slip-1

Q1

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main() {
    pid_t child_pid;
    child_pid = fork();
    if (child_pid == 0) {
        printf("Child process with PID: %d\n", getpid());
        nice(10);
        printf("Child process priority increased\n");
    } else if (child_pid > 0) {
        printf("Parent process with PID: %d\n", getpid());
        wait(NULL);
    } else {
        perror("Fork failed");
        return 1;
    }
    return 0;
}
```

Q2

```
#include <stdio.h>
int main() {
    int n = 3;
    int referenceString[] = {3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6};
    int memory[n], faults = 0, i = 0;
    for (int j = 0; j < n; j++)
        memory[j] = -1;
    printf("Page Scheduling using FIFO:\n");
    for (int r = 0; r < 15; r++) {
        int page = referenceString[r], found = 0;
        for (int j = 0; j < n; j++) {
            if (memory[j] == page) {
                found = 1;
                break;
            }
        }
        if (!found) {
            faults++;
            if (memory[i % n] != -1) {
                printf("Page %d replaced by Page %d\n", memory[i % n], page);
            }
            memory[i % n] = page;
            i++;
            printf("Page %d -> Page Fault\nMemory: ", page);
            for (int j = 0; j < n; j++) {
                printf("%d ", memory[j]);
            }
            printf("\n");
        }
    }
    printf("Total Page Faults: %d\n", faults);
    return 0;
}
```

slip-2

Q1

```
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t child_pid = fork();

    if (child_pid == 0) {
        printf("Hello World\n");
    } else if (child_pid > 0) {
        printf("Hi\n");
    } else {
        perror("Fork failed");
        return 1;
    }
    return 0;
}
```

Q2

```
#include <stdio.h>

#include <stdlib.h>

#include <time.h>


#define MAX_PROCESSES 10


typedef struct {

    int arrival, burst, turnaround, waiting;

} Process;


int cmpArrival(const void *a, const void *b) {

    return ((Process *)a)->arrival - ((Process *)b)->arrival;

}


int main() {

    int n;

    scanf("%d", &n);

    Process p[MAX_PROCESSES] = {0};

    srand(time(0));


    for (int i = 0; i < n; i++) {

        p[i].arrival = i;

        p[i].burst = rand() % 10 + 1;

    }
    qsort(p, n, sizeof(Process), cmpArrival);
    printf("Gantt Chart:\n");
    float tTAT = 0, tWT = 0, t = 0;

    for (int i = 0; i < n; i++) {
        printf("P%d ", i + 1);
        p[i].turnaround = t + p[i].burst - p[i].arrival;
        p[i].waiting = p[i].turnaround - p[i].burst;
        t += p[i].burst + 2;
        tTAT += p[i].turnaround;
        tWT += p[i].waiting;
    }

    printf("\nProcess\tTurnaround\tWaiting\n");

    for (int i = 0; i < n; i++)
        printf("P%d\t%d\t\t%d\n", i + 1, p[i].turnaround, p[i].waiting);

    printf("Avg TAT: %.2f\nAvg WT: %.2f\n", tTAT / n, tWT / n);

    return 0;
}
```

slip3

Q1

```
#include <stdio.h>
#include <unistd.h>

int main() {
   pid_t child_pid = fork();

   if (child_pid == 0) {
      printf("Child process (PID: %d)\n", getpid());
      execl("/bin/ls", "ls", NULL);
   } else if (child_pid > 0) {
      printf("Parent process (PID: %d)\n", getpid());
      wait(NULL);
      printf("Child process has terminated.\n");
   } else {
      perror("Fork failed");
      return 1;
```

```
    }
    return 0;
}
```

Q2

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_PROCESSES 10

typedef struct {
    int arrival, burst, turnaround, waiting;
} Process;

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

    Process p[MAX_PROCESSES] = {0};
    srand(time(NULL));

    for (int i = 0; i < n; i++) {
        p[i].arrival = i;
        p[i].burst = rand() % 10 + 1;
    }
    printf("Gantt Chart:\n");
    float tTAT = 0, tWT = 0, t = 0;

    for (int i = 0; i < n; i++) {
        printf("P%d ", i + 1);
        t += p[i].burst + 2;
        p[i].turnaround = t - p[i].arrival;
        p[i].waiting = p[i].turnaround - p[i].burst;
        tTAT += p[i].turnaround;
        tWT += p[i].waiting;
    }

    printf("\nProcess\tTurnaround\tWaiting\n");
    for (int i = 0; i < n; i++)
        printf("P%d\t%d\t\t%d\n", i + 1, p[i].turnaround, p[i].waiting);

    printf("Avg TAT: %.2f\nAvg WT: %.2f\n", tTAT / n, tWT / n);

    return 0;
}
```

