EXPERIMENT-1

Experiment-1: Develop a c program to implement the Process system calls (fork (), exec(), wait(), create process, terminate process).

a) fork():

Code:

#include #include #include #include int main(int argc, char **argv) { pid_t pid; pid = fork(); if(pid==0) { printf("It is the child process and pid is %d\n",getpid()); exit(0); } else if(pid > 0) { printf("It is the parent process and pid is %d\n",getpid()); } else { printf("Error while forking\n"); exit(EXIT_FAILURE); } return 0; }

Expected Output:

b) exec():

Code:

#include #include #include #include #include main(void) { pid_t pid = 0; int status; pid = fork(); if (pid == 0) { printf("I am the child."); execl("/bin/ls", "ls", "-l", "/home/ubuntu/", (char *) 0); perror("In exec(): "); } if (pid > 0) { printf("I am the parent, and the child is %d.\n", pid); pid = wait(&status); printf("End of process %d: ", pid); if (WIFEXITED(status)) { printf("The process ended with exit(%d). \n", WEXITSTATUS(status)); } if (WIFSIGNALED(status)) { printf("The process ended with kill -%d. \n", WTERMSIG(status)); } } if (pid < 0) { perror("In fork():"); } exit(0); }

Expected Output:

c) wait()

Code:

#include // printf() #include // exit() #include // pid_t #include // wait() #include // fork int main(int argc, char **argv) { pid_t pid; pid = fork(); if(pid==0) { printf("It is the child process and pid is %d\n",getpid()); int i=0; for(i=0;i<8;i++) { printf("%d\n",i); } exit(0); } else if(pid > 0) { printf("It is the parent process and pid is %d\n",getpid()); int status; wait(&status); printf("Child is reaped\n"); } else { printf("Error in forking..\n"); exit(EXIT_FAILURE); } return 0; }

Expected Output:

EXPERIMENT-2

Experiment-2: Simulate the following CPU scheduling algorithms to find turnaround time and waiting time a) FCFS b) SJF c) Round Robin d) Priority.

a) FCFS

Code:

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

struct process {
  int pid, at, bt, ct, wt, tat;
};

int cmp_at(const void *a, const void *b) {
  const struct process *p1 = a, *p2 = b;
  if (p1->at != p2->at) return p1->at - p2->at;
  if (p1->bt != p2->bt) return p1->bt - p2->bt;
  return p1->pid - p2->pid;
}

int main() {
  int n;
  printf("Enter Number of Processes: ");
  scanf("%d", &n);

  struct process *p = malloc(n * sizeof(struct process));

  for (int i = 0; i < n; i++) {
    p[i].pid = i + 1;
    printf("Process %d:\n  Arrival Time: ", i + 1);
    scanf("%d", &p[i].at);
    printf("  Burst Time: ");
    scanf("%d", &p[i].bt);
    p[i].ct = p[i].wt = p[i].tat = 0;
  }

  qsort(p, n, sizeof(struct process), cmp_at);

  int current_time = 0, twt = 0, ttat = 0;
  for (int i = 0; i < n; i++) {
    if (p[i].at > current_time) current_time = p[i].at;
    p[i].ct = current_time + p[i].bt;
    p[i].wt = current_time - p[i].at;
    p[i].tat = p[i].ct - p[i].at;
    current_time = p[i].ct;
```

```
    twt += p[i].wt;
    ttat += p[i].tat;
  }

  printf("\nPID\tAT\tBT\tCT\tWT\tTAT\n");
  for (int i = 0; i < n; i++)
    printf("P%d\t%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].at, p[i].bt, p[i].ct, p[i].wt,
p[i].tat);

  printf("\nAverage Waiting Time: %.2f\n", (float)twt / n);
  printf("Average Turnaround Time: %.2f\n", (float)ttat / n);

