Atma Ram Sanatan Dharma College

University of Delhi

Operating System

Practical File

Submitted By:

Jyotiswaroop Srivastav
College Roll No. 21/18023
Semester III
BSc. (Hons) Computer Science

Submitted To:

Ms. Parul Jain

5. Write a program to copy files using system calls.

```c
/**
 * Write a program to copy a source file into the target file
and
 * display the target file using system calls.
 */

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

int main(int argc, char **argv)
{
    if (argc < 3)
    {
        fprintf(stderr, "Usage: ./main <src> <dest>\n");
        return -1;
    }

    char buf;
    ssize_t bytes;
    int fdSrc, fdDest;
    mode_t wrMode = 0777;

    if ((fdSrc = open(argv[1], O_RDONLY)) < 0)
    {
        fprintf(stderr, "Could not read %s\n", argv[1]);
        return 2;
    }

    if ((fdDest = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC,
wrMode)) < 0)
    {
        fprintf(stderr, "Could not write to %s\n", argv[2]);
        return 2;
```

```
    }

    while ((bytes = read(fdSrc, &buf, 1)) > 0)
        write(fdDest, &buf, 1);

    if (bytes < 0)
    {
        fprintf(stderr, "Could not read contents of %s\n",
argv[1]);
        return 2;
    }

    if (bytes == 0)
        write(fdDest, "\n", 1);

    printf("Copied contents of %s to %s\n", argv[1], argv[2]);

    close(fdSrc);
    close(fdDest);

    if ((fdDest = open(argv[2], O_RDONLY)) < 0)
    {
        fprintf(stderr, "Could not read %s\n", argv[2]);
        return 2;
    }

    while ((bytes = read(fdSrc, &buf, 1)) > 0)
        printf("%c", buf);

    close(fdDest);

    return 0;
}
```

```
$ gcc -o main main.c
$ ./main
Usage: ./main <src> <dest>
$ ./main src.txt dest.txt
Copied contents of src.txt to dest.txt
This is a text file.
```

11. Write a program to implement SRTF scheduling algorithm.

```c
/**
 * Write a program to implement SRTF scheduling algorithm.
 */

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

struct process {
  int pid;
  double burstTime;
  double arrivalTime;
  double waitingTime;
  double turnAroundTime;
};

void sortByArrivalTime(struct process *processes, int
processCount) {
  struct process temp;
  int i, j, n = processCount;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      if (processes[i].arrivalTime < processes[j].arrivalTime)
{
```

```
        temp = processes[j];
        processes[j] = processes[i];
        processes[i] = temp;
      }
}

void sortForSJF(struct process *processes, int processCount) {
  struct process temp;
  double min, startTime = 0.0;
  int i, j, k = 1, n = processCount;
  for (j = 0; j < n; j++) {
    startTime += processes[j].burstTime;
    min = processes[k].burstTime;
    for (i = k; i < n; i++)
      if (startTime >= processes[i].arrivalTime &&
          processes[i].burstTime < min) {
        temp = processes[k];
        processes[k] = processes[i];
        processes[i] = temp;
      }
    k++;
  }
}

void sortByPID(struct process *processes, int processCount) {
  struct process temp;
  int i, j, n = processCount;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      if (processes[i].pid < processes[j].pid) {
        temp = processes[j];
        processes[j] = processes[i];
        processes[i] = temp;
      }
}

void computeWaitingTime(struct process *processes, int
processCount) {
```

```
  double startTime = 0.0;
  processes[0].waitingTime = 0;
  for (int i = 1; i < processCount; i++) {
    startTime += processes[i - 1].burstTime;
    processes[i].waitingTime = startTime -
processes[i].arrivalTime;
  }
}

void computeTurnAroundTime(struct process *processes, int
processCount) {
  for (int i = 0; i < processCount; i++)
    processes[i].turnAroundTime =
        processes[i].burstTime + processes[i].waitingTime;
}

void printAverageTimes(struct process *processes, int
processCount,
                       char *unit) {
  double end;
  int smallest, count = 0;
  double totalWaitingTime = 0.0;
  double totalTurnAroundTime = 0.0;
  struct process temp[processCount + 1];

  //   sortByArrivalTime(processes, processCount);
  //   sortForSJF(processes, processCount);
  //   computeWaitingTime(processes, processCount);
  //   computeTurnAroundTime(processes, processCount);
  //   sortByPID(processes, processCount);

  for (int i = 0; i < processCount; i++)
    temp[i] = processes[i];

  smallest = processCount + 1;
  temp[smallest].burstTime = 999;
  for (double time = 0; count != processCount; time++) {
    printf("%lf %d", time, count);
```

```
    for (int i = 0; i < processCount; i++) {
      if (processes[i].arrivalTime <= time &&
          processes[i].burstTime < temp[smallest].burstTime &&
          processes[i].burstTime > 0) {
        smallest = i;
      }
    }
    temp[smallest].burstTime--;
    if (temp[smallest].burstTime == 0) {
      count++;
      end = time + 1;
      processes[count].waitingTime +=
          end - processes[smallest].arrivalTime -
temp[smallest].burstTime;
      processes[count].turnAroundTime += end -
processes[smallest].arrivalTime;
    }
  }

  printf(
      "Process ID\tBurst Time\tArrival Time\tWaiting
Time\tTurn-Around Time\n");
  printf("-------------------------------------------------
---");
  printf("-------------------------\n");
  for (int i = 0; i < processCount; i++) {
    totalWaitingTime += processes[i].waitingTime;
    totalTurnAroundTime += processes[i].turnAroundTime;
    printf("%d\t\t%.2lf%s\t\t%.2lf%s\t\t%.2lf%s\t\t%.2lf%s\n",
processes[i].pid,
           processes[i].burstTime, unit,
processes[i].arrivalTime, unit,
           processes[i].waitingTime, unit,
processes[i].turnAroundTime, unit);
  }
  printf("\nAverage Waiting Time = %.2lf%s", totalWaitingTime /
processCount,
         unit);
```

```c
  printf("\nAverage Turn-Around time = %.2lf%s\n",
         totalTurnAroundTime / processCount, unit);
  return;
}

int main(void) {
  int processCount;
  char unit[4] = {'\0'};

  printf("Enter Time Unit: ");
  fgets(unit, 3, stdin);

  printf("Enter Number of Processes: ");
  scanf("%i", &processCount);

  struct process processes[processCount];

  for (int i = 0; i < processCount; i++) {
    processes[i].pid = i + 1;
    printf("Burst Time for Process %i: ", i + 1);
    scanf("%lf", &processes[i].burstTime);
    printf("Arrival Time for Process %i: ", i + 1);
    scanf("%lf", &processes[i].arrivalTime);
  }

  printf("\n");

  printAverageTimes(processes, processCount, unit);

  return 0;
}
```

```
Enter Time Unit: ms
Enter Number of Processes: 3
Burst Time for Process 1: 8
Arrival Time for Process 1: 0
Burst Time for Process 2: 4
Arrival Time for Process 2: 0.4
Burst Time for Process 3: 1
Arrival Time for Process 3: 1

Process ID      Burst Time      Arrival Time    Waiting Time    Turn-Around Time
-------------------------------------------------------------------------
1               8.00ms          0.00ms          0.00ms          8.00ms
2               4.00ms          0.40ms          8.60ms          12.60ms
3               1.00ms          1.00ms          7.00ms          8.00ms

Average Waiting Time = 5.20ms
Average Turn-Around time = 9.53ms
```

12. Write a program to calculate sum of n numbers using thread library.

```cpp
/**
 * Write a program to calculate sum of n numbers using thread
library.
 */

#include <cstdlib>
#include <iostream>
#include <pthread.h>

using namespace std;

long long sum;

void *runner(void *number);

int main(int argc, char **argv)
{
    if (argc != 2)
    {
        cerr << "Usage: ./main <upper>" << endl;
        exit(1);
```

```
    }

    if (atoi(argv[1]) < 0)
    {
        cerr << "Argument must be non-negative." << endl;
        exit(1);
    }

    pthread_t tid;
    pthread_attr_t attr;

    pthread_attr_init(&attr);
    pthread_create(&tid, &attr, runner, (void *)argv[1]);
    pthread_join(tid, NULL);

    cout << "Sum from 1 to " << atoi(argv[1])
         << " is " << sum << endl;

    return 0;
}

void *runner(void *upper)
{
    int num = atoi((const char *)(upper));
    for (int i = 1; i <= num; i++)
        sum += i;
    pthread_exit(0);
    return nullptr;
}
```

13. Write a program to implement first-fit, best-fit and worst-fit allocation strategies.

Best-Fit:

```cpp
/**
 * Write a program to implement first-fit, best-fit and
 * worst-fit allocation strategies.
 */

#include <cstring>
#include <iostream>
#define MAX_SIZE 100

using namespace std;

void bestFit(int blockSize[], int m,
             int processSize[], int n)
{
  int allocation[n];

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

  for (int i = 0; i < n; i++)
  {
    int bestIdx = -1;
    for (int j = 0; j < m; j++)
    {
      if (blockSize[j] >= processSize[i])
      {
        if (bestIdx == -1)
          bestIdx = j;
        else if (blockSize[bestIdx] > blockSize[j])
          bestIdx = j;
      }
    }

    if (bestIdx != -1)
    {
      allocation[i] = bestIdx;
      blockSize[bestIdx] -= processSize[i];
    }
```

```
  }

  cout << "\nBest-Fit Allocation Strategy\n";
  cout << "=========================================\n";
  cout << "Process No.\tProcess Size\tBlock No.\n";
  cout << "=========================================\n";
  for (int i = 0; i < n; i++)
  {
    cout << "   " << i + 1 << "\t\t" << processSize[i] <<
"\t\t";
    if (allocation[i] != -1)
      cout << allocation[i] + 1;
    else
      cout << "Not Allocated";
    cout << endl;
  }
}

int main()
{
  int holes, processes;
  int holeSizes[MAX_SIZE], processSizes[MAX_SIZE];

  cout << "Enter Number of Holes: ";
  cin >> holes;
  cout << "Enter Number of Processes: ";
  cin >> processes;

  for (int i = 0; i < holes; i++)
  {
    cout << "Enter Size of Hole " << (i + 1) << ": ";
    cin >> holeSizes[i];
  }

  for (int i = 0; i < processes; i++)
  {
    cout << "Enter Size of Process " << (i + 1) << ": ";
    cin >> processSizes[i];
```

```
  }

  bestFit(holeSizes, holes, processSizes, processes);

  return 0;
}
```

First-Fit:

```
/**
 * Write a program to implement first-fit, best-fit and
 * worst-fit allocation strategies.
 */

#include <cstring>
#include <iostream>
#define MAX_SIZE 100

using namespace std;

void firstFit(int blockSize[], int m,
              int processSize[], int n)
{
    int allocation[n];

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

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            if (blockSize[j] >= processSize[i])
            {
                allocation[i] = j;
                blockSize[j] -= processSize[i];
```

```
                break;
            }
        }
    }

    cout << "\nFirst-Fit Allocation Strategy\n";
    cout << "=======================================\n";
    cout << "\nProcess No.\tProcess Size\tBlock No.\n";
    cout << "=======================================\n";
    for (int i = 0; i < n; i++)
    {
        cout << " " << i + 1 << "\t\t"
             << processSize[i] << "\t\t";
        if (allocation[i] != -1)
            cout << allocation[i] + 1;
        else
            cout << "Not Allocated";
        cout << endl;
    }
}

int main()
{
    int holes, processes;
    int holeSizes[MAX_SIZE], processSizes[MAX_SIZE];

    cout << "Enter Number of Holes: ";
    cin >> holes;
    cout << "Enter Number of Processes: ";
    cin >> processes;

    for (int i = 0; i < holes; i++)
    {
        cout << "Enter Size of Hole " << (i + 1) << ": ";
        cin >> holeSizes[i];
    }

    for (int i = 0; i < processes; i++)
```

```
    {
        cout << "Enter Size of Process " << (i + 1) << ": ";
        cin >> processSizes[i];
    }

    firstFit(holeSizes, holes, processSizes, processes);

    return 0;
}
```

Worst-Fit:

```
/**
 * Write a program to implement first-fit, best-fit and
 * worst-fit allocation strategies.
 */

#include <cstring>
#include <iostream>
#define MAX_SIZE 100

using namespace std;

void worstFit(int blockSize[], int m,
              int processSize[], int n)
{
  int allocation[n];

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

  for (int i = 0; i < n; i++)
  {
    int wstIdx = -1;
    for (int j = 0; j < m; j++)
    {
```

```
      if (blockSize[j] >= processSize[i])
      {
        if (wstIdx == -1)
          wstIdx = j;
        else if (blockSize[wstIdx] < blockSize[j])
          wstIdx = j;
      }
    }

    if (wstIdx != -1)
    {
      allocation[i] = wstIdx;
      blockSize[wstIdx] -= processSize[i];
    }
  }

  cout << "\nWorst-Fit Allocation Strategy\n";
  cout << "=========================================\n";
  cout << "Process No.\tProcess Size\tBlock No.\n";
  cout << "=========================================\n";
  for (int i = 0; i < n; i++)
  {
    cout << "   " << i + 1 << "\t\t" << processSize[i] <<
"\t\t";
    if (allocation[i] != -1)
      cout << allocation[i] + 1;
    else
      cout << "Not Allocated";
    cout << endl;
  }
}

int main()
{
  int holes, processes;
  int holeSizes[MAX_SIZE], processSizes[MAX_SIZE];

  cout << "Enter Number of Holes: ";
```

```
  cin >> holes;
  cout << "Enter Number of Processes: ";
  cin >> processes;

  for (int i = 0; i < holes; i++)
  {
    cout << "Enter Size of Hole " << (i + 1) << ": ";
    cin >> holeSizes[i];
  }

  for (int i = 0; i < processes; i++)
  {
    cout << "Enter Size of Process " << (i + 1) << ": ";
    cin >> processSizes[i];
  }

  worstFit(holeSizes, holes, processSizes, processes);

  return 0;
}
```