slip4

Q1

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    pid_t child_pid = fork(); // Create a new process

    if (child_pid == 0) {
        // This block is executed by the child process
        printf("Child process (PID: %d) is running, and its parent (PPID: %d) has terminated.\n", getpid(), getppid());
        sleep(5); // Simulate some work in the child process
        printf("Child process (PID: %d) has completed.\n", getpid());
    } else if (child_pid > 0) {
        // This block is executed by the parent process
        printf("Parent process (PID: %d) is running and will terminate shortly.\n", getpid());
        sleep(2); // Simulate some work in the parent process
        printf("Parent process (PID: %d) has terminated.\n", getpid());
    } else {
        // Handle fork failure
        perror("Fork failed");
        return 1;
    }
    return 0;
}
```

Q2

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 10

typedef struct {
    int arrival_time;
    int burst_time;
    int priority;
    int turnaround_time;
    int waiting_time;
} Process;

int main() {
    int n, total_time = 0, total_turnaround_time = 0, total_waiting_time = 0;
    srand(time(0)); // Seed for random number generation

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

    Process p[N] = {0}; // Array of processes
    // Input arrival time, burst time and priority for each process
    for (int i = 0; i < n; i++) {
        printf("Enter Arrival Time, Burst Time, and Priority for Process P%d: ", i + 
1);
        scanf("%d %d %d", &p[i].arrival_time, &p[i].burst_time, &p[i].priority);
    }

    printf("Gantt Chart:\n");
    for (int time = 0; time < total_time; time += 2) { // Fixed I/O waiting time
        int idx = -1;
        for (int j = 0; j < n; j++) {
            if (p[j].arrival_time <= time && (idx == -1 || p[j].priority < 
p[idx].priority)) {
                idx = j;
            }
        }
        if (idx != -1) {
            printf("P%d ", idx + 1);
            p[idx].turnaround_time = time + p[idx].burst_time;
            p[idx].waiting_time = time - p[idx].arrival_time;
            total_turnaround_time += p[idx].turnaround_time;
            total_waiting_time += p[idx].waiting_time;
            time += p[idx].burst_time + 2; // Process burst time + I/O waiting time
        }
    }

    printf("\nProcess\tTurnaround Time\tWaiting Time\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t\t%d\n", i + 1, p[i].turnaround_time, p[i].waiting_time);
    }

    printf("Avg Turnaround Time: %.2f\n", (float)total_turnaround_time / n);
    printf("Avg Waiting Time: %.2f\n", (float)total_waiting_time / n);

    return 0;
}
```

slip5

Q1

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

int main() {
    pid_t child_pid = fork(); // Create a new process

    if (child_pid == 0) {
        // This block is executed by the child process
        printf("Child process with PID: %d\n", getpid());
        nice(10); // Increase child process priority
        printf("Child process priority increased\n");
    } else if (child_pid > 0) {
        // This block is executed by the parent process
        printf("Parent process with PID: %d\n", getpid());
        wait(NULL); // Wait for child process to complete
    } else {
        // Handle fork failure
        perror("Fork failed");
        return 1;
    }
    return 0;
}
```

Q2

```
#include <stdio.h>
int main() {
    int n = 3;
    int referenceString[] = {3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6};
    int memory[n];
    int faults = 0;
    int i = 0;
    for (int j = 0; j < n; j++) {
        memory[j] = -1;
    }
    printf("Page Scheduling using FIFO:\n");

    for (int r = 0; r < 15; r++) {
        int page = referenceString[r];
        int found = 0;
        for (int j = 0; j < n; j++) {
            if (memory[j] == page) {
                found = 1;
                break;
            }
        }

        if (!found) {
            faults++;
            if (memory[i % n] != -1) {
```

```
            printf("Page %d replaced by Page %d\n", memory[i % n], page);
        }
        memory[i % n] = page;
        i++;
        printf("Page %d -> Page Fault\nMemory: ", page);
        for (int j = 0; j < n; j++) {
            printf("%d ", memory[j]);
        }
        printf("\n");
      }
    }

    printf("Total Page Faults: %d\n", faults);
    return 0;
}
```

slip-6

Q1

```
#include <stdio.h>
#include <time.h>
int main() {
    clock_t start = clock();
    for (volatile int i = 0; i < 1000000; i++);
    clock_t end = clock();
    double executionTime = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Execution time: %.4f seconds\n", executionTime);
    return 0;
}
```

Q2