  free(p);
  return 0;
}
```

Expected output:

b) SJF

Code:

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

struct process{
  int pid, at, bt, ct, wt, tat, is_completed;
};

int cmp_at(const void *a, const void *b){
  const struct process *p1 = a, *p2 = b;
  if(p1->at != p2->at) return p1->at - p2->at;
  if(p1->bt != p2->bt) return p1->bt - p2->bt;
  return p1->pid - p2->pid;
}

int main()
{
  int n;
  printf("Enter Number of Processes\n");
  scanf("%d",&n);

  struct process *p = malloc(n * sizeof(struct process));

  for(int i=0; i<n; i++){
    p[i].pid = i+1;
    printf("Process -%d\nArrival time ",i+1);
```

```
    scanf("%d",&p[i].at);
    printf("Burst time ");
    scanf("%d",&p[i].bt);
    p[i].wt = p[i].ct = p[i].tat = p[i].is_completed = 0;
  }

  qsort(p, n, sizeof(struct process), cmp_at);

  int completed = 0, current_time = 0, twt = 0, ttat = 0;

  while(completed < n){
    int idx = -1, min_bt = 1000000000;

    for(int i=0; i<n; i++){
      if(p[i].at <= current_time && !p[i].is_completed){
        if(p[i].bt < min_bt){
          min_bt = p[i].bt;
          idx = i;
        }
        if(p[i].bt == min_bt && p[i].at < p[idx].at){
          idx = i;
        }
      }
    }

    if(idx == -1){
      current_time++;
    }else{
      current_time += p[idx].bt;
      p[idx].ct = current_time;
      p[idx].tat = p[idx].ct - p[idx].at;
      p[idx].wt = p[idx].tat - p[idx].bt;
      ttat += p[idx].tat;
      twt += p[idx].wt;
      p[idx].is_completed = 1;
      completed++;
    }
  }

  for(int i=0; i<n; i++){
    printf("Process\t%d\tarrival time\t%d\tburst time\t%d\twaiting
time\t%d\ttat\t%d\n",
        p[i].pid, p[i].at, p[i].bt, p[i].wt, p[i].tat);
  }

  printf("Average waiting time %.2f\n", (float)twt/n);
  printf("Average Turnaround time %.2f\n", (float)ttat/n);
```

```
    free(p);
    return 0;
}
```

Expected Output:

c) Round Robin

code:

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

struct process {
    int pid, at, bt, rem_bt, ct, wt, tat, is_completed;
};

int cmp_at(const void *a, const void *b) {
    const struct process *p1 = a, *p2 = b;
    if (p1->at != p2->at) return p1->at - p2->at;
    if (p1->bt != p2->bt) return p1->bt - p2->bt;
    return p1->pid - p2->pid;
}

typedef struct {
    int *data, cap, front, size;
} Queue;

int queue_init(Queue *q, int initial_cap) {
    if (initial_cap <= 0) initial_cap = 4;
    q->data = malloc(initial_cap * sizeof(int));
    if (!q->data) return -1;
    q->cap = initial_cap;
    q->front = q->size = 0;
    return 0;
}

void queue_free(Queue *q) {
    free(q->data);
    q->data = NULL;
    q->cap = q->front = q->size = 0;
}

int queue_ensure_capacity(Queue *q) {
    if (q->size < q->cap) return 0;
    int newcap = q->cap * 2;
```

```
    int *newdata = malloc(newcap * sizeof(int));
    if (!newdata) return -1;
    for (int i = 0; i < q->size; ++i) {
        newdata[i] = q->data[(q->front + i) % q->cap];
    }
    free(q->data);
    q->data = newdata;
    q->cap = newcap;
    q->front = 0;
    return 0;
}

int queue_enqueue(Queue *q, int value) {
    if (queue_ensure_capacity(q) != 0) return -1;
    int back = (q->front + q->size) % q->cap;
    q->data[back] = value;
    q->size++;
    return 0;
}

int queue_dequeue(Queue *q) {
    if (q->size == 0) return -1;
    int val = q->data[q->front];
    q->front = (q->front + 1) % q->cap;
    q->size--;
    return val;
}

int queue_empty(Queue *q) {
    return q->size == 0;
}

int main(void) {
    int n;
    printf("Enter Number of Processes: ");
    scanf("%d", &n);

    struct process *p = malloc(n * sizeof(struct process));

    for (int i = 0; i < n; ++i) {
        p[i].pid = i + 1;
        printf("Process %d:\n  Arrival Time: ", i + 1);
        scanf("%d", &p[i].at);
        printf("  Burst Time: ");
        scanf("%d", &p[i].bt);
        p[i].rem_bt = p[i].bt;
        p[i].ct = p[i].wt = p[i].tat = p[i].is_completed = 0;
```

```
  }

  qsort(p, n, sizeof(struct process), cmp_at);

  int time_quantum;
  printf("Enter time quantum: ");
  scanf("%d", &time_quantum);

