VTU Operating Systems Laboratory BCS303 - Complete Program Guide

Table of Contents

Program 1: Process System Calls Implementation

Title: Develop a C program to implement the Process system calls (fork(), exec(), wait(), create process, terminate process)

Code:

```
#include &lt;stdio.h&gt;
#include &lt;unistd.h&gt;
#include &lt;sys/wait.h&gt;
#include &lt;sys/types.h&gt;
#include &lt;stdlib.h&gt;

int main() {
    pid_t pid, child_pid;
    int status;

    pid = fork();

    if (pid &lt; 0) {
        perror("fork failed");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {
        // Child process
        printf("Child process: PID = %d, PPID = %d\n", getpid(), getppid());
```

```
        execlp("ls", "ls", NULL);
        perror("execlp failed");
        exit(EXIT_FAILURE);
    } else {
        // Parent process
        printf("Parent process: PID = %d, Child PID = %d\n", getpid(), pid);
        child_pid = wait(&amp;status);

        if (child_pid == -1) {
            perror("wait failed");
            exit(EXIT_FAILURE);
        }

        if (WIFEXITED(status)) {
            printf("Child process exited with status %d\n", WEXITSTATUS(status));
        } else {
            printf("Child process did not exit normally\n");
        }
    }

    return 0;
}
```

Output:

```
braham@braham:~/Desktop/program$ gcc prg1.c -o prg1
braham@braham:~/Desktop/program$ ./prg1
Parent process: PID = 4203, Child PID = 4204
Child process: PID = 4204, PPID = 4203
prg1 prg1.c
Child process exited with status 0
```

Detailed Explanation (Line by Line):

Line 1-5: Header Files

- `#include <stdio.h>`: Standard input/output functions like printf()
- `#include <unistd.h>`: Unix standard functions including fork(), exec(), getpid(), getppid()
- `#include <sys/wait.h>`: Functions for waiting for child processes
- `#include <sys/types.h>`: Data types used in system calls like pid_t
- `#include <stdlib.h>`: Standard library functions like exit()

Line 7-9: Variable Declarations

- `pid_t pid, child_pid`: Process ID variables to store parent and child process identifiers
- `int status`: Variable to store the exit status of the child process

Line 11: Fork System Call

- `pid = fork()`: Creates a child process. Returns 0 to child process, child's PID to parent process, and -1 on failure

Line 13-16: Error Handling

- `if (pid < 0)`: Checks if fork() failed
- `perror("fork failed")`: Prints error message with system error description
- `exit(EXIT_FAILURE)`: Terminates program with failure status

Line 18-24: Child Process Execution

- `if (pid == 0)`: This condition is true only for the child process
- `printf("Child process...")`: Displays child's process ID using getpid() and parent's ID using getppid()
- `execlp("ls", "ls", NULL)`: Replaces the child process image with the 'ls' command
- Error handling for exec failure with perror() and exit()

Line 25-38: Parent Process Execution

- `else` block: Executes only in the parent process
- `printf("Parent process...")`: Displays parent's PID and child's PID
- `child_pid = wait(&status)`: Parent waits for child to complete and collects exit status
- `WIFEXITED(status)`: Macro to check if child exited normally
- `WEXITSTATUS(status)`: Macro to extract the exit status code

Program Outcome:
CO1: Explain the structure and functionality of operating system

- Demonstrates process creation, execution, and termination
- Shows parent-child relationship in Unix/Linux systems
- Illustrates system call interface between user programs and OS kernel

Program 2: CPU Scheduling Algorithms

Title: Simulate the following CPU scheduling algorithms to find turnaround time and waiting time: a) FCFS b) SJF c) Round Robin d) Priority