```
#include <stdio.h>
int main() {
    int n;
    printf("Enter memory frames: ");
    scanf("%d", &n);
    int ref[] = {3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6};
    int len = sizeof(ref) / sizeof(ref[0]);
    int mem[n], faults = 0, i = 0;
    for (int j = 0; j < n; j++) mem[j] = -1;
    printf("Page Scheduling using FIFO:\n");
    for (int r = 0; r < len; r++) {
        int p = ref[r], found = 0;
        for (int j = 0; j < n; j++) {
            if (mem[j] == p) {
                found = 1;
                break;
            }
        }

        if (!found) {
            faults++;
            if (mem[i % n] != -1) {
                printf("Page %d replaced by Page %d\n", mem[i % n], p);
            }
            mem[i % n] = p;
            i++;
            printf("Page %d -> Page Fault\nMemory: ", p);
            for (int j = 0; j < n; j++) {
                printf("%d ", mem[j]);
            }
            printf("\n");
        }
    }

    printf("Total Page Faults: %d\n", faults);
    return 0;
}
```

slip-7

Q1

```
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t child_pid = fork();

    if (child_pid == 0) {
        execl("/bin/ls", "ls", (char *)NULL);
    } else if (child_pid > 0) {
        sleep(5);
    } else {
        perror("Fork failed");
        return 1;
    }

    return 0;
}
```

Q2

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    int n;
    srand(time(NULL));
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    float tTAT = 0, tWT = 0, t = 0;

    for (int i = 0; i < n; i++) {
        int burst = rand() % 10 + 1;
        t += burst + 2;
        printf("P%d %d %d\n", i + 1, (int)t, (int)t - burst);
        tTAT += t;
        tWT += t - burst;
    }

    printf("Avg TAT: %.2f\nAvg WT: %.2f\n", tTAT / n, tWT / n);
    return 0;
}
```

slip-8

Q1

```
#include <stdio.h>


int main() {
    int n, m;
    printf("Enter processes and resources: ");
    scanf("%d %d", &n, &m);
    int alloc[n][m], max[n][m], need[n][m];
    printf("Enter allocation and max matrices:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            scanf("%d %d", &alloc[i][j], &max[i][j]);
            need[i][j] = max[i][j] - alloc[i][j];
        }
    }
    printf("Need matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            printf("%d ", need[i][j]);
        }
        printf("\n");
    }


    return 0;
}
```

Q2

```
#include <stdio.h>

int main() {
    int n;
    printf("Enter memory frames: ");
    scanf("%d", &n);

    int ref[] = {12, 15, 12, 18, 6, 8, 11, 12, 19, 12, 6, 8, 12, 15, 19, 8};
    int len = sizeof(ref) / sizeof(ref[0]);
    int mem[n], faults = 0, future[len];

    for (int i = 0; i < n; i++) mem[i] = -1;

    printf("Page Scheduling using OPT:\n");
    for (int i = 0; i < len; i++) {
        int p = ref[i], found = 0;

        for (int j = 0; j < n; j++) {
            if (mem[j] == p) {
                found = 1;
                break;
            }
            future[j] = len;
            for (int k = i + 1; k < len; k++) {
                if (mem[j] == ref[k]) {
```

```
                future[j] = k;
                break;
            }
        }
    }

    if (!found) {
        faults++;
        int fIdx = 0;
        for (int j = 0; j < n; j++) {
            if (future[j] > future[fIdx]) {
                fIdx = j;
            }
        }
        mem[fIdx] = p;
    }
}

printf("Total Page Faults: %d\n", faults);
return 0;
}
```

slip-9

Q1

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    pid_t child_pid = fork();

    if (child_pid == 0) {
        execl("/bin/ls", "ls", (char *)NULL);
        perror("execl() failed");
    } else if (child_pid > 0) {
        sleep(5);
    } else {
        perror("Fork failed");
        return 1;
    }

