0..P-1`.
- Output exactly one of:
  - No deadlock:

    ```
    OK: DEADLOCK NO
    ```

  - Deadlock detected:

    ```
    OK: DEADLOCK YES
    OK: CYCLE <p0> <p1> ... <pk> <p0>
    ```

- If multiple cycles exist, output the cycle with:
  - Smallest starting node.
  - If tied, smallest lexicographic sequence.
- On error, output one `ERROR:` line and exit non-zero.

Solution Implementation

```
#!/usr/bin/env python3
import sys
```

```
from collections import deque

def error(msg):
    print(f"ERROR: {msg}")
    sys.exit(1)

def read_input():
    lines = [line.strip() for line in sys.stdin if line.strip()]
    if len(lines) < 1:
        error("No input provided")

    try:
        P, E = map(int, lines[0].split())
    except:
        error("First line must be two integers: P E")

    if len(lines) != 1 + E:
        error(f"Expected {E} edge lines, got {len(lines)-1}")

    edges = []
    for i, line in enumerate(lines[1:]):
        parts = line.split()
        if len(parts) != 2:
            error(f"Edge line {i+1} must have 2 integers: '{line}'")
        try:
            u, v = map(int, parts)
        except:
            error(f"Cannot parse edge line {i+1}: '{line}'")
        if not (0 <= u < P) or not (0 <= v < P):
            error(f"Edge ({u},{v}) out of range 0..{P-1}")
        edges.append((u, v))

    return P, edges

def build_graph(P, edges):
    graph = [[] for _ in range(P)]
    indegree = [0]*P
    for u, v in edges:
```

```
        graph[u].append(v)
        indegree[v] += 1
    return graph, indegree

def kahn_deadlock(P, graph, indegree):
    q = deque([i for i in range(P) if indegree[i] == 0])
    processed = 0
    indegree_copy = indegree.copy()
    while q:
        node = q.popleft()
        processed += 1
        for neigh in graph[node]:
            indegree_copy[neigh] -= 1
            if indegree_copy[neigh] == 0:
                q.append(neigh)
    if processed == P:
        return None

    remaining = [i for i in range(P) if indegree_copy[i] > 0]
    visited = [False]*P
    rec_stack = [False]*P
    canonical_cycle = None

    def dfs(node, path):
        nonlocal canonical_cycle
        visited[node] = True
        rec_stack[node] = True
        path.append(node)
        for neigh in sorted(graph[node]):
            if neigh not in remaining:
                continue
            if rec_stack[neigh]:
                idx = path.index(neigh)
                cycle = path[idx:] + [neigh]
                if canonical_cycle is None:
                    canonical_cycle = cycle
                else:
                    old_start = min(canonical_cycle[:-1])
```

```
                    new_start = min(cycle[:-1])
                    if new_start < old_start:
                        canonical_cycle = cycle
                    elif new_start == old_start and cycle <
                       canonical_cycle:
                        canonical_cycle = cycle
            elif not visited[neigh]:
                dfs(neigh, path.copy())
        rec_stack[node] = False

    for u in sorted(remaining):
        if not visited[u]:
            dfs(u, [])

    return canonical_cycle

def main():
    P, edges = read_input()
    graph, indegree = build_graph(P, edges)
    cycle = kahn_deadlock(P, graph, indegree)

    if cycle:
        print("OK: DEADLOCK YES")
        print("OK: CYCLE", ' '.join(map(str, cycle)))
    else:
        print("OK: DEADLOCK NO")

if __name__ == "__main__":
    main()
```

Sample Input 1

```
printf '4 4
0 1
1 2
2 0
2 3
' | python3 wfgcheck.py
```

Sample Output 1

```
OK: DEADLOCK YES
OK: CYCLE 0 1 2 0
```

Sample Input 2

```
printf '4 3
0 1
1 2
2 3
' | python3 wfgcheck.py
```

Sample Output 2

```
OK: DEADLOCK NO
```

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

Problem Statement

- Implement a CLI tool named `memfit`.
- Input is provided via `stdin` in the exact format:
  - Line 1: `B` — number of memory blocks.
  - Line 2: `B` integers — block sizes.
  - Line 3: `P` — number of processes.
  - Line 4: `P` integers — process sizes.
- Simulate the following algorithms independently, each starting from the original block list:
  - `FIRST_FIT`
  - `BEST_FIT`
  - `WORST_FIT`
- For each algorithm, output exactly:

```
ALG <name>
PROC <i> SIZE <s> -> BLOCK <j>
PROC <i> SIZE <s> -> FAIL
OK: ALLOCATED <k>/<P>
```