  Queue q;
  queue_init(&q, n + 4);

  int next = 0, completed = 0;
  int current_time = (n > 0) ? p[0].at : 0;

  while (next < n && p[next].at <= current_time) {
    queue_enqueue(&q, next);
    next++;
  }

  while (completed < n) {
    if (queue_empty(&q)) {
      if (next < n) {
        current_time = p[next].at;
        while (next < n && p[next].at <= current_time) {
          queue_enqueue(&q, next);
          next++;
        }
        continue;
      } else {
        break;
      }
    }

    int idx = queue_dequeue(&q);
    if (idx < 0) continue;

    int exec = (p[idx].rem_bt < time_quantum) ? p[idx].rem_bt : time_quantum;
    p[idx].rem_bt -= exec;
    current_time += exec;

    while (next < n && p[next].at <= current_time) {
      queue_enqueue(&q, next);
      next++;
    }

    if (p[idx].rem_bt == 0) {
      p[idx].ct = current_time;
```

```
            p[idx].tat = p[idx].ct - p[idx].at;
            p[idx].wt = p[idx].tat - p[idx].bt;
            p[idx].is_completed = 1;
            completed++;
        } else {
            queue_enqueue(&q, idx);
        }
    }

    double total_wt = 0.0, total_tat = 0.0;
    printf("\nPID\tAT\tBT\tCT\tTAT\tWT\n");
    for (int i = 0; i < n; ++i) {
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].at, p[i].bt, p[i].ct, p[i].tat,
p[i].wt);
        total_wt += p[i].wt;
        total_tat += p[i].tat;
    }
    printf("Average WT = %.2f\n", total_wt / n);
    printf("Average TAT = %.2f\n", total_tat / n);

    queue_free(&q);
    free(p);
    return 0;
}
```

Expected output:

d) Priority

code:

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

struct process {
    int pid, at, bt, ct, wt, tat, priority;
};

int cmp_at(const void *a, const void *b) {
    const struct process *p1 = (const struct process *)a, *p2 = (const struct process
*)b;
    if (p1->at != p2->at) return p1->at - p2->at;
    if (p1->priority != p2->priority) return p1->priority - p2->priority;
    return p1->pid - p2->pid;
}

int main(void) {
```

```c
  int n;
  printf("Enter Number of Processes: ");
  scanf("%d", &n);

  struct process *p = malloc(n * sizeof(struct process));

  for (int i = 0; i < n; i++) {
    p[i].pid = i + 1;
    printf("Process %d:\n", i + 1);
    printf("  Arrival Time: ");
    scanf("%d", &p[i].at);
    printf("  Burst Time: ");
    scanf("%d", &p[i].bt);
    printf("  Priority: ");
    scanf("%d", &p[i].priority);
  }

  qsort(p, n, sizeof(struct process), cmp_at);

  int current_time = 0, completed = 0;
  int *is_completed = calloc(n, sizeof(int));

  while (completed < n) {
    int idx = -1, highest_priority = 999999;

    for (int i = 0; i < n; i++) {
      if (!is_completed[i] && p[i].at <= current_time) {
        if (p[i].priority < highest_priority ||
          (p[i].priority == highest_priority && p[i].at < p[idx].at) ||
          (p[i].priority == highest_priority && p[i].at == p[idx].at && p[i].pid <
p[idx].pid)) {
          highest_priority = p[i].priority;
          idx = i;
        }
      }
    }

    if (idx == -1) {
      current_time++;
      continue;
    }

    current_time += p[idx].bt;
    p[idx].ct = current_time;
    p[idx].tat = p[idx].ct - p[idx].at;
    p[idx].wt = p[idx].tat - p[idx].bt;
    is_completed[idx] = 1;
```

```
        completed++;
    }

    double total_wt = 0.0, total_tat = 0.0;
    printf("\nPID\tAT\tBT\tPriority\tCT\tTAT\tWT\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t\t%d\t%d\t%d\n",
               p[i].pid, p[i].at, p[i].bt, p[i].priority, p[i].ct, p[i].tat, p[i].wt);
        total_wt += p[i].wt;
        total_tat += p[i].tat;
    }
    printf("Average WT = %.2f\nAverage TAT = %.2f\n", total_wt / n, total_tat / n);

    free(is_completed);
    free(p);
    return 0;
}
```

Expected Output:

EXPERIMENT-3

Experiment-3: Develop a C program to simulate producer-consumer problem using semaphores

code:

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

#define BUFFER_SIZE 10

// Semaphores and mutex
sem_t empty;  // Count of empty slots
sem_t full;   // Count of full slots
pthread_mutex_t mutex;  // Mutex for critical section
```

```
// Buffer
int buffer[BUFFER_SIZE];
int in = 0;   // Index for producer
int out = 0;  // Index for consumer

// Function to produce an item and add it to the buffer
void* producer(void* arg) {
    int item = *((int*)arg);

