COMPREHENSIVE STUDY NOTES FOR SYSTEMS PROGRAMMING EXAM

TABLE OF CONTENTS

---

1. INTRODUCTION TO SYSTEMS PROGRAMMING

Systems programming involves writing software that interacts directly with the operating system kernel. You'll work with fundamental OS concepts like files, processes, memory, and IPC, not through high-level libraries, but through direct **system calls**.

Key Concepts:

- **File Descriptor (fd)**: A non-negative integer that the kernel uses to represent an open file, socket, pipe, or other I/O resource. `0` is stdin, `1` is stdout, `2` is stderr.
- **System Calls**: The interface between a user-space program and the kernel. Functions like `open()`, `read()`, `fork()` are wrappers around these calls. Always check their return values for errors (`-1` is a common error indicator).
- **Processes**: Running instances of programs
- **Signals**: Asynchronous notifications to processes
- `perror(const char *s)`: When a system call fails, it sets a global variable `errno`. `perror` prints the string you provide, followed by a colon, a space, and a human-readable description of the current `errno` value. **Always use this for debugging.**

---

2. FILE OPERATIONS

2.1 Basic File I/O

Opening Files

```c
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Flags:

- `O_RDONLY`: Read only
- `O_WRONLY`: Write only
- `O_RDWR`: Read and write
- `O_CREAT`: Create file if it doesn't exist
- `O_TRUNC`: Truncate file to zero length
- `O_APPEND`: Append mode

Flag Concatenation Rules:

- Use bitwise OR (`|`) to combine flags: `O_CREAT | O_WRONLY | O_TRUNC`
- Common combinations:
  - `O_CREAT | O_WRONLY | O_TRUNC`: Create new file, truncate if exists
  - `O_CREAT | O_WRONLY | O_APPEND`: Create new file, append if exists
  - `O_CREAT | O_RDWR | O_EXCL`: Create new file, fail if exists

Reading and Writing

```c
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

Important: Returns the number of bytes read/written, `0` on end-of-file (for `read`), or `-1` on error. **A `read`/`write` might not process all `count` bytes, so always use a loop for complete I/O.**

Small Example: Creating and Writing to a File

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main() {
    // Concatenate flags with bitwise OR (|)
    int fd = open("example.txt", O_CREAT | O_WRONLY | O_TRUNC, 0644);
    if (fd == -1) {
        perror("open failed");
        return 1;
```

```
    }

    const char *message = "Hello, World!\n";
    ssize_t bytes_written = write(fd, message, strlen(message));
    if (bytes_written == -1) {
        perror("write failed");
        close(fd);
        return 1;
    }

    printf("Successfully wrote %zd bytes\n", bytes_written);
    close(fd);

    // Now read it back
    fd = open("example.txt", O_RDONLY);
    if (fd == -1) {
        perror("open for reading failed");
        return 1;
    }

    char buffer[100];
    ssize_t bytes_read = read(fd, buffer, sizeof(buffer) - 1);
    if (bytes_read > 0) {
        buffer[bytes_read] = '\0';  // Null-terminate
        printf("Read: %s", buffer);
    }

    close(fd);
    return 0;
}
```

Example: Safe Read Loop

```
ssize_t read_all(int fd, void *buf, size_t count) {
    char *ptr = (char *)buf;
    size_t total = 0;

    while (total < count) {
        ssize_t n = read(fd, ptr + total, count - total);
        if (n <= 0) break; // EOF or error
        total += n;
    }
```

```
    return total;
}
```

Positioned I/O

```
ssize_t pread(int fd, void *buf, size_t count, off_t offset);
ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);
```

Key Advantages:

- **pread/pwrite**: Read/write at specific offset without changing file position
- **Thread-safe**: Unlike `lseek` + `read`, these are atomic operations
- **Exam Pattern (`2023-02-16 - pcp`)**: Perfect for parallel file processing where multiple processes need to read/write specific chunks of a file without interfering with each other's file offsets.

File Closing

```
#include <unistd.h>
int close(int fd);
```

Always close file descriptors to prevent resource leaks.

2.2 Advanced File Operations

File Seeking

```
off_t lseek(int fd, off_t offset, int whence);
```

whence values:

- `SEEK_SET`: From beginning
- `SEEK_CUR`: From current position
- `SEEK_END`: From end

Creating Sparse Files

A **sparse file** contains "holes" - regions that read as zeros but don't occupy disk space.

Example:

```c
int fd = open("sparse", O_CREAT | O_WRONLY, 0644);
lseek(fd, 1000000000, SEEK_SET);  // Create 1GB hole
write(fd, "x", 1);                 // File appears 1GB but uses minimal space
```

2.3 File Metadata

stat/fstat/lstat

```c
#include <sys/stat.h>
int stat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
```

Differences:

- `stat`: Follows symbolic links
- `lstat`: Doesn't follow symbolic links
- `fstat`: Uses file descriptor

struct stat important fields:

- `st_mode`: File type (use `S_ISREG`, `S_ISDIR`, `S_ISLNK` macros) and permissions
- `st_size`: Total size in bytes
- `st_nlink`: Number of hard links
- `st_ino`: **Inode number** - All hard links to a file share the same inode
- `st_uid`/`st_gid`: User ID and Group ID of the owner
- `st_mtime`: Last modification time

File and Directory Links

Hard Link (`link()`): A new directory entry pointing to an existing inode.

```c
int link(const char *oldpath, const char *newpath);
```

Symbolic (Soft) Link (`symlink()`): A special file whose content is the path to another file.

```c
int symlink(const char *target, const char *linkpath);
```

Reading Links:

```c
ssize_t readlink(const char *pathname, char *buf, size_t bufsiz);
char *realpath(const char *path, char *resolved_path);
```

Exam Pattern (2023-09-14 - Python/bash symlinks): To find symlinks pointing to the same target file, you can `realpath()` each link and compare the resulting strings, or `lstat()` the link, `readlink()` its target, build the absolute path, and then `stat()` the target to get its `st_ino`. Comparing inodes is most robust.

Common File Operation Patterns:

File Inversion (Byte Reversal)

```c
void invert_file(const char *input, const char *output) {
    struct stat st;
    stat(input, &st);
    int in_fd = open(input, O_RDONLY);
    int out_fd = open(output, O_WRONLY | O_CREAT, 0644);

    char buffer[4096];
    for (off_t pos = st.st_size - 4096; pos >= 0; pos -= 4096) {
        ssize_t n = pread(in_fd, buffer, 4096, pos);
        // Reverse buffer contents
        for (int i = 0; i < n/2; i++) {
            char tmp = buffer[i];
            buffer[i] = buffer[n-1-i];
            buffer[n-1-i] = tmp;
        }
        write(out_fd, buffer, n);
    }
}
```

---

3. PROCESS MANAGEMENT

3.1 Process Creation

fork()

```c
#include <unistd.h>
pid_t fork(void);
```

Return values:

- **In Parent**: Returns the PID of the new child
- **In Child**: Returns `0`
- **On Error**: Returns `-1`

Creates exact copy of parent process

exec family

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *filename, char *const argv[], char *const envp[]);
```

Key differences:

- `l` vs `v`: List vs vector (array) of arguments - Arguments as a `l`ist vs. a `v`ector (array)
- `p`: Searches PATH
- `e`: Custom environment - Takes an explicit `e`nvironment array

Important: exec functions **do not return on success** - they replace the current process image completely.

Exam Pattern (`2023-01-19 - stdin2pipe`): The classic `fork()` then `execvp()` pattern is essential for launching commands.

3.2 Process Synchronization

wait/waitpid

```
#include <sys/wait.h>
pid_t wait(int *wstatus);
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

Status macros:

- `WIFEXITED(status)`: True if normal termination
- `WEXITSTATUS(status)`: Exit status (0-255) if `WIFEXITED` is true
- `WIFSIGNALED(status)`: True if terminated by signal

- `WTERMSIG(status)`: Signal number if `WIFSIGNALED` is true

Exam Pattern (2022-09-07 - rilancia): Required checking `WIFEXITED` and `WEXITSTATUS` to decide whether to re-launch a program.

3.3 Process Attributes

prctl()

```
#include <sys/prctl.h>
int prctl(int option, unsigned long arg2, ...);
```

Important options:

- `PR_SET_NAME`: Set process name (appears in `ps` or `top`)
- `PR_SET_CHILD_SUBREAPER`: If set to 1, this process becomes the new parent for any of its descendant processes that become orphaned (instead of `init`/`systemd`). This allows a "grandparent" to `wait()` for its "grandchildren".

Exam Pattern (2019-06-19): These options are commonly tested for process hierarchy management.

---

4. INTER-PROCESS COMMUNICATION (IPC)

4.1 Pipes

Anonymous Pipes

```
int pipe(int pipefd[2]);
```

- `pipefd[0]`: Read end
- `pipefd[1]`: Write end

File Descriptor Duplication:

```
int dup2(int oldfd, int newfd);
```

Duplicates a file descriptor, making the new one an exact copy of the old one.

Example: Parent-Child Communication

```c
int pipefd[2];
pipe(pipefd);
if (fork() == 0) {
    close(pipefd[1]);  // Child closes write end
    char buf[100];
    read(pipefd[0], buf, 100);
} else {
    close(pipefd[0]);  // Parent closes read end
    write(pipefd[1], "Hello", 5);
}
```

Exam Pattern (2023-01-19 - stdin2pipe): This is the canonical problem for pipes. To make a command's output the input for another (`cmd1 | cmd2`):

1. `pipe(pipefd)`
2. `fork()` for `cmd1`. In child 1:
    - `close(pipefd[0])` (don't need to read)
    - `dup2(pipefd[1], STDOUT_FILENO)` (redirect stdout to pipe's write end)
    - `close(pipefd[1])`
    - `exec("cmd1")`
3. `fork()` for `cmd2`. In child 2:
    - `close(pipefd[1])` (don't need to write)
    - `dup2(pipefd[0], STDIN_FILENO)` (redirect stdin to pipe's read end)
    - `close(pipefd[0])`
    - `exec("cmd2")`
4. Parent closes both `pipefd[0]` and `pipefd[1]` and `wait()`s for children.

4.2 Named Pipes (FIFOs)

```c
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

Usage pattern:

```c
mkfifo("/tmp/myfifo", 0666);
// Process 1: writer
int fd = open("/tmp/myfifo", O_WRONLY);
write(fd, data, size);
```

```
// Process 2: reader
int fd = open("/tmp/myfifo", O_RDONLY);
read(fd, buffer, size);
```

Exam Pattern (2023-01-23 - `fifotext`): A server process creates a FIFO. It `open()`s the FIFO for reading. Client processes `open()` the same FIFO for writing to send data. The server reads data, processes it, and might loop to re-open the FIFO to accept more connections if it's closed by the writer.

4.3 System V Message Queues

```
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

Message structure:

```
struct msgbuf {
    long mtype;     // Message type
    char mtext[1];  // Message data
};
```

4.4 POSIX Shared Memory

```
#include <sys/mman.h>
int shm_open(const char *name, int oflag, mode_t mode);
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

---

5. SIGNAL HANDLING

5.1 Traditional Signals

Common Signals:

- `SIGINT` (2): Interrupt (Ctrl+C)

- `SIGTERM` (15): Termination request
- `SIGKILL` (9): Force kill (cannot be caught)
- `SIGUSR1` (10): User-defined signal 1
- `SIGUSR2` (12): User-defined signal 2
- `SIGCHLD` (17): Child status changed
- `SIGHUP` (1): Hangup

signal() - Simple Handler

```c
#include <signal.h>
void handler(int sig) {
    printf("Received signal %d\n", sig);
}
signal(SIGUSR1, handler);
```

5.2 signalfd - Modern Approach

A modern, safer way to handle signals synchronously. It creates a file descriptor from which you can `read()` signal information, avoiding the pitfalls of async-signal-safe functions.

```c
#include <sys/signalfd.h>
int signalfd(int fd, const sigset_t *mask, int flags);
```

Steps to use signalfd:

1. **Create a signal set (`sigset_t`)**: Use `sigemptyset()` and `sigaddset()` to specify which signals to catch.
2. **Block the signals**: Use `sigprocmask(SIG_BLOCK, ...)` to prevent the default handlers from running.
3. **Create the `signalfd`.**
4. **`read()` from the `signalfd`**: This call will block until a signal in the set is delivered. The data read is a `struct signalfd_siginfo`.

Example:

```c
sigset_t mask;
sigemptyset(&mask);
sigaddset(&mask, SIGUSR1);
sigaddset(&mask, SIGUSR2);
```

```
sigprocmask(SIG_BLOCK, &mask, NULL);

int sfd = signalfd(-1, &mask, 0);
struct signalfd_siginfo si;
read(sfd, &si, sizeof(si));
printf("Got signal %d from PID %d\n", si.ssi_signo, si.ssi_pid);
```

Exam Pattern (2018-07-18 - `signal counter`): A program sets up a `signalfd` for `SIGUSR1` and `SIGUSR2`. In a loop, it reads from the `signalfd` and increments/decrements a counter based on which signal was received.

5.3 Signal Queuing with sigqueue

Allows you to send a signal along with a small integer or a pointer value.

```
#include <signal.h>
int sigqueue(pid_t pid, int sig, const union sigval value);
```

Sending data with signals:

```
union sigval value;
value.sival_int = 42;
sigqueue(target_pid, SIGUSR1, value);
```

Note: The receiving process must use `sigaction` with the `SA_SIGINFO` flag to access this data in its signal handler.

Exam Pattern (2022-06-22 - `tx/rx`): This is a very specific and advanced pattern. A `tx` program sends a string to an `rx` program 8 characters at a time. It encodes each character into an integer and sends it via `sigqueue`. The `rx` program's signal handler receives the integer, reconstructs the character, and appends it to a buffer.

---

6. DIRECTORY OPERATIONS

Basic Directory Reading

```
#include <dirent.h>
DIR *opendir(const char *name);
```

```
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);
```

struct dirent:

- `d_name`: Filename
- `d_ino`: Inode number
- `d_type`: File type (not always reliable)

Important: Remember to ignore `.` and `..` entries in your loops.

Advanced Directory Traversal

Using openat and fdopendir

These functions operate relative to a directory file descriptor, which helps avoid race conditions and simplifies handling of long paths.

Exam Pattern (`2022-07-22 - tree v2`): This question *specifically* required implementing a recursive directory traversal using `openat()` and `fdopendir()` *without* concatenating path strings:

1. `opendir()` the top-level directory
2. Loop with `readdir()`
3. If an entry is a directory, use `openat(dir_fd, entry->d_name, O_RDONLY | O_DIRECTORY)` to get a file descriptor for the subdirectory
4. Use `fdopendir()` on that new descriptor to get a `DIR*` stream for the recursive call

```
#include <fcntl.h>
int openat(int dirfd, const char *pathname, int flags);
DIR *fdopendir(int fd);
```

Directory Traversal Pattern

```
void list_directory(const char *path) {
    DIR *dir = opendir(path);
    struct dirent *entry;

    while ((entry = readdir(dir)) != NULL) {
        if (strcmp(entry->d_name, ".") == 0 ||
            strcmp(entry->d_name, "..") == 0)
```

```
            continue;
        printf("%s\n", entry->d_name);
    }
    closedir(dir);
}
```

Sorting Directory Contents

```
int compare(const void *a, const void *b) {
    return strcmp(*(char **)a, *(char **)b);
}

void sorted_directory_list(const char *path) {
    // Count entries
    DIR *dir = opendir(path);
    int count = 0;
    while (readdir(dir)) count++;
    rewinddir(dir);

    // Allocate array
    char **names = malloc(count * sizeof(char*));

    // Fill array
    struct dirent *entry;
    int i = 0;
    while ((entry = readdir(dir))) {
        names[i++] = strdup(entry->d_name);
    }

    // Sort and print
    qsort(names, count, sizeof(char*), compare);
    for (i = 0; i < count; i++) {
        printf("%s\n", names[i]);
    }
}
```

7. FILE SYSTEM MONITORING - inotify

A Linux kernel subsystem that provides notifications about filesystem events.

inotify API

```
#include <sys/inotify.h>
int inotify_init(void);
int inotify_add_watch(int fd, const char *pathname, uint32_t mask);
int inotify_rm_watch(int fd, int wd);
```

Steps to use inotify:

- `inotify_init()`: Creates an inotify instance, returns an fd
- `inotify_add_watch()`: Adds a watch for events on a specific path
- `read()` **from the inotify fd**: This blocks until events occur. You read into a buffer and cast the contents to `struct inotify_event`

Event Masks:

- `IN_CREATE`: File/directory created
- `IN_DELETE`: File/directory deleted
- `IN_MODIFY`: File modified
- `IN_OPEN`: File opened
- `IN_ACCESS`: File accessed
- `IN_CLOSE_WRITE`: Writable file closed

Exam Pattern (`2021-09-16 - execname`): A program uses `inotify` to watch a directory. When a new file is created (`IN_CREATE` event), the program reads the event to get the filename, forks, and `exec`s a command using that filename, then deletes the file.

inotify Example

```
int fd = inotify_init();
int wd = inotify_add_watch(fd, "/tmp", IN_CREATE | IN_DELETE);

char buffer[4096];
while (1) {
```

```
    int len = read(fd, buffer, sizeof(buffer));
    struct inotify_event *event = (struct inotify_event *)buffer;

    if (event->mask & IN_CREATE)
        printf("Created: %s\n", event->name);
    if (event->mask & IN_DELETE)
        printf("Deleted: %s\n", event->name);
}
```

---

8. THREADING

pthread Basics

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
int pthread_join(pthread_t thread, void **retval);
```

Exam Pattern (2019-05-29 - producer/consumer): This question required using two threads synchronized with an `eventfd` in semaphore mode (`EFD_SEMAPHORE`). The producer writes a value to shared memory and then writes `1` to the `eventfd` to "signal". The consumer `read()`s from the `eventfd` to "wait" and then consumes the value.

Thread Synchronization with eventfd

```
#include <sys/eventfd.h>
int eventfd(unsigned int initval, int flags);
```

Flags:

- `EFD_SEMAPHORE`: Semaphore semantics
- `EFD_CLOEXEC`: Close on exec
- `EFD_NONBLOCK`: Non-blocking I/O

Producer-Consumer Example:

```c
int efd = eventfd(0, EFD_SEMAPHORE);

void *producer(void *arg) {
    while (1) {
        int value = rand();
        shared_data = value;
        uint64_t u = 1;
        write(efd, &u, sizeof(u));  // Signal consumer
        sleep(rand() % 3);
    }
}

void *consumer(void *arg) {
    while (1) {
        uint64_t u;
        read(efd, &u, sizeof(u));   // Wait for signal
        printf("Consumed: %d\n", shared_data);
        sleep(rand() % 3);
    }
}
```

---

9. EVENT-DRIVEN PROGRAMMING

9.1 poll()

```c
#include <poll.h>
int poll(struct pollfd *fds, nfds_t nfds, int timeout);

struct pollfd {
    int   fd;      // File descriptor
    short events;  // Requested events
    short revents; // Returned events
};
```

Events:

- `POLLIN`: Data available to read

- POLLOUT: Writing possible
- POLLERR: Error condition
- POLLHUP: Hang up

9.2 select()

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

FD_SET operations:

```
FD_ZERO(&readfds);     // Clear set
FD_SET(fd, &readfds); // Add fd to set
FD_CLR(fd, &readfds); // Remove fd from set
FD_ISSET(fd, &readfds); // Test if fd is in set
```

9.3 epoll() - Linux specific

```
int epoll_create(int size);
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

9.4 timerfd

```
#include <sys/timerfd.h>
int timerfd_create(int clockid, int flags);
int timerfd_settime(int fd, int flags, const struct itimerspec *new_value,
                    struct itimerspec *old_value);
```

Timer Example:

```
int tfd = timerfd_create(CLOCK_MONOTONIC, 0);
struct itimerspec ts;
ts.it_interval.tv_sec = 1;  // Repeat every 1 second
ts.it_interval.tv_nsec = 0;
ts.it_value = ts.it_interval; // Initial expiration
timerfd_settime(tfd, 0, &ts, NULL);

uint64_t expirations;
read(tfd, &expirations, sizeof(expirations)); // Blocks until timer expires
```

Exam Pattern (2023-06-14 - `tfdtest`): A program is given an interval and a count. It configures a `timerfd` to expire at that interval. It then enters a loop, `read()`ing from the timer fd `count` times, printing a message after each expiration. The `mftdtest` (multiple timers) version requires using `poll` to monitor multiple `timerfd`s.

---

10. SHELL SCRIPTING AND PYTHON

Common Bash Patterns

File Processing

```
# Process all .c files
for file in *.c; do
    echo "Processing $file"
done

# Find files by pattern
find . -name "*.txt" -type f

# Count lines in files
wc -l *.c | sort -n

# Process file line by line
while IFS= read -r line; do
    echo "Line: $line"
done < input.txt
```

Text Processing

```
# Extract fields
awk '{print $1}' file.txt

# Pattern matching
grep -r "pattern" directory/

# Replace text
```

```
sed 's/old/new/g' file.txt

# Sort and unique
sort file.txt | uniq
```

Python File Operations

```
import os
import subprocess

# List directory
for entry in os.listdir('.'):
    if os.path.isfile(entry):
        print(f"File: {entry}")

# Walk directory tree - The go-to for recursive traversal
# Yields a 3-tuple (dirpath, dirnames, filenames) for each directory
for root, dirs, files in os.walk('.'):
    for file in files:
        full_path = os.path.join(root, file)
        print(full_path)

# Execute command - The modern way to run external commands
result = subprocess.run(['ls', '-l'], capture_output=True, text=True)
print(result.stdout)

# Check file types and attributes
import stat
file_stat = os.stat('filename')
if stat.S_ISREG(file_stat.st_mode):
    print("Regular file")
elif stat.S_ISDIR(file_stat.st_mode):
    print("Directory")
elif stat.S_ISLNK(os.lstat('filename').st_mode):
    print("Symbolic link")
```

Common Python Modules for System Programming:

- `os` **module**: `os.walk`, `os.listdir`, `os.stat`, `os.lstat`, `os.path.join`, `os.access`
- `sys` **module**: `sys.argv` for command line arguments
- `subprocess` **module**: For running external commands

- `magic` **library**: Often useful for identifying file types

---

11. COMMON EXAM PATTERNS

Pattern 1: File Inversion/Reversal

Task: Reverse byte order in a file **Key APIs**: pread, pwrite, stat **Strategy**: Read from end, write to beginning

Pattern 2: Directory Monitoring

Task: Watch directory for changes **Key APIs**: inotify_init, inotify_add_watch **Events**: IN_CREATE, IN_DELETE, IN_MODIFY

Pattern 3: Process Launching

Task: Execute multiple programs **Key APIs**: fork, exec*, wait/waitpid **Patterns**: Sequential, concurrent, with timeouts

Pattern 4: IPC Communication

Task: Exchange data between processes **Options**:

- Pipes (unnamed/named)
- Message queues
- Shared memory
- Signals

Pattern 5: File System Traversal

Task: Process files in directory tree **Key APIs**: opendir, readdir, stat **Common requirements**: Find duplicates, links, specific types

Pattern 6: Signal-based Communication

Task: Send data using signals **Key APIs**: signalfd, sigqueue **Techniques**: Using signal values, acknowledgments

Pattern 7: Time-based Execution

Task: Execute at specific times/intervals **Key APIs**: timerfd_create, poll/select **Applications**: Delayed execution, periodic tasks

---

12. SAMPLE SOLUTIONS

Example 1: File Byte Reversal

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

void reverse_file(const char *filename) {
    struct stat st;
    if (stat(filename, &st) < 0) {
        perror("stat");
        exit(1);
    }

    int fd = open(filename, O_RDWR);
    if (fd < 0) {
        perror("open");
        exit(1);
    }

    char buffer[4096];
    off_t start = 0;
    off_t end = st.st_size;
```

```
    while (start < end - 4096) {
        // Read from end
        ssize_t n = pread(fd, buffer, 4096, end - 4096);

        // Reverse buffer
        for (int i = 0; i < n/2; i++) {
            char tmp = buffer[i];
            buffer[i] = buffer[n-1-i];
            buffer[n-1-i] = tmp;
        }

        // Write to start
        pwrite(fd, buffer, n, start);

        start += n;
        end -= n;
    }

    // Handle remaining bytes
    if (start < end) {
        ssize_t remaining = end - start;
        char *temp = malloc(remaining);
        pread(fd, temp, remaining, start);

        // Reverse in place
        for (int i = 0; i < remaining/2; i++) {
            char tmp = temp[i];
            temp[i] = temp[remaining-1-i];
            temp[remaining-1-i] = tmp;
        }

        pwrite(fd, temp, remaining, start);
        free(temp);
    }

    close(fd);
}

int main(int argc, char *argv[]) {
```

```
    if (argc != 2) {
        fprintf(stderr, "Usage: %s filename\n", argv[0]);
        exit(1);
    }

    reverse_file(argv[1]);
    return 0;
}
```

Example 2: Parallel File Copy (Exam Pattern from 2023-02-16)

```
#define _XOPEN_SOURCE 500 // For pread
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <getopt.h>

void copy_chunk(int src_fd, int dest_fd, off_t offset, off_t chunk_size) {
    char buffer[4096];
    off_t bytes_to_copy = chunk_size;
    off_t current_pos = offset;

    while (bytes_to_copy > 0) {
        ssize_t to_read = (bytes_to_copy < sizeof(buffer)) ? bytes_to_copy :
        ↪  sizeof(buffer);
        ssize_t bytes_read = pread(src_fd, buffer, to_read, current_pos);
        if (bytes_read <= 0) break; // EOF or error

        ssize_t bytes_written = pwrite(dest_fd, buffer, bytes_read, current_pos);
        if (bytes_written < 0) {
            perror("pwrite failed");
            exit(1);
        }

        bytes_to_copy -= bytes_written;
        current_pos += bytes_written;
    }
```

```
}

int main(int argc, char *argv[]) {
    int jobs = 2; // Default to 2 processes
    int opt;

    while ((opt = getopt(argc, argv, "j:")) != -1) {
        if (opt == 'j') {
            jobs = atoi(optarg);
        }
    }

    if (optind >= argc - 1) {
        fprintf(stderr, "Usage: %s [-j num_jobs] <source> <destination>\n",
↪   argv[0]);
        exit(1);
    }

    char *src_file = argv[optind];
    char *dest_file = argv[optind + 1];

    int src_fd = open(src_file, O_RDONLY);
    if (src_fd < 0) { perror("open source"); exit(1); }

    struct stat st;
    if (fstat(src_fd, &st) < 0) { perror("fstat"); exit(1); }
    off_t file_size = st.st_size;

    int dest_fd = open(dest_file, O_WRONLY | O_CREAT | O_TRUNC, st.st_mode);
    if (dest_fd < 0) { perror("open dest"); exit(1); }
    // Pre-allocate space for the destination file
    if (ftruncate(dest_fd, file_size) < 0) { perror("ftruncate"); exit(1); }

    off_t chunk_size = file_size / jobs;

    for (int i = 0; i < jobs; i++) {
        if (fork() == 0) { // Child process
            off_t offset = i * chunk_size;
            off_t size = (i == jobs - 1) ? (file_size - offset) : chunk_size;
            copy_chunk(src_fd, dest_fd, offset, size);
```

```
            exit(0);
        }
    }

    // Parent waits for all children to finish
    for (int i = 0; i < jobs; i++) {
        wait(NULL);
    }

    close(src_fd);
    close(dest_fd);
    printf("Copied %s to %s using %d processes.\n", src_file, dest_file, jobs);

    return 0;
}
```

Example 3: Find Symlinks Pointing to the Same File (Python)

Corresponds to 2023-09-14, Esercizio 3.

```
#!/usr/bin/env python3
import os
import sys
from collections import defaultdict

def find_duplicate_symlink_targets(directory):
    # This dictionary will map inode numbers of targets to a list of symlinks
    target_inode_map = defaultdict(list)

    for root, _, files in os.walk(directory):
        for filename in files:
            path = os.path.join(root, filename)
            # We only care about symbolic links
            if os.path.islink(path):
                try:
                    # Get the stat of the file the link *points to*
                    target_stat = os.stat(path)
                    # Use the inode as a unique identifier for the target file
                    target_inode = target_stat.st_ino
                    target_inode_map[target_inode].append(path)
```

```
                except FileNotFoundError:
                    # It's a broken link, ignore it
                    pass

    # Print the results
    for inode, links in target_inode_map.items():
        if len(links) > 1:
            print(f"Target Inode {inode}: {' '.join(links)}")

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print(f"Usage: {sys.argv[0]} <directory>")
        sys.exit(1)

    find_duplicate_symlink_targets(sys.argv[1])
```

Example 2: Directory Monitor with inotify

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/inotify.h>

#define EVENT_SIZE (sizeof(struct inotify_event))
#define BUF_LEN (1024 * (EVENT_SIZE + 16))

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s directory\n", argv[0]);
        exit(1);
    }

    int fd = inotify_init();
    if (fd < 0) {
        perror("inotify_init");
        exit(1);
    }

    int wd = inotify_add_watch(fd, argv[1],
                               IN_CREATE | IN_DELETE | IN_MODIFY);
```

```
    if (wd < 0) {
        perror("inotify_add_watch");
        exit(1);
    }

    char buffer[BUF_LEN];

    while (1) {
        int length = read(fd, buffer, BUF_LEN);
        if (length < 0) {
            perror("read");
            exit(1);
        }

        int i = 0;
        while (i < length) {
            struct inotify_event *event =
                (struct inotify_event *)&buffer[i];

            if (event->len) {
                if (event->mask & IN_CREATE) {
                    printf("Created: %s\n", event->name);
                }
                if (event->mask & IN_DELETE) {
                    printf("Deleted: %s\n", event->name);
                }
                if (event->mask & IN_MODIFY) {
                    printf("Modified: %s\n", event->name);
                }
            }

            i += EVENT_SIZE + event->len;
        }
    }

    close(fd);
    return 0;
}
```

Example 3: Concurrent Process Execution

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>

void execute_commands_concurrently(char *commands[], int count) {
    pid_t *pids = malloc(count * sizeof(pid_t));

    // Launch all processes
    for (int i = 0; i < count; i++) {
        pids[i] = fork();
        if (pids[i] == 0) {
            // Child process
            char *args[100];
            char *cmd = strdup(commands[i]);
            int j = 0;

            // Parse command
            char *token = strtok(cmd, " ");
            while (token != NULL) {
                args[j++] = token;
                token = strtok(NULL, " ");
            }
            args[j] = NULL;

            execvp(args[0], args);
            perror("execvp");
            exit(1);
        }
    }

    // Wait for all to finish
    for (int i = 0; i < count; i++) {
        int status;
        waitpid(pids[i], &status, 0);
        printf("Process %d finished with status %d\n",
               pids[i], WEXITSTATUS(status));
    }
```

```
    free(pids);
}
```

Example 4: Signal-based String Transfer

```
// Receiver (rx.c)
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/signalfd.h>

int main() {
    printf("PID: %d\n", getpid());

    sigset_t mask;
    sigemptyset(&mask);
    sigaddset(&mask, SIGUSR1);
    sigaddset(&mask, SIGUSR2);
    sigprocmask(SIG_BLOCK, &mask, NULL);

    int sfd = signalfd(-1, &mask, 0);
    char received[9] = {0};
    int bit_count = 0;
    int byte_count = 0;

    while (1) {
        struct signalfd_siginfo si;
        read(sfd, &si, sizeof(si));

        if (si.ssi_signo == SIGUSR1) {
            received[byte_count] |= (1 << bit_count);
        }
        // SIGUSR2 means bit is 0, no action needed

        bit_count++;
        if (bit_count == 8) {
            bit_count = 0;
            byte_count++;
            if (byte_count == 8 || received[byte_count-1] == '\0') {
```

```
                printf("Received: %s\n", received);
                break;
            }
        }

        // Send acknowledgment
        kill(si.ssi_pid, SIGUSR1);
    }

    return 0;
}

// Transmitter (tx.c)
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <unistd.h>

void send_string(pid_t target, const char *str) {
    sigset_t mask;
    sigemptyset(&mask);
    sigaddset(&mask, SIGUSR1);
    sigprocmask(SIG_BLOCK, &mask, NULL);

    int sfd = signalfd(-1, &mask, 0);

    for (int i = 0; i < strlen(str) + 1; i++) {
        for (int bit = 0; bit < 8; bit++) {
            if (str[i] & (1 << bit)) {
                kill(target, SIGUSR1);
            } else {
                kill(target, SIGUSR2);
            }

            // Wait for acknowledgment
            struct signalfd_siginfo si;
            read(sfd, &si, sizeof(si));
        }
    }
}
```

```c
int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s pid message\n", argv[0]);
        return 1;
    }

    pid_t target = atoi(argv[1]);
    send_string(target, argv[2]);

    return 0;
}
```

Common Bash/Python Solutions

Find Files with Same Content (Bash)

```bash
#!/bin/bash
find "$1" -type f -exec md5sum {} \; | sort | \
    awk '{if ($1 == prev) {print prevfile, $2} prev=$1; prevfile=$2}'
```

Directory Tree Walker (Python)

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

def walk_tree(directory):
    for root, dirs, files in os.walk(directory):
        level = root.replace(directory, '').count(os.sep)
        indent = ' ' * 2 * level
        print(f'{indent}{os.path.basename(root)}/')
        sub_indent = ' ' * 2 * (level + 1)
        for file in files:
            print(f'{sub_indent}{file}')

if __name__ == '__main__':
    directory = sys.argv[1] if len(sys.argv) > 1 else '.'
    walk_tree(directory)
```

KEY EXAM TIPS

1. **Always check return values** - System calls can fail
2. **Handle edge cases** - Empty files, directories, permission issues
3. **Free allocated memory** - Avoid memory leaks
4. **Close file descriptors** - Prevent resource leaks
5. **Use proper error handling** - perror() for system call errors
6. **Test with various inputs** - Empty, large, special files
7. **Follow exact specifications** - Parameter order, output format
8. **Consider efficiency** - Use appropriate buffer sizes
9. **Handle signals properly** - Block when using signalfd
10. **Document assumptions** - Explain design choices

SUMMARY

This comprehensive guide covers all major topics for systems programming exams:

- File I/O operations (basic and advanced)
- Process management and IPC
- Signal handling (traditional and modern)
- Directory operations and monitoring
- Event-driven programming
- Shell scripting essentials

Focus on understanding the patterns and practicing implementation of common tasks like file processing, process coordination, and system monitoring.