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

Solution Implementation

```
1
```

```python
import sys

def error(msg):
    print(f"ERROR: {msg}")
    sys.exit(1)

def read_input():
    try:
        lines = [line.strip() for line in sys.stdin if line.strip()]
        if len(lines) < 4:
            error("Not enough input lines")

        B = int(lines[0])
        if B <= 0:
            error("B must be positive")

        block_sizes = list(map(int, lines[1].split()))
        if len(block_sizes) != B or any(b <= 0 for b in block_sizes):
            error("Block sizes invalid or non-positive")

        P = int(lines[2])
        if P <= 0:
            error("P must be positive")

        process_sizes = list(map(int, lines[3].split()))
        if len(process_sizes) != P or any(p <= 0 for p in process_sizes
            ):
            error("Process sizes invalid or non-positive")

        return block_sizes, process_sizes
    except Exception as e:
        error(f"E_INPUT: {e}")

def first_fit(blocks, processes):
    allocation = []
    blocks_copy = blocks.copy()
    allocated_count = 0
    for i, p_size in enumerate(processes):
```

```
        placed = False
        for j, b_size in enumerate(blocks_copy):
            if b_size >= p_size:
                allocation.append(f"PROC {i} SIZE {p_size} -> BLOCK {j}
                    ")
                blocks_copy[j] -= p_size
                allocated_count += 1
                placed = True
                break
        if not placed:
            allocation.append(f"PROC {i} SIZE {p_size} -> FAIL")
    return allocation, allocated_count

def best_fit(blocks, processes):
    allocation = []
    blocks_copy = blocks.copy()
    allocated_count = 0
    for i, p_size in enumerate(processes):
        best_index = -1
        best_size = None
        for j, b_size in enumerate(blocks_copy):
            if b_size >= p_size and (best_size is None or b_size <
                best_size):
                best_index = j
                best_size = b_size
        if best_index != -1:
            allocation.append(f"PROC {i} SIZE {p_size} -> BLOCK {
                best_index}")
            blocks_copy[best_index] -= p_size
            allocated_count += 1
        else:
            allocation.append(f"PROC {i} SIZE {p_size} -> FAIL")
    return allocation, allocated_count

def worst_fit(blocks, processes):
    allocation = []
    blocks_copy = blocks.copy()
    allocated_count = 0
```

```
    for i, p_size in enumerate(processes):
        worst_index = -1
        worst_size = None
        for j, b_size in enumerate(blocks_copy):
            if b_size >= p_size and (worst_size is None or b_size >
                worst_size):
                worst_index = j
                worst_size = b_size
        if worst_index != -1:
            allocation.append(f"PROC {i} SIZE {p_size} -> BLOCK {
                worst_index}")
            blocks_copy[worst_index] -= p_size
            allocated_count += 1
        else:
            allocation.append(f"PROC {i} SIZE {p_size} -> FAIL")
    return allocation, allocated_count

def run_all_algorithms(blocks, processes):
    algorithms = [
        ("FIRST_FIT", first_fit),
        ("BEST_FIT", best_fit),
        ("WORST_FIT", worst_fit)
    ]
    for name, func in algorithms:
        allocation, allocated_count = func(blocks, processes)
        print(f"\nALG {name}")
        for line in allocation:
            print(line)
        print(f"OK: ALLOCATED {allocated_count}/{len(processes)}")

def main():
    blocks, processes = read_input()
    run_all_algorithms(blocks, processes)

if __name__ == "__main__":
    main()
```

Sample Input

```
printf '5
100 500 200 300 600
4
212 417 112 426
' | python3 mem_allocation.py
```