    // Wait if buffer is full
    sem_wait(&empty);

    // Lock the critical section
    pthread_mutex_lock(&mutex);

    // Add item to buffer
    buffer[in] = item;
    printf("\nProducer produces item %d at position %d", item, in);
    in = (in + 1) % BUFFER_SIZE;

    // Unlock the critical section
    pthread_mutex_unlock(&mutex);

    // Signal that buffer has one more full slot
    sem_post(&full);

    return NULL;
}
```

```
// Function to consume an item and remove it from buffer
void* consumer(void* arg) {
    int item;

    // Wait if buffer is empty
    sem_wait(&full);

    // Lock the critical section
    pthread_mutex_lock(&mutex);

    // Remove item from buffer
    item = buffer[out];
    printf("\nConsumer consumes item %d from position %d", item, out);
    out = (out + 1) % BUFFER_SIZE;

    // Unlock the critical section
    pthread_mutex_unlock(&mutex);

    // Signal that buffer has one more empty slot
    sem_post(&empty);

    return NULL;
}

// Driver Code
int main() {
    int choice;
    int item_count = 0;
    pthread_t tid;
```

```
// Initialize semaphores and mutex
sem_init(&empty, 0, BUFFER_SIZE);  // Initially all slots are empty
sem_init(&full, 0, 0);             // Initially no slots are full
pthread_mutex_init(&mutex, NULL);

printf("\n1. Press 1 for Producer");
printf("\n2. Press 2 for Consumer");
printf("\n3. Press 3 for Exit\n");

while (1) {
    printf("\nEnter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
        case 1: {
            // Check if buffer is full
            int empty_val;
            sem_getvalue(&empty, &empty_val);

            if (empty_val > 0) {
                item_count++;
                int* item = malloc(sizeof(int));
                *item = item_count;

                pthread_t prod_thread;
                pthread_create(&prod_thread, NULL, producer, item);
                pthread_detach(prod_thread);
            } else {
                printf("Buffer is full!");
            }
```

```
            break;
        }

        case 2: {
            // Check if buffer is empty
            int full_val;
            sem_getvalue(&full, &full_val);

            if (full_val > 0) {
                pthread_t cons_thread;
                pthread_create(&cons_thread, NULL, consumer, NULL);
                pthread_detach(cons_thread);
            } else {
                printf("Buffer is empty!");
            }
            break;
        }

        case 3:
            // Cleanup
            sem_destroy(&empty);
            sem_destroy(&full);
            pthread_mutex_destroy(&mutex);
            printf("\nExiting...\n");
            exit(0);

        default:
            printf("Invalid choice!");
    }
```

```
        // Small delay to allow threads to complete
        usleep(100000);
    }

    return 0;
}
```

EXPERIMENT-4

Experiment-4: Develop a C program which demonstrates Inter-process communication between a reader process and a writer process. Use mkfifo, open, read, write and close APIs in your program.

4-a code:

```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>

int main()
{
    int fd;

    // FIFO file path
    char *myfifo = "/tmp/myfifo";

    // Creating the named file(FIFO)
    // Check if it already exists
    if (mkfifo(myfifo, 0666) == -1) {
        if (errno != EEXIST) {
```

```
        perror("mkfifo");
        return 1;
    }
}

char arr1[80], arr2[80];

while (1)
{
    // Open FIFO for write only
    fd = open(myfifo, O_WRONLY);
    if (fd == -1) {
        perror("open for write");
        return 1;
    }

    // Take an input string from user.
    printf("You: ");
    fflush(stdout);
    if (fgets(arr2, 80, stdin) == NULL) {
        break;
    }

    // Write the input string on FIFO and close it
    if (write(fd, arr2, strlen(arr2) + 1) == -1) {
        perror("write");
        close(fd);
        return 1;
    }
    close(fd);
```

```
        // Open FIFO for Read only
        fd = open(myfifo, O_RDONLY);
        if (fd == -1) {
            perror("open for read");
            return 1;
        }

        // Read from FIFO
        ssize_t bytes_read = read(fd, arr1, sizeof(arr1));
        if (bytes_read == -1) {
            perror("read");
            close(fd);
            return 1;
        }

        // Print the read message
        printf("User2: %s\n", arr1);
        close(fd);
    }

    return 0;
}
```

4-b code:

```
// C program to implement one side of FIFO
// This side reads first, then writes
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
```

```
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>

int main()
{
    int fd1;

    // FIFO file path
    char *myfifo = "/tmp/myfifo";

