OS Lab07

Author: Dipankar Das

Date: 7-4-2022

Roll: 20051554

Github Link

Headers

timer.h

```
#ifndef STDIO_H
#include <stdio.h>
#endif

size_t CLK_CYCLE;
```

proc.h

```
#ifndef STDLIB_H
#include <stdlib.h>
#endif

enum state {
  RUNNING,
  RUNNABLE,
  // WAITING,
  TERMINATED,
  EMBRYO
};


struct proc {
  int pid;
  int arrTime;
  int burstTime;
  enum state currState;
  int initStartTime;
  int finalEndTime;
};

struct proc* Rqueue = NULL;
int *tempStoreBT = NULL;
```

Question 1

Write a program in C to implement FCFS CPU scheduling without arrival time. (o/p-response time, turnaround time, idle time, CPU utilization time, completion time, CPU Utilization)

Solution

```
#include "proc.h"
#include "timer.h"

#ifndef STDIO_H
#include <stdio.h>
#endif
#ifndef STDLIB_H
#include <stdlib.h>
#endif


int NoOfProcesses;

void enterData() {
  printf("Enter the PID, BurstTime for each proc\n");
  int id;
  int bt;
  for (int i = 0; i < NoOfProcesses; i++)
  {
    scanf("%d %d", &id, &bt);
    Rqueue[i].arrTime = 0;
    Rqueue[i].burstTime = bt;
    Rqueue[i].currState = EMBRYO;
    Rqueue[i].pid = id;
    Rqueue[i].initStartTime = Rqueue[i].finalEndTime = 0;
    tempStoreBT[i] = bt;
  }
}

void __PS() {
  printf("PID\tArr\tBurst\n");
  for (int i = 0; i < NoOfProcesses; i++)
    printf("%d\t%d\t%d\n", Rqueue[i].pid, Rqueue[i].arrTime, Rqueue[i].burstTime);
}

void sched() {
  for (int i = 0; i < NoOfProcesses; i++) {
    if (CLK_CYCLE >= Rqueue[i].arrTime
        && Rqueue[i].currState == EMBRYO) {

      int minArr = i;
      for (int j = 0; j < NoOfProcesses; j++) {

        if (Rqueue[j].currState == EMBRYO
            && CLK_CYCLE >= Rqueue[j].arrTime
```

```
            && Rqueue[j].arrTime < Rqueue[minArr].arrTime)

              minArr = j;
      }
      i = minArr;
      Rqueue[i].currState = RUNNABLE;
      Rqueue[i].initStartTime = CLK_CYCLE;
      return;
    }
  }
}

int isAllDone() {
  for (int i = 0; i < NoOfProcesses; i++) {
    if (Rqueue[i].currState != TERMINATED)
      return 0;
  }
  return 1;
}

void __CPU_SCHED(int idx) {
  Rqueue[idx].currState = RUNNING;
  int BT = Rqueue[idx].burstTime;
  while (BT > 0) {
    CLK_CYCLE++;
    BT--;
  }
  Rqueue[idx].burstTime = 0;
  if (BT == 0) {
    Rqueue[idx].finalEndTime = CLK_CYCLE;
  }
  // record the Complition time for a process
  Rqueue[idx].currState = TERMINATED;
  printf("COMPLETED!!\tpid: %d\tCLK: %ld\n", Rqueue[idx].pid, CLK_CYCLE);
}

// returns the index for that process to run
void proc() {
  while (1) {
    // check if all have done
    if (isAllDone() == 1)
      return;

    for (int i = 0; i < NoOfProcesses; i++) {
      if (Rqueue[i].currState == RUNNABLE) {
        // found the process
        __CPU_SCHED(i);
        // Rqueue[i].currState = TERMINATED;
        break;
      }
    }
    // when a process gets completed the scheduler is called
    sched();
  }
```

```
}

void ReportDis() {
  int totalBT = 0;
  for (int i = 0; i < NoOfProcesses; i++)
  {
    int TT = Rqueue[i].finalEndTime - Rqueue[i].arrTime;
    int RT = Rqueue[i].initStartTime - Rqueue[i].arrTime;
    int WT = TT - tempStoreBT[i];
    printf("Process\tPID: %d\tBT: %d\tAT: %d\tTT: %d\tWT: %d\tRT: %d\n",
     Rqueue[i].pid, tempStoreBT[i], Rqueue[i].arrTime, TT, WT, RT);
    totalBT += tempStoreBT[i];
  }
  printf("TOTAL TIME: %ld\n", CLK_CYCLE);
  size_t idleTime = CLK_CYCLE - totalBT;
  printf("TOTAL IDLE Time: %ld\n", idleTime);
  printf("CPU UTIT: %d\tCPU UTIL PER: %f\n", totalBT, (float)(totalBT) /
(CLK_CYCLE));
}

int main() {
  CLK_CYCLE = 0;
  printf("Enter number of processes");
  scanf("%d", &NoOfProcesses);
  Rqueue = (struct proc *)malloc(sizeof(struct proc) * NoOfProcesses);
  tempStoreBT = (int *)malloc(sizeof(int) * NoOfProcesses);
  enterData();
  __PS();
  // initial scheduler is called so as to make the process as runnable
  sched();
  proc();

  // all have done display Report
  ReportDis();

  free(Rqueue);
  free(tempStoreBT);
  return 0;
}
```