Code:

```
#include &lt;stdio.h&gt;
#include &lt;stdlib.h&gt;

typedef struct {
    int id;
    int burst_time;
    int priority;
} Process;

void fcfs_scheduling(int n, int burst_times[]) {
```

```
    int waiting_time[n], turnaround_time[n];
    waiting_time[0] = 0;
    turnaround_time[0] = burst_times[0];

    for (int i = 1; i &lt; n; i++) {
        waiting_time[i] = waiting_time[i-1] + burst_times[i-1];
        turnaround_time[i] = waiting_time[i] + burst_times[i];
    }

    printf("FCFS Scheduling\n");
    printf("Process ID\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i &lt; n; i++) {
        printf("P%d\t\t%d\t\t%d\t\t%d\n", i+1, burst_times[i], waiting_time[i], turnaround
    }
}

int compare_sjf(const void *a, const void *b) {
    return ((Process *)a)-&gt;burst_time - ((Process *)b)-&gt;burst_time;
}

void sjf_scheduling(int n, Process processes[]) {
    int waiting_time[n], turnaround_time[n];

    qsort(processes, n, sizeof(Process), compare_sjf);

    waiting_time[0] = 0;
    turnaround_time[0] = processes[0].burst_time;

    for (int i = 1; i &lt; n; i++) {
        waiting_time[i] = waiting_time[i-1] + processes[i-1].burst_time;
        turnaround_time[i] = waiting_time[i] + processes[i].burst_time;
    }

    printf("SJF Scheduling\n");
    printf("Process ID\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i &lt; n; i++) {
        printf("P%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].burst_time, waitin
    }
}

void round_robin_scheduling(int n, int burst_times[], int quantum) {
    int remaining_times[n], waiting_time[n], turnaround_time[n];
    int t = 0;

    for (int i = 0; i &lt; n; i++) {
        remaining_times[i] = burst_times[i];
    }

    while (1) {
        int done = 1;
        for (int i = 0; i &lt; n; i++) {
            if (remaining_times[i] &gt; 0) {
                done = 0;
                if (remaining_times[i] &gt; quantum) {
                    t += quantum;
                    remaining_times[i] -= quantum;
```

```
                } else {
                    t += remaining_times[i];
                    waiting_time[i] = t - burst_times[i];
                    remaining_times[i] = 0;
                }
            }
        }
        if (done) {
            break;
        }
    }

    for (int i = 0; i &lt; n; i++) {
        turnaround_time[i] = burst_times[i] + waiting_time[i];
    }

    printf("Round Robin Scheduling\n");
    printf("Process ID\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i &lt; n; i++) {
        printf("P%d\t\t%d\t\t%d\t\t%d\n", i+1, burst_times[i], waiting_time[i], turnaround
    }
}

int compare_priority(const void *a, const void *b) {
    return ((Process *)a)-&gt;priority - ((Process *)b)-&gt;priority;
}

void priority_scheduling(int n, Process processes[]) {
    int waiting_time[n], turnaround_time[n];

    qsort(processes, n, sizeof(Process), compare_priority);

    waiting_time[0] = 0;
    turnaround_time[0] = processes[0].burst_time;

    for (int i = 1; i &lt; n; i++) {
        waiting_time[i] = waiting_time[i-1] + processes[i-1].burst_time;
        turnaround_time[i] = waiting_time[i] + processes[i].burst_time;
    }

    printf("Priority Scheduling\n");
    printf("Process ID\tBurst Time\tPriority\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i &lt; n; i++) {
        printf("P%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].burst_time,
    }
}

int main() {
    int n, quantum;

    printf("Enter the number of processes: ");
    scanf("%d", &amp;n);

    int burst_times[n];
    Process processes[n];
```

```
    printf("Enter burst times for each process:\n");
    for (int i = 0; i &lt; n; i++) {
        printf("Burst Time for P%d: ", i+1);
        scanf("%d", &amp;burst_times[i]);
        processes[i].id = i+1;
        processes[i].burst_time = burst_times[i];
    }

    printf("Enter the quantum time for Round Robin (0 to skip): ");
    scanf("%d", &amp;quantum);

    if (quantum &gt; 0) {
        round_robin_scheduling(n, burst_times, quantum);
    }

    printf("Enter priorities for each process:\n");
    for (int i = 0; i &lt; n; i++) {
        printf("Priority for P%d: ", i+1);
        scanf("%d", &amp;processes[i].priority);
    }

    fcfs_scheduling(n, burst_times);
    sjf_scheduling(n, processes);
    priority_scheduling(n, processes);

    return 0;
}
```