    // Creating the named file(FIFO)
    // Check if it already exists
    if (mkfifo(myfifo, 0666) == -1) {
        if (errno != EEXIST) {
            perror("mkfifo");
            return 1;
        }
    }

    char str1[80], str2[80];

    while (1)
    {
        // First open in read only and read
        fd1 = open(myfifo, O_RDONLY);
        if (fd1 == -1) {
            perror("open for read");
            return 1;
```

```
}

ssize_t bytes_read = read(fd1, str1, sizeof(str1));
if (bytes_read == -1) {
    perror("read");
    close(fd1);
    return 1;
}

// Ensure null termination
str1[bytes_read < 80 ? bytes_read : 79] = '\0';

// Print the read string and close
printf("User1: %s", str1);
close(fd1);

// Now open in write mode and write
// string taken from user.
printf("You: ");
fflush(stdout);

if (fgets(str2, 80, stdin) == NULL) {
    break;
}

fd1 = open(myfifo, O_WRONLY);
if (fd1 == -1) {
    perror("open for write");
    return 1;
}
```

```
        if (write(fd1, str2, strlen(str2) + 1) == -1) {
            perror("write");
            close(fd1);
            return 1;
        }
        close(fd1);
    }

    return 0;
}
```

Expected Output:

EXPERIMENT-5

Experiment-5: Develop a C program to simulate Bankers Algorithm for DeadLock Avoidance.

Code:

```
#include<stdio.h>
#include<string.h>
void main()
{
    int alloc[10][10],max[10][10];
    int avail[10],work[10],total[10];
    int i,j,k,n,need[10][10];
    int m;
    int count=0,c=0;
    char finish[10];
    printf("Enter the no. of processes and resources:");
    scanf("%d%d",&n,&m);
```

```
for(i=0;i<n;i++)
    finish[i]='n';
printf("Enter the claim matrix:\n");
for(i=0;i<n;i++)
    for(j=0;j<m;j++)
        scanf("%d",&max[i][j]);
printf("Enter the allocation matrix:\n");
for(i=0;i<n;i++)
    for(j=0;j<m;j++)
        scanf("%d",&alloc[i][j]);
printf("Resource vector:");
for(i=0;i<m;i++)
    scanf("%d",&total[i]);
for(i=0;i<m;i++)
    avail[i]=0;
for(i=0;i<n;i++)
    for(j=0;j<m;j++)
        avail[j]+=alloc[i][j];
for(i=0;i<m;i++)
    work[i]=total[i]-avail[i];  // simplified logic
for(i=0;i<n;i++)
    for(j=0;j<m;j++)
        need[i][j]=max[i][j]-alloc[i][j];
A:for(i=0;i<n;i++)
{
    c=0;
    for(j=0;j<m;j++)
    {  // added missing braces
        if((need[i][j]<=work[j])&&(finish[i]=='n'))
            c++;
```

```
        }
        if(c==m) {
            printf("All the resources can be allocated to Process %d", i+1);
            printf("\n\nAvailable resources are:");
            for(k=0;k<m;k++)
            {
                work[k]+=alloc[i][k];
                printf("%4d",work[k]);
            }
            printf("\n");
            finish[i]='y';
            printf("\nProcess %d executed?:%c \n",i+1,finish[i]);
            count++;
            goto A;  // restart search after finding a process
        }
    }
    if(count==n)
        printf("\n System is in safe state");
    else
        printf("\n System is NOT in safe state");
}
```

EXPERIMENT-6

Experiment 6: Contiguous Memory Allocation Techniques

Program Statement:

Develop a C program to simulate the following contiguous memory allocation techniques:

1. Best Fit
2. First Fit
3. Worst Fit

a) Best Fit

Code;

```
#include <stdio.h> #define max 25 int main() { int frag[max], b[max], f[max], i, j, nb, nf, temp, lowest = 10000; static int bf[max], ff[max]; printf("\nEnter the number of blocks:"); scanf("%d", &nb); printf("Enter the number of files:"); scanf("%d", &nf); printf("\nEnter the size of the blocks:-\n"); for (i = 1; i <= nb; i++) { printf("Block %d:", i); scanf("%d", &b[i]); } printf("Enter the size of the files :-\n"); for (i = 1; i <= nf; i++) { printf("File %d:", i); scanf("%d", &f[i]); } for (i = 1; i <= nf; i++) { lowest = 10000; ff[i] = 0; // Initialize to 0 (no block allocated) for (j = 1; j <= nb; j++) { if (bf[j] != 1) // If block is not allocated { temp = b[j] - f[i]; if (temp >= 0) // If file fits in block { if (lowest > temp) // If this is the best fit so far { ff[i] = j; lowest = temp; } } } } // Only allocate if a suitable block was found if (ff[i] != 0) { frag[i] = lowest; // Store fragment bf[ff[i]] = 1; // Mark block as allocated } else { frag[i] = -1; // Indicate allocation failure } } printf("\nFile No\tFile Size \tBlock No\tBlock Size\tFragment"); for (i = 1; i <= nf; i++) { if (ff[i] != 0) printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i, f[i], ff[i], b[ff[i]], frag[i]); else printf("\n%d\t\t%d\t\tNot Allocated", i, f[i]); } printf("\n"); return 0; }
```

Expected output:

b) First fit:

Code:

```
#include <stdio.h> #define MAX 25 // Function to allocate memory to blocks as per First fit algorithm void firstFit(int blockSize[], int m, int processSize[], int n) { int i, j; // Stores block id of the block allocated to a process int allocation[n]; // Create a copy of blockSize to preserve original values int availableBlock[m]; for (i = 0; i < m; i++) { availableBlock[i] = blockSize[i]; } // Initially no block is assigned to any process for (i = 0; i < n; i++) { allocation[i] = -1; } // Pick each process and find suitable blocks // according to its size and assign to it for (i = 0; i < n; i++) // n -> number of processes { for (j = 0; j < m; j++) // m -> number of blocks { if (availableBlock[j] >= processSize[i]) { // Allocating block j to the ith process allocation[i] = j; // Reduce available memory in this block availableBlock[j] -= processSize[i]; break; // Go to the next process in the queue } } } printf("\nProcess No.\tProcess Size\tBlock no.\n"); for (i = 0; i < n; i++) { printf(" %d\t\t%d KB\t\t", i + 1, processSize[i]); if (allocation[i] != -1) printf("%d\n", allocation[i] + 1); else printf("Not Allocated\n"); } } // Driver code int main() { int nb, np; int blockSize[MAX]; int processSize[MAX]; int m, n; printf("Enter number of Processes: "); scanf("%d", &np); printf("Enter number of Blocks: "); scanf("%d", &nb); for(int i = 0; i < np; i++){ printf("Enter size of Process %d: ", i + 1); scanf("%d", &processSize[i]); } for(int i = 0; i < nb; i++){ printf("Enter size of Block %d: ", i + 1); scanf("%d", &blockSize[i]); } m = nb; n = np; firstFit(blockSize, m, processSize, n); return 0; }
```

Expected output:

c) Worst fit:

Code:

```
#include <stdio.h> #define MAX 25 int main(void) { int frag[MAX], b[MAX], f[MAX]; int i, j, nb, nf, temp, highest; static int bf[MAX], ff[MAX]; printf("\n\tMemory Management Scheme - Worst Fit"); printf("\nEnter the number of blocks: "); scanf("%d", &nb); printf("Enter the number of files: "); scanf("%d", &nf); // Optional: basic bounds check if (nb > MAX || nf > MAX) { printf("Error: nb and nf must each be <= %d\n", MAX); return 1; } printf("\nEnter the size of the blocks:\n"); for (i = 0; i < nb; i++) { printf("Block %d: ", i + 1); scanf("%d", &b[i]); } printf("Enter the size of the files:\n"); for (i
```

= 0; i < nf; i++) { printf("File %d: ", i + 1); scanf("%d", &f[i]); } // Initialize bf and ff arrays for (i = 0; i < nb; i++) bf[i] = 0; // 0 = free, 1 = allocated for (i = 0; i < nf; i++) ff[i] = -1; // -1 = no block assigned // Worst Fit allocation for (i = 0; i < nf; i++) { int worst_fit_block_idx; // ✅ Declare at start of block highest = -1; // Reset highest for each file worst_fit_block_idx = -1; // No block chosen yet for (j = 0; j < nb; j++) { if (bf[j] != 1) { // Block is free temp = b[j] - f[i]; if (temp >= 0) { // Block can accommodate file if (temp > highest) { highest = temp; worst_fit_block_idx = j; } } } } if (worst_fit_block_idx != -1) { // A suitable block was found ff[i] = worst_fit_block_idx; // Assign block index to file frag[i] = highest; // Fragmentation bf[worst_fit_block_idx] = 1; // Mark block as allocated } else { frag[i] = -1; // File could not be allocated } } printf("\nFile_no:\tFile_size:\tBlock_no: \tBlock_size:\tFragment"); for (i = 0; i < nf; i++) { if (ff[i] != -1)
{ printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i + 1, f[i], ff[i] + 1, b[ff[i]], frag[i]); } else
{ printf("\n%d\t\t%d\t\tNot Allocated\t--\t\t--", i + 1, f[i]); } } printf("\n"); return 0; }

Expected output:

EXPERIMENT-7

Develop a C program to simulate page replacement algorithms:

a) FIFO

b) LRU

a)FIFO

#include<stdio.h> int fr[3]; void display() { int i; printf("\n"); for(i=0;i<3;i++) printf("%d\t",fr[i]); } int main() { int i,j,n,page[50]; int flag1=0,flag2=0,pf=0,frsize=3,top=0; printf("Enter number of pages (max 50): "); scanf("%d",&n); printf("Enter %d page numbers: ",n); for(i=0;i<n;i++) scanf("%d",&page[i]); for(i=0;i<3;i++) { fr[i]=-1; } for(j=0;j<n;j++) { flag1=0; flag2=0; for(i=0;i<3;i++) { if(fr[i]==page[j]) { flag1=1; flag2=1; break; } } if(flag1==0) { for(i=0;i<frsize;i++) { if(fr[i]==-1) { fr[i]=page[j]; flag2=1; pf++; // ← minimal bug fix break; } } } if(flag2==0) { fr[top]=page[j]; top++; pf++; if(top>=frsize) top=0; } display(); } printf("\nNumber of page faults : %d ",pf); return 0; }

Expected Output:

b) LRU

Code:

#include <stdio.h> #define FRAMES 3 #define PAGES 12 int fr[FRAMES]; void display() { int i; printf("\n"); for (i = 0; i < FRAMES; i++) printf("%d\t", fr[i]); } int main() { int page[PAGES] = {2,3,2,1,5,2,4,5,3,2,5,2}; int last_used[FRAMES]; // store "time" when each frame was last used int time = 0; int pf = 0; int i, j; // initialize frames as empty for (i = 0; i < FRAMES; i++) { fr[i] = -1; last_used[i] = -1; } for (j = 0; j < PAGES; j++) { int current = page[j]; int hit = 0; time++; // 1. Check if page is already in a frame (HIT) for (i = 0; i < FRAMES; i++) { if (fr[i] == current) { hit = 1; last_used[i] = time; // update last used time break; } } if (!hit) { // 2. Check for an empty frame first int placed = 0; for (i = 0; i < FRAMES; i++) { if (fr[i] == -1) { fr[i] = current; last_used[i] = time; pf++; placed = 1; break; } } // 3. If no empty frame, replace LRU page if (!placed) { int lru_index = 0; int min_time = last_used[0]; for (i = 1; i < FRAMES; i++) { if (last_used[i] < min_time) { min_time = last_used[i]; lru_index = i; } } fr[lru_index] = current; last_used[lru_index] = time; pf++; } } display(); } printf("\nNumber of page faults (LRU): %d\n", pf); return 0; }

Expected Output:

EXPERIMENT-8

Simulate following File Organization Techniques

A) Single level directory.

B) Two level directory.

A) Single level directory

Code;

```
#include<stdio.h>

// Structure to represent a directory
struct Directory {
   char name[20];
   int fileCount;
   char files[20][20];
};

int main()
{
   struct Directory directories[20];
   int master;
   int i, j;
```

```
    printf("Enter number of directories: ");
    scanf("%d", &master);

    // Input directory information
    for(i = 0; i < master; i++) {
        printf("\nDirectory %d:\n", i + 1);

        printf("  Enter directory name: ");
        scanf("%s", directories[i].name);

        printf("  Enter number of files: ");
        scanf("%d", &directories[i].fileCount);

        printf("  Enter file names:\n");
        for(j = 0; j < directories[i].fileCount; j++) {
            printf("    File %d: ", j + 1);
            scanf("%s", directories[i].files[j]);
        }
    }