Sample Output

```
ALG FIRST_FIT
PROC 0 SIZE 212 -> BLOCK 1
PROC 1 SIZE 417 -> BLOCK 4
PROC 2 SIZE 112 -> BLOCK 1
PROC 3 SIZE 426 -> FAIL
OK: ALLOCATED 3/4

ALG BEST_FIT
PROC 0 SIZE 212 -> BLOCK 3
PROC 1 SIZE 417 -> BLOCK 1
PROC 2 SIZE 112 -> BLOCK 2
PROC 3 SIZE 426 -> BLOCK 4
OK: ALLOCATED 4/4

ALG WORST_FIT
PROC 0 SIZE 212 -> BLOCK 4
PROC 1 SIZE 417 -> BLOCK 1
PROC 2 SIZE 112 -> BLOCK 4
PROC 3 SIZE 426 -> FAIL
OK: ALLOCATED 3/4
```

Experiment 17: Paging Address Translation with Optional TLB

Problem Statement

- Implement a CLI tool named `pagetrans`.
- Command-line arguments:
  - `--pagesize <S>` where `S` is a power of two in the range 256..65536.
  - `--tlb <K>` where `K` is the TLB size (0..64).
- Input via `stdin` in the following 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:
  - Valid mapping:

    `OK: VA <vaddr> -> PA <paddr> (TLB HIT|TLB MISS)`

    when `K>0`; omit the TLB part when `K=0`.
  - Page fault:

    `OK: VA <vaddr> -> PAGEFAULT`
- After all queries, if `K>0`, output exactly:

  `OK: TLB_HITS <h> TLB_MISSES <m>`
- On error, output one `ERROR:` line and exit non-zero.

Solution Implementation

```python
#!/usr/bin/env python3
import sys
import argparse

# --- Parse command-line arguments ---
parser = argparse.ArgumentParser()
parser.add_argument("--pagesize", type=int, required=True, help="Page size (must be power of 2)")
parser.add_argument("--tlb", type=int, required=True, help="TLB size (0..64)")
args = parser.parse_args()

pagesize = args.pagesize
tlb_size = args.tlb

# --- Validate arguments ---
if pagesize < 256 or pagesize > 65536 or (pagesize & (pagesize - 1)) != 0:
    print(f"ERROR: E_RANGE: pagesize must be power of 2 between 256 and 65536")
    sys.exit(1)

if tlb_size < 0 or tlb_size > 64:
    print(f"ERROR: E_RANGE: TLB size must be 0..64")
    sys.exit(1)

# --- Read input from stdin ---
lines = [line.strip() for line in sys.stdin if line.strip()]
try:
    # Number of page table entries
    n = int(lines[0])
    page_table = {}

    # Read page table
    for i in range(1, n+1):
        parts = lines[i].split()
```

```
        if len(parts) != 3:
            raise ValueError(f"Invalid page table entry on line {i+1}")
        vpn, pfn, valid = map(int, parts)
        if valid not in (0, 1):
            print(f"ERROR: E_INPUT: valid must be 0 or 1")
            sys.exit(1)
        page_table[vpn] = (pfn, valid)

    # Number of queries
    q_index = n + 1
    Q = int(lines[q_index])
    queries = [int(v) for v in lines[q_index+1 : q_index+1+Q]]
    for v in queries:
        if v < 0:
            print(f"ERROR: E_RANGE: virtual address cannot be negative"
                )
            sys.exit(1)

except Exception as e:
    print(f"ERROR: E_INPUT: {e}")
    sys.exit(1)

# --- Initialize TLB if needed ---
TLB = [None] * tlb_size if tlb_size > 0 else None
tlb_hits = 0
tlb_misses = 0

# --- Process queries ---
for va in queries:
    vpn = va // pagesize
    offset = va % pagesize

    # Page table check
    if vpn not in page_table or page_table[vpn][1] == 0:
        print(f"OK: VA {va} -> PAGEFAULT")
        continue

    pfn = page_table[vpn][0]
```

```
70
71      # TLB logic
72      if tlb_size > 0:
73          index = vpn % tlb_size
74          entry = TLB[index]
75          if entry is not None and entry[0] == vpn:
76              # TLB hit
77              tlb_hits += 1
78              print(f"OK: VA {va} -> PA {pfn*pagesize + offset} (TLB HIT)
                    ")
79          else:
80              # TLB miss
81              tlb_misses += 1
82              TLB[index] = (vpn, pfn)
83              print(f"OK: VA {va} -> PA {pfn*pagesize + offset} (TLB MISS
                    )")
84      else:
85          # No TLB
86          print(f"OK: VA {va} -> PA {pfn*pagesize + offset}")
87
88  # --- Print TLB stats ---
89  if tlb_size > 0:
90      print(f"OK: TLB_HITS {tlb_hits} TLB_MISSES {tlb_misses}")
```