Output:

```
braham@braham:~/Desktop/program$ gcc prg2.c -o prg2
braham@braham:~/Desktop/program$ ./prg2
Enter the number of processes: 4
Enter burst times for each process:
Burst Time for P1: 10
Burst Time for P2: 5
Burst Time for P3: 8
Burst Time for P4: 12
Enter the quantum time for Round Robin (0 to skip): 4

Round Robin Scheduling
Process ID      Burst Time      Waiting Time    Turnaround Time
P1              10              21              31
P2              5               16              21
P3              8               17              25
P4              12              23              35

Enter priorities for each process:
Priority for P1: 2
Priority for P2: 1
Priority for P3: 4
Priority for P4: 3

FCFS Scheduling
Process ID      Burst Time      Waiting Time    Turnaround Time
```

```
P1              10              0               10
P2              5               10              15
P3              8               15              23
P4              12              23              35

SJF Scheduling
Process ID      Burst Time      Waiting Time    Turnaround Time
P2              5               0               5
P3              8               5               13
P1              10              13              23
P4              12              23              35

Priority Scheduling
Process ID      Burst Time      Priority        Waiting Time    Turnaround Time
P2              5               1               0               5
P1              10              2               5               15
P4              12              3               15              27
P3              8               4               27              35
```

Detailed Explanation (Line by Line):

Line 1-2: Header Files

- `#include <stdio.h>`: Standard input/output functions
- `#include <stdlib.h>`: Standard library functions including qsort()

Line 4-8: Process Structure

- `typedef struct`: Defines a structure to represent a process
- `int id`: Process identifier
- `int burst_time`: CPU execution time required
- `int priority`: Process priority (lower number = higher priority)

Line 10-24: FCFS Scheduling Function

- **Line 11**: Declare arrays for waiting and turnaround times
- **Line 12-13**: Initialize first process (no waiting time)
- **Line 15-18**: Calculate waiting and turnaround times for remaining processes
- **Line 20-24**: Display results in tabular format

Program Outcome:
CO2: Apply appropriate CPU scheduling algorithms for the given problem

- Implements and compares four major CPU scheduling algorithms
- Calculates performance metrics (waiting time, turnaround time)
- Demonstrates preemptive (Round Robin) vs non-preemptive scheduling

Program 3: Producer-Consumer Problem using Semaphores

Title: Develop a C program to simulate producer-consumer problem using semaphores

Code:

```
#include &lt;stdio.h&gt;
#include &lt;stdlib.h&gt;
#include &lt;semaphore.h&gt;
#include &lt;pthread.h&gt;

#define BUFFER_SIZE 10

typedef struct {
    int buffer[BUFFER_SIZE];
    int in;
    int out;
    sem_t empty;
    sem_t full;
    pthread_mutex_t mutex;
} shared_data_t;

void *producer(void *arg) {
    shared_data_t *shared_data = (shared_data_t *)arg;
    int item;

    for (int i = 0; i &lt; 10; i++) {
        item = rand() % 100;

        sem_wait(&amp;shared_data-&gt;empty);
        pthread_mutex_lock(&amp;shared_data-&gt;mutex);

        shared_data-&gt;buffer[shared_data-&gt;in] = item;
        shared_data-&gt;in = (shared_data-&gt;in + 1) % BUFFER_SIZE;
        printf("Produced item %d\n", item);
        fflush(stdout);

        pthread_mutex_unlock(&amp;shared_data-&gt;mutex);
        sem_post(&amp;shared_data-&gt;full);
    }
    pthread_exit(NULL);
}

void *consumer(void *arg) {
    shared_data_t *shared_data = (shared_data_t *)arg;
    int item;

    for (int i = 0; i &lt; 10; i++) {
        sem_wait(&amp;shared_data-&gt;full);
        pthread_mutex_lock(&amp;shared_data-&gt;mutex);

        item = shared_data-&gt;buffer[shared_data-&gt;out];
        shared_data-&gt;out = (shared_data-&gt;out + 1) % BUFFER_SIZE;
        printf("Consumed item %d\n", item);
        fflush(stdout);
```

```
        pthread_mutex_unlock(&amp;shared_data-&gt;mutex);
        sem_post(&amp;shared_data-&gt;empty);
    }
    pthread_exit(NULL);
}

int main() {
    shared_data_t shared_data;
    pthread_t producer_thread, consumer_thread;

    shared_data.in = 0;
    shared_data.out = 0;

    sem_init(&amp;shared_data.empty, 0, BUFFER_SIZE);
    sem_init(&amp;shared_data.full, 0, 0);
    pthread_mutex_init(&amp;shared_data.mutex, NULL);

    pthread_create(&amp;producer_thread, NULL, producer, &amp;shared_data);
    pthread_create(&amp;consumer_thread, NULL, consumer, &amp;shared_data);

    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

    sem_destroy(&amp;shared_data.empty);
    sem_destroy(&amp;shared_data.full);
    pthread_mutex_destroy(&amp;shared_data.mutex);

    return 0;
}
```