Output

```
[Lab08] git:(master) ))) make q1 && make run
gcc -Wall Q1.c
./a.out
Enter number of processes3
Enter the PID, BurstTime for each proc
1 10
2 5
3 3
PID     Arr     Burst
1       0       10
2       0       5
3       0       3
COMPLETED!!     pid: 1  CLK: 10
COMPLETED!!     pid: 2  CLK: 15
COMPLETED!!     pid: 3  CLK: 18
Process PID: 1  BT: 10  AT: 0   TT: 10  WT: 0   RT: 0
Process PID: 2  BT: 5   AT: 0   TT: 15  WT: 10  RT: 10
Process PID: 3  BT: 3   AT: 0   TT: 18  WT: 15  RT: 15
TOTAL TIME: 18
TOTAL IDLE Time: 0
CPU UTIT: 18    CPU UTIL PER: 1.000000
```

Question 2

Write a program in C to implement FCFS CPU scheduling with arrival time.(o/p-response time, turnaround time , idle time, CPU utilization time, completion time and CPU Utilization)

Solution

```c
#include "proc.h"
#include "timer.h"

#ifndef STDIO_H
#include <stdio.h>
#endif
#ifndef STDLIB_H
#include <stdlib.h>
#endif


int NoOfProcesses;

void enterData() {
  printf("Enter the PID, ArrivialTime & BurstTime for each proc\n");
  int id, bt, at;
  for (int i = 0; i < NoOfProcesses; i++)
  {
```

```
    scanf("%d %d %d", &id, &at, &bt);
    Rqueue[i].arrTime = at;
    Rqueue[i].burstTime = bt;
    Rqueue[i].currState = EMBRYO;
    Rqueue[i].pid = id;
    Rqueue[i].initStartTime = Rqueue[i].finalEndTime = 0;
    tempStoreBT[i] = bt;
  }
}

void __PS() {
  printf("PID\tArr\tBurst\n");
  for (int i = 0; i < NoOfProcesses; i++)
    printf("%d\t%d\t%d\n", Rqueue[i].pid, Rqueue[i].arrTime, Rqueue[i].burstTime);
}

void sched() {
  for (int i = 0; i < NoOfProcesses; i++) {
    if (CLK_CYCLE >= Rqueue[i].arrTime
        && Rqueue[i].currState == EMBRYO) {

      int minArr = i;
      for (int j = 0; j < NoOfProcesses; j++) {

        if (Rqueue[j].currState == EMBRYO
            && CLK_CYCLE >= Rqueue[j].arrTime
            && Rqueue[j].arrTime < Rqueue[minArr].arrTime)

              minArr = j;
      }
      i = minArr;
      Rqueue[i].currState = RUNNABLE;
      Rqueue[i].initStartTime = CLK_CYCLE;
      return;
    }
  }
}

int isAllDone() {
  for (int i = 0; i < NoOfProcesses; i++) {
    if (Rqueue[i].currState != TERMINATED)
      return 0;
  }
  return 1;
}

void __CPU_SCHED(int idx) {
  Rqueue[idx].currState = RUNNING;
  int BT = Rqueue[idx].burstTime;
  while (BT > 0) {
    CLK_CYCLE++;
    BT--;
  }
  Rqueue[idx].burstTime = 0;
```

```
    if (BT == 0) {
      Rqueue[idx].finalEndTime = CLK_CYCLE;
      Rqueue[idx].currState = TERMINATED;
      printf("COMPLETED!!\tpid: %d\tCLK: %ld\n", Rqueue[idx].pid, CLK_CYCLE);
    }
    // record the Complition time for a process
  }

  // returns the index for that process to run
  void proc() {
    while (1) {
      // check if all have done
      if (isAllDone() == 1)
        return;

      int i;

      for (i = 0; i < NoOfProcesses; i++) {
        if (Rqueue[i].currState == RUNNABLE) {

          // find the minBT process
          int minBT = i;
          for (int j = 0; j < NoOfProcesses; j++) {
            if (Rqueue[j].currState == RUNNABLE &&
                  Rqueue[minBT].arrTime > Rqueue[j].arrTime)
                minBT = j;
          }

          i = minBT;
          __CPU_SCHED(i);

          break;
        }

      }
      if (i == NoOfProcesses) {
        // no process was found
        CLK_CYCLE++;
      }
      // when a process gets completed the scheduler is called
      sched();
    }
  }

  void ReportDis() {
    int totalBT = 0;
    for (int i = 0; i < NoOfProcesses; i++)
    {
      int TT = Rqueue[i].finalEndTime - Rqueue[i].arrTime;
      int RT = Rqueue[i].initStartTime - Rqueue[i].arrTime;
      int WT = TT - tempStoreBT[i];
      printf("Process\tPID: %d\tBT: %d\tAT: %d\tTT: %d\tWT: %d\tRT: %d\n",
             Rqueue[i].pid, tempStoreBT[i], Rqueue[i].arrTime, TT, WT, RT);
      totalBT += tempStoreBT[i];
```

```
    }
    printf("TOTAL TIME: %ld\n", CLK_CYCLE);
    size_t idleTime = CLK_CYCLE - totalBT;
    printf("TOTAL IDLE Time: %ld\n", idleTime);
    printf("CPU UTIT: %d\tCPU UTIL PER: %f\n", totalBT, (float)(totalBT) /
(CLK_CYCLE));
}

int main() {
    CLK_CYCLE = 0;
    printf("Enter number of processes");
    scanf("%d", &NoOfProcesses);
    Rqueue = (struct proc *)malloc(sizeof(struct proc) * NoOfProcesses);
    tempStoreBT = (int *)malloc(sizeof(int) * NoOfProcesses);
    enterData();
    __PS();
    // initial scheduler is called so as to make the process as runnable
    sched();
    proc();

    // all have done display Report
    ReportDis();

    free(Rqueue);
    free(tempStoreBT);
    return 0;
}
```

Output

```
[Lab08] git:(master) ››› make q2 && make run
gcc -Wall Q2.c
./a.out
Enter number of processes4
Enter the PID, ArrivialTime & BurstTime for each proc
1 6 4
2 0 2
3 4 2
4 5 6
PID     Arr     Burst
1       6       4
2       0       2
3       4       2
4       5       6
COMPLETED!!     pid: 2  CLK: 2
COMPLETED!!     pid: 3  CLK: 6
COMPLETED!!     pid: 4  CLK: 12
COMPLETED!!     pid: 1  CLK: 16
Process PID: 1  BT: 4   AT: 6   TT: 10  WT: 6   RT: 6
Process PID: 2  BT: 2   AT: 0   TT: 2   WT: 0   RT: 0
Process PID: 3  BT: 2   AT: 4   TT: 2   WT: 0   RT: 0
Process PID: 4  BT: 6   AT: 5   TT: 7   WT: 1   RT: 1
TOTAL TIME: 16
TOTAL IDLE Time: 2
CPU UTIT: 14    CPU UTIL PER: 0.875000
[Lab08] git:(master) ›››
```

Question 3

Write a program in C to implement SJF CPU scheduling (non-preemptive)(o/p-response time, turnaround time , waiting time, average waiting time.)

Solution

```c
#include "proc.h"
#include "timer.h"

#ifndef STDIO_H
#include <stdio.h>
#endif
#ifndef STDLIB_H
#include <stdlib.h>
#endif


int NoOfProcesses;

void enterData() {
```

```
  printf("Enter the PID, ArrivialTime & BurstTime for each proc\n");
  int id, bt, at;
  for (int i = 0; i < NoOfProcesses; i++)
  {
    scanf("%d %d %d", &id, &at, &bt);
    Rqueue[i].arrTime = at;
    Rqueue[i].burstTime = bt;
    Rqueue[i].currState = EMBRYO;
    Rqueue[i].pid = id;
    Rqueue[i].initStartTime = Rqueue[i].finalEndTime = 0;
    tempStoreBT[i] = bt;
  }
}

void __PS() {
  printf("PID\tArr\tBurst\n");
  for (int i = 0; i < NoOfProcesses; i++)
    printf("%d\t%d\t%d\n", Rqueue[i].pid, Rqueue[i].arrTime, Rqueue[i].burstTime);
}

void sched() {
  for (int i = 0; i < NoOfProcesses; i++) {
    if (CLK_CYCLE >= Rqueue[i].arrTime
        && Rqueue[i].currState == EMBRYO) {

      int minBT = i;
      for (int j = 0; j < NoOfProcesses; j++) {

        if (Rqueue[j].currState == EMBRYO
            && CLK_CYCLE >= Rqueue[j].arrTime
            && Rqueue[j].burstTime < Rqueue[minBT].burstTime)

              minBT = j;
      }
      i = minBT;
      Rqueue[i].currState = RUNNABLE;
      Rqueue[i].initStartTime = CLK_CYCLE;
      return;
    }
  }
}

int isAllDone() {
  for (int i = 0; i < NoOfProcesses; i++) {
    if (Rqueue[i].currState != TERMINATED)
      return 0;
  }
  return 1;
}

void __CPU_SCHED(int idx) {
  Rqueue[idx].currState = RUNNING;
  int BT = Rqueue[idx].burstTime;
  while (BT > 0) {
```

```
      CLK_CYCLE++;
      BT--;
    }
    Rqueue[idx].burstTime = 0;
    if (BT == 0) {
      Rqueue[idx].finalEndTime = CLK_CYCLE;
      Rqueue[idx].currState = TERMINATED;
      printf("COMPLETED!!\tpid: %d\tCLK: %ld\n", Rqueue[idx].pid, CLK_CYCLE);
    }
    // record the Complition time for a process
  }

  // returns the index for that process to run
  void proc() {
    while (1) {
      // check if all have done
      if (isAllDone() == 1)
        return;

      int i;

      for (i = 0; i < NoOfProcesses; i++) {
        if (Rqueue[i].currState == RUNNABLE) {

          // find the minBT process
          int minBT = i;
          for (int j = 0; j < NoOfProcesses; j++) {
            if (Rqueue[j].currState == RUNNABLE &&
                  Rqueue[minBT].burstTime > Rqueue[j].burstTime)
                minBT = j;
            if (Rqueue[j].currState == RUNNABLE &&
                  Rqueue[minBT].burstTime == Rqueue[j].burstTime &&
                  Rqueue[minBT].arrTime > Rqueue[j].arrTime)
                minBT = j;
          }

          i = minBT;
          __CPU_SCHED(i);

          break;
        }

      }
      if (i == NoOfProcesses) {
        // no process was found
        CLK_CYCLE++;
      }
      // when a process gets completed the scheduler is called
      sched();
    }
  }

  void ReportDis() {
    for (int i = 0; i < NoOfProcesses; i++)
```

```
    {
      int TT = Rqueue[i].finalEndTime - Rqueue[i].arrTime;
      int RT = Rqueue[i].initStartTime - Rqueue[i].arrTime;
      int WT = TT - tempStoreBT[i];
      printf("Process\tPID: %d\tBT: %d\tAT: %d\tTT: %d\tWT: %d\tRT: %d\n",
             Rqueue[i].pid, tempStoreBT[i], Rqueue[i].arrTime, TT, WT, RT);
    }
  }

  int main() {
    CLK_CYCLE = 0;
    printf("Enter number of processes");
    scanf("%d", &NoOfProcesses);
    Rqueue = (struct proc *)malloc(sizeof(struct proc) * NoOfProcesses);
    tempStoreBT = (int *)malloc(sizeof(int) * NoOfProcesses);
    enterData();
    __PS();
    // initial scheduler is called so as to make the process as runnable
    sched();
    proc();

    // all have done display Report
    ReportDis();

    free(Rqueue);
    free(tempStoreBT);
    return 0;
  }
```

Output

```
[Lab08] git:(master) ››› make q3 && make run
gcc -Wall Q3.c
./a.out
Enter number of processes4
Enter the PID, ArrivialTime & BurstTime for each proc
1 6 4
2 0 2
3 4 2
4 5 6
PID     Arr     Burst
1       6       4
2       0       2
3       4       2
4       5       6
COMPLETED!!     pid: 2  CLK: 2
COMPLETED!!     pid: 3  CLK: 6
COMPLETED!!     pid: 1  CLK: 10
COMPLETED!!     pid: 4  CLK: 16
Process PID: 1  BT: 4   AT: 6   TT: 4   WT: 0   RT: 0
Process PID: 2  BT: 2   AT: 0   TT: 2   WT: 0   RT: 0
Process PID: 3  BT: 2   AT: 4   TT: 2   WT: 0   RT: 0
Process PID: 4  BT: 6   AT: 5   TT: 11  WT: 5   RT: 5
[Lab08] git:(master) ›››
```

Question 4

Write a program in C to implement SJF CPU scheduling (preemptive)(o/p-response time, turnaround time , waiting time, average waiting time.)

Solution

```c
#include <stdbool.h>
#include "proc.h"
#include "timer.h"

#ifndef STDIO_H
#include <stdio.h>
#endif
#ifndef STDLIB_H
#include <stdlib.h>
#endif


int NoOfProcesses;

void enterData() {
  printf("Enter the PID, ArrivialTime & BurstTime for each proc\n");
```

```
  int id, bt, at;
  for (int i = 0; i < NoOfProcesses; i++)
  {
    scanf("%d %d %d", &id, &at, &bt);
    Rqueue[i].arrTime = at;
    Rqueue[i].burstTime = bt;
    Rqueue[i].currState = EMBRYO;
    Rqueue[i].pid = id;
    Rqueue[i].initStartTime = Rqueue[i].finalEndTime = 0;
    tempStoreBT[i] = bt;
  }
}

void __PS() {
  printf("PID\tArr\tBurst\n");
  for (int i = 0; i < NoOfProcesses; i++)
    printf("%d\t%d\t%d\n", Rqueue[i].pid, Rqueue[i].arrTime, Rqueue[i].burstTime);
}

void sched() {
  for (int i = 0; i < NoOfProcesses; i++) {
    if (CLK_CYCLE >= Rqueue[i].arrTime
        && Rqueue[i].currState == EMBRYO) {

      int minBT = i;
      for (int j = 0; j < NoOfProcesses; j++) {

        if (Rqueue[j].currState == EMBRYO
            && CLK_CYCLE >= Rqueue[j].arrTime
            && Rqueue[j].burstTime < Rqueue[minBT].burstTime)

              minBT = j;
      }
      i = minBT;
      Rqueue[i].currState = RUNNABLE;
      // it is not always gaurantee that once the program get its RUNNABLE it it
loaded on CPU
      // Rqueue[i].initStartTime = CLK_CYCLE;
      return;
    }
  }
}

int isAllDone() {
  for (int i = 0; i < NoOfProcesses; i++) {
    if (Rqueue[i].currState != TERMINATED)
      return 0;
  }
  return 1;
}

void __CPU_SCHED(int idx) {
  Rqueue[idx].currState = RUNNING;
  int BT = Rqueue[idx].burstTime;
```

```
  // if the process starts its exquition for the first time it saves it
  if (BT == tempStoreBT[idx]) {
    Rqueue[idx].initStartTime = CLK_CYCLE;
  }
  bool flag = true;
  while (BT > 0 && flag) {
    CLK_CYCLE++;
    BT--;
    flag = false; // ensuring that cpu runs for only one clk so that we can check
continuously
    // for the new arrivial process
  }
  Rqueue[idx].burstTime = BT;
  if (BT == 0) {
    Rqueue[idx].finalEndTime = CLK_CYCLE;
    Rqueue[idx].currState = TERMINATED;
    printf("COMPLETED!!\tpid: %d\tCLK: %ld\n", Rqueue[idx].pid, CLK_CYCLE);
    return;
  }
  Rqueue[idx].currState = RUNNABLE;
  // record the Complition time for a process
}

// returns the index for that process to run
void proc() {
  while (1) {
    // check if all have done
    if (isAllDone() == 1)
      return;

    int i;

    for (i = 0; i < NoOfProcesses; i++) {
      if (Rqueue[i].currState == RUNNABLE) {

        // find the minBT process
        int minBT = i;
        for (int j = 0; j < NoOfProcesses; j++) {
          if (Rqueue[j].currState == RUNNABLE &&
                Rqueue[minBT].burstTime > Rqueue[j].burstTime)
              minBT = j;
          if (Rqueue[j].currState == RUNNABLE &&
                Rqueue[minBT].burstTime == Rqueue[j].burstTime &&
                Rqueue[minBT].arrTime > Rqueue[j].arrTime)
              minBT = j;
        }

        i = minBT;
        __CPU_SCHED(i);

        break;
      }

    }
```

```
      if (i == NoOfProcesses) {
        // no process was found
        CLK_CYCLE++;
      }
      // when a process gets completed the scheduler is called
      sched();
    }
  }

  void ReportDis() {
    for (int i = 0; i < NoOfProcesses; i++)
    {
      int TT = Rqueue[i].finalEndTime - Rqueue[i].arrTime;
      int RT = Rqueue[i].initStartTime - Rqueue[i].arrTime;
      int WT = TT - tempStoreBT[i];
      printf("Process\tPID: %d\tBT: %d\tAT: %d\tTT: %d\tWT: %d\tRT: %d\n",
             Rqueue[i].pid, tempStoreBT[i], Rqueue[i].arrTime, TT, WT, RT);
    }
  }

  int main() {
    CLK_CYCLE = 0;
    printf("Enter number of processes");
    scanf("%d", &NoOfProcesses);
    Rqueue = (struct proc *)malloc(sizeof(struct proc) * NoOfProcesses);
    tempStoreBT = (int *)malloc(sizeof(int) * NoOfProcesses);
    enterData();
    __PS();
    // initial scheduler is called so as to make the process as runnable
    sched();
    proc();

    // all have done display Report
    ReportDis();

    free(Rqueue);
    free(tempStoreBT);
    return 0;
  }
```

Output

```
[Lab08] git:(master) ››› make q4 && ./a.out
gcc -Wall Q4.c
Enter number of processes5
Enter the PID, ArrivialTime & BurstTime for each proc
1 2 5
2 4 6
3 1 7
4 5 1
5 8 2
PID     Arr     Burst
1       2       5
2       4       6
3       1       7
4       5       1
5       8       2
COMPLETED!!     pid: 4  CLK: 6
COMPLETED!!     pid: 1  CLK: 8
COMPLETED!!     pid: 5  CLK: 10
COMPLETED!!     pid: 3  CLK: 16
COMPLETED!!     pid: 2  CLK: 22
Process PID: 1  BT: 5   AT: 2   TT: 6   WT: 1   RT: 0
Process PID: 2  BT: 6   AT: 4   TT: 18  WT: 12  RT: 12
Process PID: 3  BT: 7   AT: 1   TT: 15  WT: 8   RT: 0
Process PID: 4  BT: 1   AT: 5   TT: 1   WT: 0   RT: 0
Process PID: 5  BT: 2   AT: 8   TT: 2   WT: 0   RT: 0
[Lab08] git:(master) ›››
```