    return 0;
}
```

Q2

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
typedef struct {
    int a, b, r, t, w;
} P;

int main() {
    int n, q, s = 0, i = 0;
    srand(time(0));
    printf("Enter n and quantum: ");
    scanf("%d%d", &n, &q);
    P p[n];
    for (int j = 0; j < n; j++) {
        p[j].a = j;
        p[j].b = rand() % 10 + 1;
        p[j].r = p[j].b;
    }
    printf("Gantt Chart:\n");
    while (1) {
        int d = 1;
        for (int j = 0; j < n; j++) {
            if (p[j].r > 0) {
                d = 0;
                int u = p[j].r > q ? q : p[j].r;
                p[j].r -= u;
                s += u;
                printf("P%d ", j + 1);
                p[j].a = s;
            }
        }
        if (d) break;
    }
    printf("\nTurnaround\tWaiting\n");
    for (int j = 0; j < n; j++) {
        p[j].t = p[j].a - j;
        p[j].w = p[j].t - p[j].b;
        printf("P%d\t%d\t\t%d\n", j + 1, p[j].t, p[j].w);
        i += p[j].t;
    }
    printf("Avg TAT: %.2f\nAvg WT: %.2f\n", (float)i / n, (float)i / n);
    return 0;
}
```

slip10

Q1

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    pid_t child_pid = fork();
    if (child_pid == 0) {
        printf("Child process (PID %d) is running.\n", getpid());
        sleep(2);
        printf("Child process (PID %d) is done.\n", getpid());
    } else if (child_pid > 0) {
        printf("Parent process (PID %d) is running.\n", getpid());
        sleep(1);
        printf("Parent process (PID %d) is done.\n", getpid());
    } else {
        perror("Fork failed");
        return 1;
    }
    return 0;
}
```

Q2

```
#include <stdio.h>

int main() {
```

```
int n = 3;
int ref[] = {12, 15, 12, 18, 6, 8, 11, 12, 19, 12, 6, 8, 12, 15, 19, 8};
int len = sizeof(ref) / sizeof(ref[0]);
int mem[n], faults = 0, future[len];

for (int i = 0; i < n; i++)
    mem[i] = -1;

printf("Page Reference String: ");
for (int i = 0; i < len; i++)
    printf("%d ", ref[i]);
printf("\n");

for (int i = 0; i < len; i++) {
    int p = ref[i], found = 0;
    for (int j = 0; j < n; j++) {
        if (mem[j] == p) {
            found = 1;
            break;
        }
    }
    if (!found) {
        faults++;
        int fIdx = 0;
        for (int j = 0; j < n; j++) {
            future[j] = len;
            for (int k = i + 1; k < len; k++) {
                if (mem[j] == ref[k]) {
                    future[j] = k;
                    break;
                }
            }
        }
        for (int j = 1; j < n; j++) {
            if (future[j] > future[fIdx]) {
                fIdx = j;
            }
        }
        mem[fIdx] = p;
        printf("Page Fault: Replaced Page %d with Page %d\n", mem[fIdx], p);
    }
    printf("Memory State: ");
    for (int j = 0; j < n; j++) {
        if (mem[j] != -1) {
            printf("%d ", mem[j]);
        }
    }
    printf("\n");
  }
  printf("Total Page Faults: %d\n", faults);
  return 0;
}
```

slip-11

Q1

```
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t child_pid = fork();

    if (child_pid == 0) {
        printf("Child Process ID: %d\n", getpid());
        printf("Hello World\n");
    } else if (child_pid > 0) {
        printf("Parent Process ID: %d\n", getpid());
        printf("Hi\n");
    } else {
        perror("Fork failed");
        return 1;
    }
    return 0;
}
```

Q2

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int n;

    printf("Enter the number of memory frames: ");

    scanf("%d", &n);


    int referenceString[] = {0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1};

    int referenceStringLength = sizeof(referenceString) / 
sizeof(referenceString[0]);

    int memory[n];

    int pageFaults = 0;

    int nextToReplace = 0;


    for (int i = 0; i < n; i++) {

        memory[i] = -1;

    }

    printf("Page Scheduling using FIFO:\n");


    for (int i = 0; i < referenceStringLength; i++) {

        int page = referenceString[i];

        int found = 0;


        for (int j = 0; j < n; j++) {

            if (memory[j] == page) {

                found = 1;

                break;

            }

        }
        if (!found) {
            pageFaults++;
            if (memory[nextToReplace] != -1) {
                printf("Page %d replaced by Page %d\n", memory[nextToReplace],
page);
            }
            memory[nextToReplace] = page;
            nextToReplace = (nextToReplace + 1) % n;
            printf("Page %d -> Page Fault\n", page);
        }
    }
    printf("Total Page Faults: %d\n", pageFaults);
    return 0;
}
```

slip12

Q1

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    pid_t child_pid = fork();

    if (child_pid == 0) {
        printf("Child process (PID: %d) is running, parent (PPID: %d)\n", getpid(),
getppid());
        sleep(5);
        printf("Child process (PID: %d) is done, new parent (PPID: %d)\n", getpid(),
getppid());
    }
    else if (child_pid > 0) {
        printf("Parent process (PID: %d) is running.\n", getpid());
        sleep(2);
        printf("Parent process (PID: %d) is done.\n", getpid());
    }
    else {
        perror("Fork failed");
        return 1;
    }
    return 0;
}
```

Q2

```
#include <stdio.h>

int main() {
    int n;
    int ref[] = {12, 15, 12, 18, 6, 8, 11, 12, 19, 12, 6, 8, 12, 15, 19, 8};
    int len = sizeof(ref) / sizeof(ref[0]);
    int faults = 0, mem[10], future[10];

    printf("Enter number of memory frames: ");
    scanf("%d", &n);

    for (int i = 0; i < n; i++)
        mem[i] = -1;

    printf("Page Reference String: ");
    for (int i = 0; i < len; i++)
        printf("%d ", ref[i]);
    printf("\n");

    for (int i = 0; i < len; i++) {
        int page = ref[i], found = 0;

        for (int j = 0; j < n; j++) {
            if (mem[j] == page) {
                found = 1;
                break;
            }
        }

        if (!found) {
            faults++;
            int fIdx = -1;

            for (int j = 0; j < n; j++) {
                if (mem[j] == -1) {
                    fIdx = j;
                    break;
                }
                future[j] = len;

                for (int k = i + 1; k < len; k++) {
                    if (mem[j] == ref[k]) {
                        future[j] = k;
                        break;
                    }
                }
            }

            if (fIdx == -1) {
                fIdx = 0;
                for (int j = 1; j < n; j++) {
```

```c
                    if (future[j] > future[fIdx]) {
                        fIdx = j;
                    }
                }
            }

            printf("Page %d replaced by Page %d\n", mem[fIdx], page);
            mem[fIdx] = page;
        }

        printf("Memory State: ");
        for (int j = 0; j < n; j++) {
            if (mem[j] != -1)
                printf("%d ", mem[j]);
            else
                printf("X ");
        }
        printf("\n");
    }

    printf("Total Page Faults: %d\n", faults);
    return 0;
}
```

slip13

Q1

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    pid_t child_pid = fork();

    if (child_pid == 0) {
        printf("Child process (PID %d) is running.\n", getpid());
        sleep(100);  // Simulate a long-running child process
        printf("Child process (PID %d) is done.\n", getpid());
    } else if (child_pid > 0) {
        printf("Parent process (PID %d) is running.\n", getpid());
        sleep(2);  // Parent sleeps for a short time
        printf("Parent process (PID %d) is done.\n", getpid());
    } else {
        perror("Fork failed");
        return 1;
    }
    return 0;
}
```

Q2

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <time.h>

typedef struct {
    int pid;
    int arrival;
    int burst;
    int wait;
    int turnaround;
    int completion;
    int remaining;
} Process;

void sortProcesses(Process p[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if ((p[j].arrival < p[i].arrival) ||
               (p[j].arrival == p[i].arrival && p[j].burst < p[i].burst)) {
                Process temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }
        }
    }
}

void calculateTimes(Process p[], int n) {
```

```
int currentTime = 0, completed = 0;
while (completed < n) {
    int idx = -1;
    int minBurst = 9999;

    for (int i = 0; i < n; i++) {
        if (p[i].arrival <= currentTime && p[i].remaining > 0) {
            if (p[i].remaining < minBurst) {
                minBurst = p[i].remaining;
                idx = i;
            }
        }
    }

    if (idx == -1) {
        currentTime++;
    } else {
        p[idx].remaining = 0;
        p[idx].completion = currentTime + p[idx].burst;
        p[idx].turnaround = p[idx].completion - p[idx].arrival;
        p[idx].wait = p[idx].turnaround - p[idx].burst;

        currentTime = p[idx].completion + 2;
        completed++;
    }
}
```

```
}

int main() {
    srand(time(0));
    int n;

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

    Process p[n];

    for (int i = 0; i < n; i++) {
        p[i].pid = i + 1;
        printf("Enter Arrival time and Initial CPU Burst time for Process P%d: ", i +
1);
        scanf("%d%d", &p[i].arrival, &p[i].burst);

        p[i].remaining = p[i].burst;
    }

    sortProcesses(p, n);
    calculateTimes(p, n);

    int totalWait = 0, totalTurnaround = 0;

    printf("\nProcess\tArrival\tBurst\tCompletion\tTurnaround\tWaiting\n");
    for (int i = 0; i < n; i++) {
        totalWait += p[i].wait;

        totalTurnaround += p[i].turnaround;

        printf("P%d\t%d\t%d\t%d\t\t%d\t\t%d\n", p[i].pid, p[i].arrival, p[i].burst,
p[i].completion, p[i].turnaround, p[i].wait);

    }


    printf("\nAverage Turnaround Time: %.2f\n", (float)totalTurnaround / n);

    printf("Average Waiting Time: %.2f\n", (float)totalWait / n);


    return 0;

}
```

slip14

Q1

```
#include <stdio.h>
#include <time.h>

int main() {
    clock_t start_time, end_time;
    double execution_time;

    start_time = clock();
    for (int i = 0; i < 1000000; i++) {
        int result = i * i;  // Sample computation
    }
    end_time = clock();

    execution_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;
    printf("Execution time: %f seconds\n", execution_time);
    return 0;
}
```

Q2

```
#include <stdio.h>
int main() {
    int n = 3;
    int referenceString[] = {0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1};
    int len = sizeof(referenceString) / sizeof(referenceString[0]);
    int memory[n];
int faults = 0, i = 0;
for (int j = 0; j < n; j++)
    memory[j] = -1;
printf("Page Scheduling using FIFO:\n");
for (int r = 0; r < len; r++) {
    int page = referenceString[r];
    int found = 0;
    for (int j = 0; j < n; j++) {
        if (memory[j] == page) {
            found = 1;
            break;
        }
    }
    if (!found) {
        faults++;
        if (memory[i % n] != -1) {
            printf("Page %d replaced by Page %d\n", memory[i % n], page);
        }
        memory[i % n] = page;
        i++;
        printf("Page %d -> Page Fault\nMemory: ", page);
        for (int j = 0; j < n; j++) {
            if (memory[j] == -1)
                printf("- ");
            else
                printf("%d ", memory[j]);
            }
            printf("\n");
        }
    }
    printf("Total Page Faults: %d\n", faults);
    return 0;
}
```

slip 15

Q1

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    pid_t child_pid = fork();

    if (child_pid < 0) {
        perror("Fork failed");
        exit(1);
    } else if (child_pid == 0) {
        printf("Child Process (PID %d) is running and executing 'ls'.\n", getpid());
        execl("/bin/ls", "ls", (char *)NULL);
        perror("execl() failed");
        exit(1);
    } else {
        printf("Parent Process (PID %d) is going to sleep.\n", getpid());
        sleep(2);  // Parent sleeps for 2 seconds
        printf("Parent Process (PID %d) is awake.\n", getpid());
    }
    return 0;
}
```

Q2

```
#include <stdio.h>
```

```
#include <stdlib.h>


int main() {
   int n, p = 0, r, q = 0;
   printf("Enter the number of memory frames: ");
   scanf("%d", &n);
   int frames[n];


   // Reference String: You can modify this as needed
   int referenceString[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};
   int referenceStringLength = sizeof(referenceString) /
sizeof(referenceString[0]);


   printf("Reference String: ");
   for (int i = 0; i < referenceStringLength; i++) {
      printf("%d ", referenceString[i]);
   }
   printf("\n");


   for (int i = 0; i < n; i++) {
      frames[i] = -1;  // Initialize frames to -1
   }


   for (int i = 0; i < referenceStringLength; i++) {
      r = referenceString[i];
      int found = 0;
```

```
        // Check if page is in frames
        for (int j = 0; j < n; j++) {
            if (frames[j] == r) {
                found = 1;
                break;
            }
        }

        // If page is not found, replace using LRU
        if (!found) {
            printf("Page %d -> Page Fault\n", r);
            frames[p % n] = r;  // Replace the frame
            p++;
        }
        q++;
    }
    printf("Total Page Faults: %d\n", p);
    return 0;
}
```

slip16

Q1

```
#include <stdio.h>
#include <time.h>

int main() {
   clock_t start_time, end_time;
   double execution_time;

   start_time = clock();

   for (int i = 0; i < 1000000; i++) {
      int result = i * i; // Sample operation
   }

   end_time = clock();
   execution_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;

   printf("Execution time: %f seconds\n", execution_time);
   return 0;
}
```