Sample Input

```
printf '4
0 5 1
1 9 1
2 3 0
3 7 1
5
0
256
512
768
1024
' | python3 pagetrans.py --pagesize 512 --tlb 4
```

Sample Output

```
OK: VA 0 -> PA 2560 (TLB MISS)
OK: VA 256 -> PA 2816 (TLB HIT)
OK: VA 512 -> PA 4608 (TLB MISS)
OK: VA 768 -> PA 4864 (TLB HIT)
OK: VA 1024 -> PAGEFAULT
OK: TLB_HITS 2 TLB_MISSES 2
```

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

Problem Statement

- Implement a CLI tool named `pagerepl`.
- Command-line argument:
  - `--frames <F>` where `F` is the number of frames (1..64).
- Input via `stdin` in the following format:
  - Line 1: `L` — length of the reference string.
  - Line 2: `L` integers — page numbers (≥ 0).
- Simulate the following algorithms independently:
  - `FIFO`
  - `LRU`
  - `OPT` (Belady's optimal)
- For each algorithm, output exactly:

```
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.

Solution Implementation

```
1 # 18
2 import sys
```

```python
import argparse
from typing import List

def print_result(alg: str, faults: int, frames: List[int]) -> None:
    print(f"ALG {alg}")
    print(f"OK: FAULTS {faults}")
    print("OK: FINAL " + " ".join(str(f) if f != -1 else "-1" for f in
        frames))

def simulate_fifo(refs: List[int], F: int) -> tuple[int, List[int]]:
    """FIFO with circular pointer implementation."""
    frames = [-1] * F
    ptr = 0  # Pointer to next frame to replace
    faults = 0
    occupied = 0  # How many frames are actually occupied

    for page in refs:
        if page in frames:
            continue  # Hit

        faults += 1  # Miss

        if occupied < F:
            # Fill empty frame
            frames[occupied] = page
            occupied += 1
        else:
            # Replace at pointer
            frames[ptr] = page
            ptr = (ptr + 1) % F

    return faults, frames

def simulate_lru(refs: List[int], F: int) -> tuple[int, List[int]]:
    """LRU with timestamp tracking."""
    frames = [-1] * F
    last_used = {}  # page -> last access time
    time = 0
```

```
    faults = 0
    occupied = 0

    for page in refs:
        if page in frames:
            # Update last used time
            last_used[page] = time
        else:
            faults += 1

            if occupied < F:
                # Fill first empty frame
                frames[occupied] = page
                last_used[page] = time
                occupied += 1
            else:
                # Find LRU page in frames
                lru_page = frames[0]
                lru_time = last_used.get(lru_page, float('inf'))

                for i in range(1, F):
                    current_page = frames[i]
                    current_time = last_used.get(current_page, float('
                        inf'))
                    if current_time < lru_time:
                        lru_page = current_page
                        lru_time = current_time

                # Replace LRU page
                idx = frames.index(lru_page)
                del last_used[lru_page]
                frames[idx] = page
                last_used[page] = time

        time += 1

    return faults, frames

```

```python
def simulate_opt(refs: List[int], F: int) -> tuple[int, List[int]]:
    """Optimal page replacement."""
    frames = [-1] * F
    faults = 0
    occupied = 0

    for i, page in enumerate(refs):
        if page in frames:
            continue

        faults += 1

        if occupied < F:
            # Fill empty frame
            frames[occupied] = page
            occupied += 1
        else:
            # Find page to evict
            victim_idx = 0
            farthest_next = -1

            for idx, frame_page in enumerate(frames):
                # Find next use of this page
                next_use = len(refs) + 1  # Default: never used again
                for j in range(i + 1, len(refs)):
                    if refs[j] == frame_page:
                        next_use = j
                        break

                if next_use == len(refs) + 1:
                    # Never used again - evict this one immediately
                    victim_idx = idx
                    break
                elif next_use > farthest_next:
                    farthest_next = next_use
                    victim_idx = idx
                elif next_use == farthest_next and idx < victim_idx:
                    # Tie-break: smaller frame index
```

```
                    victim_idx = idx

            frames[victim_idx] = page

    return faults, frames

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--frames', type=int, required=True)
    args = parser.parse_args()

    if args.frames < 1 or args.frames > 64:
        print("ERROR: E_RANGE: frames must be 1..64", file=sys.stderr)
        return 1

    try:
        L_line = sys.stdin.readline()
        if not L_line:
            return 1
        L = int(L_line.strip())

        refs_line = sys.stdin.readline()
        if not refs_line:
            return 1
        refs = list(map(int, refs_line.strip().split()))

        if len(refs) != L:
            print("ERROR: E_INPUT: length mismatch", file=sys.stderr)
            return 1

        if any(x < 0 for x in refs):
            print("ERROR: E_RANGE: page numbers must be >= 0", file=sys
                .stderr)
            return 1
    except Exception:
        print("ERROR: E_INPUT: invalid input", file=sys.stderr)
        return 1

```

```
    # Run all algorithms
    f_faults, f_frames = simulate_fifo(refs, args.frames)
    print_result("FIFO", f_faults, f_frames)

    l_faults, l_frames = simulate_lru(refs, args.frames)
    print_result("LRU", l_faults, l_frames)

    o_faults, o_frames = simulate_opt(refs, args.frames)
    print_result("OPT", o_faults, o_frames)

    return 0

if __name__ == "__main__":
    sys.exit(main())
```

Sample Input

```
printf '12
1 2 3 4 1 2 5 1 2 3 4 5
' | python3 pagerepl.py --frames 3
```

Sample Output

```
ALG FIFO
OK: FAULTS 9
OK: FINAL 5 3 4
ALG LRU
OK: FAULTS 10
OK: FINAL 3 4 5
ALG OPT
OK: FAULTS 7
OK: FINAL 4 2 5
```

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

Problem Statement

- Implement a CLI tool named `filealloc`.
- Input is provided via `stdin` in the following format:
  - Line 1: `N` — total number of disk blocks (1..10000).
  - Line 2: `F` — number of free blocks.
  - Line 3: `F` integers — free block IDs (0..N-1, unique).
  - Line 4: `M` — number of files.
  - Next `M` lines: `name size`, where `size` is blocks required (1..N).
- Simulate the following allocation strategies independently, each starting from the original free list:
  - `CONTIGUOUS`
  - `LINKED`
  - `INDEXED`
- Output per algorithm:

```
ALG <name>
FILE <name> -> <map>
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.

Solution Implementation

```
1 # 19
```

```python
import sys
from typing import List, Tuple, Optional, Set

def error_exit(code: str, message: str) -> None:
    """Print error message and exit with non-zero status."""
    print(f"ERROR: {code}: {message}", file=sys.stderr)
    sys.exit(1)

def read_input() -> Tuple[int, Set[int], List[Tuple[str, int]]]:
    """Read and validate input from stdin."""
    try:
        # Read total blocks
        line = sys.stdin.readline()
        if not line:
            error_exit("E_INPUT", "empty input")
        N = int(line.strip())
        if not (1 <= N <= 10000):
            error_exit("E_RANGE", "total blocks must be 1..10000")

        # Read number of free blocks
        line = sys.stdin.readline()
        if not line:
            error_exit("E_INPUT", "missing free blocks count")
        F = int(line.strip())

        # Read free block IDs
        line = sys.stdin.readline()
        if not line:
            error_exit("E_INPUT", "missing free block list")
        free_blocks = list(map(int, line.strip().split()))

        if len(free_blocks) != F:
            error_exit("E_INPUT", f"free block count mismatch: expected
                {F}, got {len(free_blocks)}")

        # Validate free blocks
        seen = set()
        for block in free_blocks:
```

```
            if not (0 <= block < N):
                error_exit("E_RANGE", f"block ID {block} out of range 0..{N-1}")
            if block in seen:
                error_exit("E_DUPBLOCK", "free block list must contain unique IDs")
            seen.add(block)

        # Read number of files
        line = sys.stdin.readline()
        if not line:
            error_exit("E_INPUT", "missing file count")
        M = int(line.strip())

        # Read file requests
        files = []
        for _ in range(M):
            line = sys.stdin.readline()
            if not line:
                error_exit("E_INPUT", "incomplete file list")
            parts = line.strip().split()
            if len(parts) != 2:
                error_exit("E_INPUT", "invalid file line format")

            name = parts[0]
            try:
                size = int(parts[1])
            except ValueError:
                error_exit("E_INPUT", f"invalid file size for {name}")

            if not (1 <= size <= N):
                error_exit("E_RANGE", f"file size for {name} must be 1..{N}")

            files.append((name, size))

        return N, set(free_blocks), files

```

```
    except ValueError:
        error_exit("E_INPUT", "invalid number format")

def simulate_contiguous(N: int, free_blocks: Set[int], files: List[Tuple[str, int]]) -> List[Tuple[str, str]]:
    """Simulate contiguous allocation strategy."""
    results = []
    # Create a sorted list of free blocks for easier contiguous search
    sorted_free = sorted(free_blocks)

    for name, size in files:
        allocated = False

        # Find smallest starting block with enough consecutive free blocks
        for i in range(len(sorted_free) - size + 1):
            start = sorted_free[i]
            # Check if blocks start..start+size-1 are all free
            consecutive = True
            for j in range(1, size):
                if sorted_free[i + j] != start + j:
                    consecutive = False
                    break

            if consecutive:
                # Allocation successful
                results.append((name, f"START {start} LEN {size}"))

                # Remove allocated blocks from free list
                for j in range(size):
                    sorted_free.remove(start + j)

                allocated = True
                break

        if not allocated:
            results.append((name, "FAIL"))

```

```
110     return results
111
112 def simulate_linked(N: int, free_blocks: Set[int], files: List[Tuple[
    str, int]]) -> List[Tuple[str, str]]:
113     """Simulate linked allocation strategy."""
114     results = []
115     # Use sorted list for deterministic smallest block selection
116     available_blocks = sorted(free_blocks)
117
118     for name, size in files:
119         if len(available_blocks) >= size:
120             # Allocate smallest 'size' blocks
121             allocated = available_blocks[:size]
122
123             # Create chain string
124             chain = "->".join(str(b) for b in allocated)
125             results.append((name, f"CHAIN {chain}"))
126
127             # Remove allocated blocks
128             available_blocks = available_blocks[size:]
129         else:
130             results.append((name, "FAIL"))
131
132     return results
133
134 def simulate_indexed(N: int, free_blocks: Set[int], files: List[Tuple[
    str, int]]) -> List[Tuple[str, str]]:
135     """Simulate indexed allocation strategy."""
136     results = []
137     # Use sorted list for deterministic smallest block selection
138     available_blocks = sorted(free_blocks)
139
140     for name, size in files:
141         # Indexed allocation needs: 1 index block + size data blocks
142         total_needed = size + 1
143
144         if len(available_blocks) >= total_needed:
```

```
145             # Allocate blocks: first block is index, next 'size' blocks
                    are data
146             index_block = available_blocks[0]
147             data_blocks = available_blocks[1:size+1]
148 
149             # Create data blocks string
150             data_str = ",".join(str(b) for b in data_blocks)
151             results.append((name, f"INDEX {index_block} DATA {data_str}
                    "))
152 
153             # Remove allocated blocks
154             available_blocks = available_blocks[total_needed:]
155         else:
156             results.append((name, "FAIL"))
157 
158     return results
159 
160 def main() -> None:
161     """Main function implementing filealloc tool."""
162     # Read and validate input
163     N, free_blocks, files = read_input()
164 
165     # Simulate each strategy independently
166     print("ALG CONTIGUOUS")
167     contiguous_results = simulate_contiguous(N, free_blocks.copy(),
            files)
168     for name, result in contiguous_results:
169         print(f"FILE {name} -> {result}")
170 
171     print("\nALG LINKED")
172     linked_results = simulate_linked(N, free_blocks.copy(), files)
173     for name, result in linked_results:
174         print(f"FILE {name} -> {result}")
175 
176     print("\nALG INDEXED")
177     indexed_results = simulate_indexed(N, free_blocks.copy(), files)
178     for name, result in indexed_results:
179         print(f"FILE {name} -> {result}")
```

```
180
181 if __name__ == "__main__":
182     main()
```

Sample Input

```
printf '20
10
1 2 3 4 5 7 8 10 11 12
3
A 3
B 4
C 6
' | python3 filealloc.py
```

Sample Output

```
ALG CONTIGUOUS
FILE A -> START 1 LEN 3
FILE B -> FAIL
FILE C -> FAIL

ALG LINKED
FILE A -> CHAIN 1->2->3
FILE B -> CHAIN 4->5->7->8
FILE C -> FAIL

ALG INDEXED
FILE A -> INDEX 1 DATA 2,3,4
FILE B -> INDEX 5 DATA 7,8,10,11
FILE C -> FAIL
```

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

Problem Statement

- Implement a CLI tool named `disksched`.
- Command-line arguments:
  - `--max <C>` — maximum cylinder number ($C \geq 1$).
  - `--start <S>` — starting head position ($0 \leq S \leq C$).
  - `--dir left|right` — initial scan direction (used for SCAN and C-SCAN).
- Input via `stdin` in the following format:
  - Line 1: `L` — number of disk requests.
  - Line 2: `L` integers — requested cylinder numbers ($0..C$).
- Simulate the following algorithms independently:
  - `FCFS`
  - `SSTF`
  - `SCAN`
  - `C-SCAN`
- For each algorithm, output exactly:

```
ALG <name>
OK: ORDER <c1> <c2> ... <cL>
OK: MOVES <m>
```

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

Solution Implementation

```
1
2  import sys
3  import argparse
4  from typing import List
5
6  def error_exit(code: str, message: str):
7      print(f"ERROR: {code}: {message}", file=sys.stderr)
8      sys.exit(1)
9
10 class DiskScheduler:
11     def __init__(self, max_cylinder: int, start: int, direction: str):
12         self.max_cylinder = max_cylinder
13         self.start = start
14         self.direction = direction  # 'left' or 'right'
15
16     def fcfs(self, requests: List[int]) -> tuple[List[int], int]:
17         """First-Come, First-Served."""
18         order = requests.copy()
19         moves = self._calculate_moves(order)
20         return order, moves
21
22     def sstf(self, requests: List[int]) -> tuple[List[int], int]:
23         """Shortest Seek Time First."""
24         order = []
25         remaining = requests.copy()
26         current = self.start
27
28         while remaining:
29             # Find closest request
30             closest_idx = 0
31             closest_dist = abs(remaining[0] - current)
32
33             for i in range(1, len(remaining)):
34                 dist = abs(remaining[i] - current)
35                 if dist < closest_dist or (dist == closest_dist and
                       remaining[i] < remaining[closest_idx]):
```

```
                    closest_dist = dist
                    closest_idx = i

            order.append(remaining[closest_idx])
            current = remaining[closest_idx]
            remaining.pop(closest_idx)

        moves = self._calculate_moves(order)
        return order, moves

    def scan(self, requests: List[int]) -> tuple[List[int], int]:
        """SCAN (Elevator) algorithm."""
        order = []
        current = self.start

        if self.direction == 'right':
            # Moving right first
            right_requests = [r for r in requests if r >= current]
            right_requests.sort()
            order.extend(right_requests)

            # Then move to max cylinder if needed
            if right_requests:
                order.append(self.max_cylinder)

            # Then serve left side
            left_requests = [r for r in requests if r < current]
            left_requests.sort(reverse=True)
            order.extend(left_requests)

        else:  # left
            # Moving left first
            left_requests = [r for r in requests if r <= current]
            left_requests.sort(reverse=True)
            order.extend(left_requests)

            # Then move to 0 if needed
            if left_requests:
```

```
74                  order.append(0)
75
76              # Then serve right side
77              right_requests = [r for r in requests if r > current]
78              right_requests.sort()
79              order.extend(right_requests)
80
81          # Remove duplicates of end cylinders if they weren't in
               original requests
82          # (We added them to simulate going to the end)
83          final_order = []
84          for cyl in order:
85              if cyl in requests or cyl == 0 or cyl == self.max_cylinder:
86                  final_order.append(cyl)
87                  if cyl in requests:
88                      # Remove one occurrence from requests list
89                      idx = requests.index(cyl)
90                      requests.pop(idx)
91
92          moves = self._calculate_moves(final_order)
93          return final_order, moves
94
95      def cscan(self, requests: List[int]) -> tuple[List[int], int]:
96          """C-SCAN (Circular SCAN) algorithm."""
97          order = []
98          current = self.start
99