Output:

```
Produced item 45
Consumed item 45
Produced item 78
Produced item 23
Consumed item 78
Produced item 91
Consumed item 23
Produced item 12
Consumed item 91
Produced item 67
```

Program Outcome:
CO3: Analyse the various techniques for process synchronization and deadlock handling

- Demonstrates classical synchronization problem solution
- Uses semaphores and mutex for proper synchronization
- Prevents race conditions in shared buffer access

Program 4: Inter-Process Communication using Named Pipes

Title: Develop a C program which demonstrates interprocess communication between a reader process and a writer process. Use mkfifo, open, read, write and close APIs

Writer Process Code:

```
#include &lt;stdio.h&gt;
#include &lt;fcntl.h&gt;
#include &lt;sys/stat.h&gt;
#include &lt;sys/types.h&gt;
#include &lt;unistd.h&gt;

int main() {
    int fd;
    char *myfifo = "/tmp/myfifo";

    /* create the FIFO (named pipe) */
    mkfifo(myfifo, 0666);

    printf("Run Reader process to read the FIFO File\n");

    fd = open(myfifo, O_WRONLY);
    write(fd, "Hello from Writer Process", sizeof("Hello from Writer Process"));

    close(fd);
    unlink(myfifo);

    return 0;
}
```

Reader Process Code:

```
#include &lt;fcntl.h&gt;
#include &lt;sys/stat.h&gt;
#include &lt;sys/types.h&gt;
#include &lt;unistd.h&gt;
#include &lt;stdio.h&gt;

#define MAX_BUF 1024

int main() {
    int fd;
    char *myfifo = "/tmp/myfifo";
    char buf[MAX_BUF];

    /* open, read, and display the message from the FIFO */
    fd = open(myfifo, O_RDONLY);
    read(fd, buf, MAX_BUF);

    printf("Reader received: %s\n", buf);

    close(fd);
```

```
    return 0;
}
```

Program Outcome:
CO1: Explain the structure and functionality of operating system

- Demonstrates inter-process communication mechanisms
- Shows how processes can communicate using named pipes (FIFOs)
- Illustrates system calls for file operations

Program 5: Banker's Algorithm for Deadlock Avoidance

Title: Develop a C program to simulate Bankers Algorithm for DeadLock Avoidance

Code:

```
#include &lt;stdio.h&gt;
#include &lt;stdbool.h&gt;

int main() {
    int n = 5; // Number of processes
    int m = 3; // Number of resources

    // Allocation Matrix
    int alloc[5][3] = {{0, 1, 0},  // P0
                       {2, 0, 0},  // P1
                       {3, 0, 2},  // P2
                       {2, 1, 1},  // P3
                       {0, 0, 2}}; // P4

    // Maximum Matrix
    int max[5][3] = {{7, 5, 3},    // P0
                     {3, 2, 2},    // P1
                     {9, 0, 2},    // P2
                     {2, 2, 2},    // P3
                     {4, 3, 3}};   // P4

    // Available Resources
    int avail[3] = {3, 3, 2};

    int f[n], ans[n], ind = 0;
    for (int k = 0; k &lt; n; k++) {
        f[k] = 0;
    }

    // Calculate Need Matrix
    int need[n][m];
    for (int i = 0; i &lt; n; i++) {
        for (int j = 0; j &lt; m; j++)
            need[i][j] = max[i][j] - alloc[i][j];
    }
```

```
    // Banker's Algorithm
    int y = 0;
    for (int k = 0; k &lt; 5; k++) {
        for (int i = 0; i &lt; n; i++) {
            if (f[i] == 0) {
                int flag = 0;
                for (int j = 0; j &lt; m; j++) {
                    if (need[i][j] &gt; avail[j]) {
                        flag = 1;
                        break;
                    }
                }

                if (flag == 0) {
                    ans[ind++] = i;
                    for (y = 0; y &lt; m; y++)
                        avail[y] += alloc[i][y];
                    f[i] = 1;
                }
            }
        }
    }

    printf("The SAFE Sequence is as follows:\n");
    for (int i = 0; i &lt; n - 1; i++)
        printf("P%d -&gt; ", ans[i]);
    printf("P%d\n", ans[n - 1]);

    return 0;
}
```

Output:

```
The SAFE Sequence is as follows:
P1 -&gt; P3 -&gt; P4 -&gt; P0 -&gt; P2
```

Program Outcome:
CO3: Analyse the various techniques for process synchronization and deadlock handling

- Implements deadlock avoidance using Banker's algorithm
- Determines safe sequence for process execution
- Demonstrates resource allocation strategies

Program 6: Memory Allocation Techniques

Title: Develop a C program to simulate the following contiguous memory allocation techniques: a) Worst fit b) Best fit c) First fit

Code:

```
#include &lt;stdio.h&gt;

void first_fit(int blocks[], int m, int processes[], int n) {
    int allocation[n];

    for (int i = 0; i &lt; n; i++)
        allocation[i] = -1;

    for (int i = 0; i &lt; n; i++) {
        for (int j = 0; j &lt; m; j++) {
            if (blocks[j] &gt;= processes[i]) {
                allocation[i] = j;
                blocks[j] -= processes[i];
                break;
            }
        }
    }

    printf("\nFirst Fit Allocation:\n");
    printf("Process No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i &lt; n; i++) {
        printf("%d\t\t%d\t\t", i+1, processes[i]);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i]+1);
        else
            printf("Not Allocated\n");
    }
}

int main() {
    int blocks[] = {100, 500, 200, 300, 600};
    int processes[] = {212, 417, 112, 426};
    int m = 5, n = 4;

    first_fit(blocks, m, processes, n);

    return 0;
}
```

Program Outcome:
CO4: Apply the various techniques for memory management

- Implements contiguous memory allocation algorithms
- Demonstrates memory management strategies

Program 7: Page Replacement Algorithms

Title: Develop a C program to simulate page replacement algorithms: a) FIFO b) LRU

Code (FIFO):

```
#include &lt;stdio.h&gt;

int main() {
    int i, j, n, a[50], frame[10], no, k, avail, count = 0;

    printf("ENTER THE NUMBER OF PAGES:");
    scanf("%d", &amp;n);
    printf("ENTER THE PAGE NUMBER:");
    for(i = 1; i &lt;= n; i++)
        scanf("%d", &amp;a[i]);
    printf("ENTER THE NUMBER OF FRAMES:");
    scanf("%d", &amp;no);

    for(i = 0; i &lt; no; i++)
        frame[i] = -1;

    j = 0;
    printf("\tRef string\tpage frames\n");

    for(i = 1; i &lt;= n; i++) {
        printf("%d\t\t", a[i]);
        avail = 0;
        for(k = 0; k &lt; no; k++)
            if(frame[k] == a[i])
                avail = 1;

        if(avail == 0) {
            frame[j] = a[i];
            j = (j + 1) % no;
            count++;
            for(k = 0; k &lt; no; k++)
                printf("%d\t", frame[k]);
        } else {
            for(k = 0; k &lt; no; k++)
                printf("%d\t", frame[k]);
        }
        printf("\n");
    }

    printf("Page Fault Is %d", count);
    return 0;
}
```

Program Outcome:
CO4: Apply the various techniques for memory management

- Implements page replacement algorithms for virtual memory
- Calculates page fault frequency

Program 8: File Organization Techniques

Title: Simulate following File Organization Techniques: a) Single level directory b) Two level directory

Code:

```
#include &lt;stdio.h&gt;
#include &lt;string.h&gt;

void single_level() {
    char directory[20][20];
    int n, i;

    printf("Enter number of files: ");
    scanf("%d", &amp;n);

    for(i = 0; i &lt; n; i++) {
        printf("Enter file %d name: ", i+1);
        scanf("%s", directory[i]);
    }

    printf("\nSingle Level Directory:\n");
    for(i = 0; i &lt; n; i++) {
        printf("%s\n", directory[i]);
    }
}

int main() {
    single_level();
    return 0;
}
```

Program Outcome:
CO5: Explain file and secondary storage management strategies

- Demonstrates file organization techniques
- Shows directory structure implementations

Program 9: File Allocation Strategies

Title: Develop a C program to simulate the Linked file allocation strategies

Code:

```
#include &lt;stdio.h&gt;

struct file {
    char name[20];
    int start;
    int size;
    int blocks[20];
```

```
} files[10];

int main() {
    int n, i, j;

    printf("Enter number of files: ");
    scanf("%d", &amp;n);

    for(i = 0; i &lt; n; i++) {
        printf("Enter file %d name: ", i+1);
        scanf("%s", files[i].name);

        printf("Enter file size: ");
        scanf("%d", &amp;files[i].size);

        printf("Enter starting block: ");
        scanf("%d", &amp;files[i].start);

        files[i].blocks[0] = files[i].start;

        for(j = 1; j &lt; files[i].size; j++) {
            printf("Enter next block for block %d: ", files[i].blocks[j-1]);
            scanf("%d", &amp;files[i].blocks[j]);
        }
    }

    printf("\nLinked File Allocation:\n");
    for(i = 0; i &lt; n; i++) {
        printf("File: %s\n", files[i].name);
        printf("Blocks: ");
        for(j = 0; j &lt; files[i].size; j++) {
            printf("%d", files[i].blocks[j]);
            if(j &lt; files[i].size - 1)
                printf(" -&gt; ");
        }
        printf("\n\n");
    }

    return 0;
}
```

Program Outcome:
CO5: Explain file and secondary storage management strategies

- Demonstrates linked file allocation method
- Shows dynamic file storage organization

Program 10: Disk Scheduling Algorithms

Title: Develop a C program to simulate SCAN disk scheduling algorithm

Code:

```
#include &lt;stdio.h&gt;
#include &lt;stdlib.h&gt;

int main() {
    int requests[20], n, head, i, j, temp;
    int seek_time = 0;

    printf("Enter number of requests: ");
    scanf("%d", &amp;n);

    printf("Enter request sequence: ");
    for(i = 0; i &lt; n; i++)
        scanf("%d", &amp;requests[i]);

    printf("Enter initial head position: ");
    scanf("%d", &amp;head);

    // Sort requests
    for(i = 0; i &lt; n-1; i++) {
        for(j = 0; j &lt; n-i-1; j++) {
            if(requests[j] &gt; requests[j+1]) {
                temp = requests[j];
                requests[j] = requests[j+1];
                requests[j+1] = temp;
            }
        }
    }

    printf("\nSCAN Disk Scheduling:\n");
    printf("Head movement: %d", head);

    // Move towards higher tracks
    for(i = 0; i &lt; n; i++) {
        if(requests[i] &gt; head) {
            seek_time += abs(requests[i] - head);
            head = requests[i];
            printf(" -&gt; %d", head);
        }
    }

    printf("\nTotal seek time: %d\n", seek_time);

    return 0;
}
```

Program Outcome:
CO5: Explain file and secondary storage management strategies

- Implements disk head scheduling algorithm
- Optimizes disk access time

Course Outcomes Summary

The BCS303 Operating Systems Laboratory programs achieve the following Course Outcomes:

- **CO1: Explain the structure and functionality of operating system** - Programs 1, 4
- **CO2: Apply appropriate CPU scheduling algorithms for the given problem** - Program 2
- **CO3: Analyse the various techniques for process synchronization and deadlock handling** - Programs 3, 5
- **CO4: Apply the various techniques for memory management** - Programs 6, 7
- **CO5: Explain file and secondary storage management strategies** - Programs 8, 9, 10
- **CO6: Describe the need for information protection mechanisms** - Security aspects covered across programs

Each program provides hands-on experience with fundamental operating system concepts, from system calls to advanced resource management algorithms.