Q2

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int n, faults = 0, r, q = 0;
    printf("Enter the number of memory frames: ");
    scanf("%d", &n);

    int frames[n];
    int nextUse[n];
    int referenceString[] = {12, 15, 12, 18, 6, 8, 11, 12, 19, 12, 6, 8, 12, 15, 19, 8};
    int referenceStringLength = sizeof(referenceString) /
sizeof(referenceString[0]);

    for (int i = 0; i < n; i++) frames[i] = -1;

    printf("Page Scheduling using OPT:\n");
    while (q < referenceStringLength) {
        r = referenceString[q];
        int found = 0;

        for (int i = 0; i < n; i++) {
            if (frames[i] == r) {
                found = 1;
                break;
            }
        }

        if (!found) {
            int toReplace = -1;
            for (int i = 0; i < n; i++) {
```

```
    nextUse[i] = q + 1;
    for (int j = q + 1; j < referenceStringLength; j++) {
        if (frames[i] == referenceString[j]) {
            nextUse[i] = j;
            break;
        }
    }
}

for (int i = 0; i < n; i++) {
    if (nextUse[i] == q + 1) {
        toReplace = i;
        break;
    }
}

if (toReplace == -1) {
    toReplace = 0;
    for (int i = 1; i < n; i++) {
        if (nextUse[i] > nextUse[toReplace]) {
            toReplace = i;
        }
    }
}

frames[toReplace] = r;
```

```
            faults++;
            printf("Page %d -> Page Fault\n", r);
        }
        q++;
    }

    printf("Total Page Faults: %d\n", faults);
    return 0;
}
```

slip17

Q1

```
#include <stdio.h>


int main() {
    int allocated[3][3] = {{1, 2, 3}, {2, 3, 4}, {3, 4, 5}};
    int max_demand[3][3] = {{4, 4, 4}, {5, 5, 5}, {6, 6, 6}};
    int available[3] = {0};


    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            available[j] += max_demand[i][j] - allocated[i][j];
        }
    }


    printf("Minimum resources needed to avoid deadlock: ");
    for (int i = 0; i < 3; i++) {
        printf("%d ", available[i]);
    }
    printf("\n");
    return 0;
}
```

Q2

```
#include <stdio.h>
int main() {
    int n = 3, ref[] = {12, 15, 12, 18, 6, 8, 11, 12, 19, 12, 6, 8, 12, 15, 19, 8};
int len = sizeof(ref) / sizeof(ref[0]), mem[n], faults = 0, future[len];

for (int i = 0; i < n; i++) mem[i] = -1;

for (int i = 0; i < len; i++) {
    int p = ref[i], found = 0;
    for (int j = 0; j < n; j++) {
        if (mem[j] == p) {
            found = 1;
            break;
        }
    }

    if (!found) {
        faults++;
        for (int j = 0; j < n; j++) future[j] = len;
        for (int j = 0; j < n; j++) {
            for (int k = i + 1; k < len; k++) {
                if (mem[j] == ref[k]) {
                    future[j] = k;
                    break;
                }
            }
        }

        int fIdx = 0;
```

```
            for (int j = 0; j < n; j++) {
                if (future[j] > future[fIdx]) fIdx = j;
            }
            mem[fIdx] = p;

            printf("Page %d -> Page Fault\nMemory: ", p);
            for (int j = 0; j < n; j++) printf("%d ", mem[j]);
            printf("\n");
        }
    }

    printf("Total Page Faults: %d\n", faults);
    return 0;
}
```

slip18

Q1

```
#include <stdio.h>

int main() {
    int n, m;
    printf("Enter number of processes and resources: ");
    scanf("%d %d", &n, &m);

    int alloc[n][m], max[n][m], need[n][m];
    printf("Enter allocation and max matrices:\n");

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            scanf("%d %d", &alloc[i][j], &max[i][j]);
            need[i][j] = max[i][j] - alloc[i][j];
        }
    }

    printf("Need matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            printf("%d ", need[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Q2

```
#include <stdio.h>
int main() {
    int n, ref[] = {12, 15, 12, 18, 6, 8, 11, 12, 19, 12, 6, 8, 12, 15, 19, 8};
    int len = sizeof(ref) / sizeof(ref[0]), faults = 0;

    printf("Enter number of memory frames: ");
    scanf("%d", &n);

    int mem[n], future[n];
    for (int i = 0; i < n; i++) mem[i] = -1;

    for (int i = 0; i < len; i++) {
        int p = ref[i], found = 0;
        for (int j = 0; j < n; j++) {
            if (mem[j] == p) {
                found = 1;
                break;
            }
        }

        if (!found) {
            faults++;
            for (int j = 0; j < n; j++) future[j] = len;
            for (int j = 0; j < n; j++) {
                for (int k = i + 1; k < len; k++) {
                    if (mem[j] == ref[k]) {
                        future[j] = k;
                        break;
                    }
                }
            }

            int fIdx = 0;
            for (int j = 0; j < n; j++) {
                if (future[j] > future[fIdx]) fIdx = j;
            }
            mem[fIdx] = p;

            printf("Page %d -> Page Fault\nMemory: ", p);
            for (int j = 0; j < n; j++) printf("%d ", mem[j]);
            printf("\n");
        }
    }

    printf("Total Page Faults: %d\n", faults);
    return 0;
}
```

slip19

Q1

```
#include <stdio.h>

int main() {
   int n, m;
   printf("Enter number of processes and resources: ");
   scanf("%d %d", &n, &m);

   int alloc[n][m], max[n][m], need[n][m];
   printf("Enter allocation and max matrices:\n");

   for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
         scanf("%d %d", &alloc[i][j], &max[i][j]);
         need[i][j] = max[i][j] - alloc[i][j];
      }
   }

   printf("Need matrix:\n");
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
         printf("%d ", need[i][j]);
      }
      printf("\n");
```

```
    }
    return 0;
}
```

Q2

```
#include <stdio.h>
#include <stdlib.h>

struct Process {
    int pid;
    int arrival_time;
    int burst_time;
    int priority;
};

void sort_by_priority(struct Process processes[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (processes[i].priority > processes[j].priority) {
                struct Process temp = processes[i];
                processes[i] = processes[j];
                processes[j] = temp;
            }
        }
    }
}
```

```
int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    struct Process processes[n];
    int total_turnaround_time = 0;
    int total_waiting_time = 0;

    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        printf("Enter arrival time for Process %d: ", i + 1);
        scanf("%d", &processes[i].arrival_time);
        printf("Enter burst time for Process %d: ", i + 1);
        scanf("%d", &processes[i].burst_time);
        printf("Enter priority for Process %d: ", i + 1);
        scanf("%d", &processes[i].priority);
    }

    sort_by_priority(processes, n);

    printf("\nGantt Chart:\n");
    int current_time = 0;
    for (int i = 0; i < n; i++) {
        printf("P%d -> ", processes[i].pid);
        total_waiting_time += current_time - processes[i].arrival_time;
        current_time += processes[i].burst_time;

        total_turnaround_time += current_time - processes[i].arrival_time;

    }


    printf("\n\nAverage Turnaround Time: %.2f", (float)total_turnaround_time /
n);

    printf("\nAverage Waiting Time: %.2f\n", (float)total_waiting_time / n);


    return 0;

}
```

slip20

Q1

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>


int main() {

    pid_t child_pid = fork();


    if (child_pid < 0) {

        perror("Fork failed");
        exit(1);
    }

    if (child_pid == 0) {
        execl("/bin/ls", "ls", NULL);
        perror("Execl failed");
        exit(1);
    } else {
        printf("Parent process is going to sleep for 3 seconds...\n");
        sleep(3);
        wait(NULL);
        printf("Parent process has woken up.\n");
    }

    return 0;
}
```

Q2

```
#include <stdio.h>
#include <stdlib.h>

int findLRU(int frames[], int n, int page, int current) {
    int index = -1;
    int farthest = current;

    for (int i = 0; i < n; i++) {
        if (frames[i] == -1) {
            return i;
        }

        for (int j = current - 1; j >= 0; j--) {
            if (frames[i] == page) {
                break;
            }
            if (frames[i] == frames[j]) {
                if (j < farthest) {
                    farthest = j;
                    index = i;
                }
                break;
            }
        }
    }
    return index;
}

int main() {
    int n = 3;
    int referenceString[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};
    int referenceStringLength = sizeof(referenceString) /
sizeof(referenceString[0]);
    int frames[n];
    int pageFaults = 0;
```

```
for (int i = 0; i < n; i++) {
    frames[i] = -1;
}

printf("Page Scheduling using LRU:\n");

for (int i = 0; i < referenceStringLength; i++) {
    int page = referenceString[i];
    int found = 0;

    for (int j = 0; j < n; j++) {
        if (frames[j] == page) {
            found = 1;
            break;
        }
    }

    if (!found) {
        int replaceIndex = findLRU(frames, n, page, i);
        frames[replaceIndex] = page;
        pageFaults++;
        printf("Page %d -> Page Fault\n", page);
    }
}

printf("Total Page Faults: %d\n", pageFaults);
```

```
    return 0;
}
```