100         if self.direction == 'right':
101             # Moving right first
102             right_requests = [r for r in requests if r >= current]
103             right_requests.sort()
104             order.extend(right_requests)
105
106             # Go to max cylinder
107             if right_requests:
108                 order.append(self.max_cylinder)
109
110             # Jump to 0 (counts as movement)
```

```
            order.append(0)

            # Then serve remaining left side
            left_requests = [r for r in requests if r < current]
            left_requests.sort()
            order.extend(left_requests)

        else:  # left
            # Moving left first
            left_requests = [r for r in requests if r <= current]
            left_requests.sort(reverse=True)
            order.extend(left_requests)

            # Go to 0
            if left_requests:
                order.append(0)

            # Jump to max cylinder (counts as movement)
            order.append(self.max_cylinder)

            # Then serve remaining right side
            right_requests = [r for r in requests if r > current]
            right_requests.sort(reverse=True)
            order.extend(right_requests)

        # Remove end cylinders if they weren't in original requests
        final_order = []
        temp_requests = requests.copy()
        for cyl in order:
            if cyl in temp_requests or cyl == 0 or cyl == self.
               max_cylinder:
                final_order.append(cyl)
                if cyl in temp_requests:
                    idx = temp_requests.index(cyl)
                    temp_requests.pop(idx)

        moves = self._calculate_moves(final_order)
        return final_order, moves
```

```
148
149     def _calculate_moves(self, order: List[int]) -> int:
150         """Calculate total head movement for service order."""
151         total = 0
152         current = self.start
153         for cyl in order:
154             total += abs(cyl - current)
155             current = cyl
156         return total
157
158 def main():
159     parser = argparse.ArgumentParser()
160     parser.add_argument('--max', type=int, required=True)
161     parser.add_argument('--start', type=int, required=True)
162     parser.add_argument('--dir', choices=['left', 'right'], required=
        True)
163
164     args = parser.parse_args()
165
166     # Validate
167     if args.max < 1:
168         error_exit('E_RANGE', 'max cylinder must be >= 1')
169     if not (0 <= args.start <= args.max):
170         error_exit('E_RANGE', f'start must be 0..{args.max}')
171
172     # Read input
173     try:
174         line = sys.stdin.readline()
175         if not line:
176             error_exit('E_INPUT', 'empty input')
177         L = int(line.strip())
178
179         line = sys.stdin.readline()
180         if not line:
181             error_exit('E_INPUT', 'missing requests')
182         requests = list(map(int, line.strip().split()))
183
184         if len(requests) != L:
```

```
            error_exit('E_INPUT', f'expected {L} requests, got {len(requests)}')

        for req in requests:
            if not (0 <= req <= args.max):
                error_exit('E_RANGE', f'request {req} out of range 0..{args.max}')

    except ValueError:
        error_exit('E_INPUT', 'invalid number format')

    # Create scheduler and run algorithms
    scheduler = DiskScheduler(args.max, args.start, args.dir)

    # FCFS
    order, moves = scheduler.fcfs(requests.copy())
    print("ALG FCFS")
    print(f"OK: ORDER {' '.join(map(str, order))}")
    print(f"OK: MOVES {moves}")

    # SSTF
    order, moves = scheduler.sstf(requests.copy())
    print("\nALG SSTF")
    print(f"OK: ORDER {' '.join(map(str, order))}")
    print(f"OK: MOVES {moves}")

    # SCAN
    order, moves = scheduler.scan(requests.copy())
    print("\nALG SCAN")
    print(f"OK: ORDER {' '.join(map(str, order))}")
    print(f"OK: MOVES {moves}")

    # C-SCAN
    order, moves = scheduler.cscan(requests.copy())
    print("\nALG C-SCAN")
    print(f"OK: ORDER {' '.join(map(str, order))}")
    print(f"OK: MOVES {moves}")

```

```
221     return 0
222
223 if __name__ == "__main__":
224     sys.exit(main())
```

Sample Input

```
printf '8
98 183 37 122 14 124 65 67
' | python3 disksched.py --max 199 --start 53 --dir right
```

Sample Output

```
ALG FCFS
OK: ORDER 98 183 37 122 14 124 65 67
OK: MOVES 640

ALG SSTF
OK: ORDER 65 67 37 14 98 122 124 183
OK: MOVES 236

ALG SCAN
OK: ORDER 65 67 98 122 124 183 199 37 14
OK: MOVES 331

ALG C-SCAN
OK: ORDER 65 67 98 122 124 183 199 0 14 37
OK: MOVES 382
```