    // Display directory structure
    printf("\n\n");
    printf("=================================================\n");
    printf(" Directory\t\tSize\tFilenames\n");
    printf("=================================================\n");

    for(i = 0; i < master; i++)
    {
        printf("%-20s\t%2d\t", directories[i].name, directories[i].fileCount);

        for(j = 0; j < directories[i].fileCount; j++) {
            if(j == 0)
                printf("%s\n", directories[i].files[j]);
            else
                printf("\t\t\t\t%s\n", directories[i].files[j]);
        }
        printf("\n");
    }
    printf("=================================================\n");

    return 0;
}
```

Expected output:

b) Two level directory

```
#include<stdio.h>
#include<conio.h>
struct st
{
char dname[10];
char sdname[10][10];
char fname[10][10][10];
int ds,sds[10];
}dir[10];
void main()
{
int i,j,k,n;
printf("enter number of directories:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("enter directory %d names:",i+1);
scanf("%s",&dir[i].dname);
printf("enter size of directories:");
scanf("%d",&dir[i].ds);
for(j=0;j<dir[i].ds;j++)
{
printf("enter subdirectory name and size:");
scanf("%s",&dir[i].sdname[j]);
scanf("%d",&dir[i].sds[j]);
for(k=0;k<dir[i].sds[j];k++)
{
printf("enter file name:");
scanf("%s",&dir[i].fname[j][k]);
}}}
printf("\ndirname\t\tsize\tsubdirname\tsize\tfiles");
printf("\n************************************************\n");
for(i=0;i<n;i++)
{
printf("%s\t\t%d",dir[i].dname,dir[i].ds);
for(j=0;j<dir[i].ds;j++)
{
printf("\t%s\t\t%d\t",dir[i].sdname[j],dir[i].sds[j]);
for(k=0;k<dir[i].sds[j];k++)
printf("%s\t",dir[i].fname[j][k]);
printf("\n\t\t");
}
printf("\n");
}
```

```
}
```

Expected Output;

EXPERIMENT-9

Develop a C Program to simulate the Linked file allocation strategies

Code;

#include<stdio.h> #include<stdlib.h> // Added for exit() function void main() { int f[50],p,i,j,k,a,st,len,n,c; for(i=0;i<50;i++) f[i]=0; printf("Enter how many blocks that are already allocated: "); scanf("%d",&p); printf("\nEnter the blocks no.s that are already allocated: "); for(i=0;i<p;i++) { scanf("%d",&a); f[a]=1; } X: printf("\nEnter the starting index block & length: "); scanf("%d%d",&st,&len); k=len; for(j=st;j<(k+st);j++) { if(f[j]==0) { f[j]=1; printf("\n%d->%d",j,f[j]); } else { printf("\n%d->file is already allocated",j); k++; } } printf("\nIf u want to enter one more file? (yes-1/no-0): "); scanf("%d",&c); if(c==1) goto X; else exit(0); // Changed from exit() to exit(0) }

Expected output:

EXPERIMENT-10

Develop a C program to simulate SCAN disk scheduling algorithm.

Code;

#include <stdio.h> #include <stdlib.h> #include <string.h> #define size 10 #define disk_size 200 int comp(const void * l, const void * n) { return (*(int*)l - *(int*)n); } void SCAN(int arr[], int head, char* dn){ int seek_num = 0; int dt, cur_track; int leftside[size + 1] = {0}; // Initialize array int rightside[size + 1] = {0}; // Initialize array int seek_seq[size + 2]; // Correct size int m_scan = 0, s_scan = 0; if (strcmp(dn, "leftside") == 0) leftside[m_scan++] = 0; else if (strcmp(dn, "rightside") == 0) rightside[s_scan++] = disk_size - 1; for (int p_s = 0; p_s < size; p_s++) { if (arr[p_s] < head) leftside[m_scan++] = arr[p_s]; if (arr[p_s] > head) rightside[s_scan++] = arr[p_s]; } qsort(leftside, m_scan, sizeof(int), comp); qsort(rightside, s_scan, sizeof(int), comp); int go = 2; int ind = 0; while (go--) { if (strcmp(dn, "leftside") == 0) { for (int p_s = m_scan - 1; p_s >= 0; p_s--) { cur_track =

```
leftside[p_s]; seek_seq[ind++] = cur_track; dt = abs(cur_track - head); seek_num += dt; head = cur_track; } dn = "rightside"; } else if (strcmp(dn, "rightside") == 0) { for (int p_s = 0; p_s < s_scan; p_s++) { cur_track = rightside[p_s]; seek_seq[ind++] = cur_track; dt = abs(cur_track - head); seek_num += dt; head = cur_track; } dn = "leftside"; } } printf("Num of seek process = %d\n", seek_num); printf("Sequence is:\n"); for (int p_s = 0; p_s < ind; p_s++) { printf("%d\n", seek_seq[p_s]); } } int main(){ int arr[size] = { 126, 90, 14, 50, 25, 42, 51, 78, 102, 100 }; int head = 42; char dn[] = "leftside"; SCAN(arr, head, dn); return 0; }
```

Expected output: