Documentation
for Operating Systems

/***
* **Programmer**: Andrew Ybarra
*
* **Course**: CSCI 4354.01
*
* **Date**: April 19, 2018
*
* **Assignment**: Program #4: CPU Scheduler
*
* **Environment**: C using XCODE and implementation through the UNIX server sapphire
*
* **Files Included**: cpuServer.c
* cpuClient.c
*
* **Purpose**: To write code that simulates a CPU scheduler using round robin, a CPU(Ready)
* queue and an I/O queue.
*
* **Input**: Number of clients, how many burst (3 or 5 for simplification), each IO and CPU burst.
*
* **Preconditions**: Must only open the commFIFO once. Must use round robin. Must use an array
* to hold burst for CPU and IO alternating and starting/ending with CPU.
*
* **Output**: Clock, what process is at head of Ready and IO. CPU utilization. Finishing times
* and when a process dequeues completely.
*
* **Postconditions**: N/A
*
* **Est/Actual Design Time:** **2 hours** / **1 hours**
*
* **Est/Actual Implementation Time:** **6 hours** / **8 hours**
*
* **Est/Actual Test Time:** **3 hour** / **4 hours**
*
* **Algorithm**:
* Ask user for number of clients
* Enter amount of burst
* Enter burst sizes
* open commFIFO
* open privateFIFO
* send structs to server
* for every client
* state information recieved
* close commFIFO

```
*       start clock
*       while both queues are not empty
*               if ready > 1
*               count++
*               if count %timequantum = 0 and count != burst size
*                       requeue to the ready
*               if  count = burst size
*                       dequeue from ready
*                       pointer++
*                       if pointer has reached last spot of array
*                       remove the process and send back info
*                       close privateFIFO
*                       else enque into the IO
*               if io >= 1
*                       count2++
*               if count2 = total burst
*               dequeue IO
*               pointer++
*               enqueue to Ready
*       destroy queues
*       print totalCPU utilization
*
**************************************************************************/
```

CLIENT

```
/*////////////////////////////////////////////////////
/
/       Andrew Ybarra
/       1081010
/       Programming Assignment 3
/       cpuClient.c
/
/       The purpose of this program is to take in
/       burst size of cpu and IO from the client
/       and send them to the server. The client
/       waits to recieve the completion time
/       and the CPU utilization and prints
```

```
/          both results to screen.
/
*////////////////////////////////////////////////

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


typedef struct values {
   char privateFIFO[14];
   int burst[5];
   int taround;
   int pointer;
   int max;
   int counter;
   float cpu;
} Input;


main (void)
{

   Input input;
```

```
int i = 0;
int fdIN; //to write to character server
int fdOUT; //to read from character server
int clientID = getpid();
input.pointer = 0;
input.counter = 0;

printf("How many CPU/IO burst are there in total(3 or 5): ");
scanf("%d", &input.max);

for(i = 0; i < input.max; i++)
{
   if( i%2 == 0)
      printf("Please enter the CPU-burst: " );
   else
      printf("Please enter the I/O-burst: " );
   scanf("%d", &input.burst[i]);
}


sprintf(input.privateFIFO, "FIFO_%d", clientID);

printf("---------------------------------\n");
printf("Private FIFO name: %s", input.privateFIFO);
printf("\n---------------------------------\n\n");

if((mkfifo(input.privateFIFO, 0666)<0 && errno != EEXIST))
{
```

```
    perror("Can't create private FIFO\n");
    exit(-1);
  }



  if((fdIN=open("commFIFO", O_WRONLY))<0)  //writting into fifo
    printf("cant open fifo to write");

  write(fdIN, &input, sizeof(input));



  if((fdOUT=open(input.privateFIFO, O_RDONLY))<0) //reading from fifo
    printf("cant open fifo to read");



  read(fdOUT, &input, sizeof(input));

  printf("\nCompletion Time: %d\nCPU Utilization: %.2f\n\n\nCLIENT DONE\n", input.taround,
input.cpu);



  unlink ("commFIFO");
  unlink (input.privateFIFO);

  close(fdIN);
  close(fdOUT);

}
```

SERVER----------

```
/*////////////////////////////////////////////////
 /
 /        Andrew Ybarra
 /        1081010
 /        Programming Assignment 3
 /        cpuServer.c
 /
 /        The purpose of this program is to take in
 /        multiple clients and store them into
 /        queues. Using round-robin we can dequeue
 /        each client based on their burst size.
 /        The program shows each step of the
 /        clock cycle including what process is at
 /        head and when they dequeue.
 /        Once they dequeue they either go to the IO
 /        queue or requeue depending on their burst
 /        size remaining.
 /
 *////////////////////////////////////////////////
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
```

```
typedef struct values {
   char privateFIFO[14];
   int burst[5];
   int taround;
   int pointer;
   int max;
   int counter;
   float cpu;
} Input;

typedef struct node{ /*Nodes stored in the linked list*/

   struct values elements;

   struct node *next;

} Node;

//Queue Definition

typedef struct queue{ /*A struct facilitates passing a queue as an argument*/

   Node *head;      /*Pointer to the first node holding the queue's data*/

   Node *tail;      /*Pointer to the last node holding the queue's data*/

   int sz;          /*Number of nodes in the queue*/

} Queue;
```

```
int size( Queue *Q ){

   return Q->sz;

}

int isEmpty( Queue *Q ){

   if( Q->sz == 0 ) return 1;

   return 0;

}

void enqueue( Queue *Q, struct values elem ){

   Node *v = (Node*)malloc(sizeof(Node));/*Allocate memory for the Node*/

   if( !v ){

      printf("ERROR: Insufficient memory\n");

      return;

   }

   v->elements = elem;
```

```
    v->next = NULL;

    if( isEmpty(Q) ) Q->head = v;

    else Q->tail->next = v;

    Q->tail = v;

    Q->sz++;

}

struct values dequeue( Queue *Q ){

    Node *oldHead;

    struct values temp;

    if( isEmpty(Q) ){

        printf("ERROR: Queue is empty\n");

        return temp;

    }

    oldHead = Q->head;

    temp = Q->head->elements; //72
```

```
    Q->head = Q->head->next;

    free(oldHead);

    Q->sz--;

    return temp;

}

struct values first( Queue *Q ){

    if( isEmpty(Q) ){

        printf("ERROR: Queue is empty\n");

        return Q->head->elements;

    }


    return Q->head->elements;

}

struct values printFirst( Queue *Q ){

    if( isEmpty(Q) ){
```

```
        printf("ERROR: Queue is empty\n");

        return Q->head->elements;

    }
    printf("\t\tReady: %s", Q->head->elements.privateFIFO);

    return Q->head->elements;

}

struct values printFirstIO( Queue *Q ){

    if( isEmpty(Q) ){

        printf("ERROR: Queue is empty\n");

        return Q->head->elements;

    }
    printf("\t\tI/O: %s", Q->head->elements.privateFIFO);

    return Q->head->elements;

}

void destroyQueue( Queue *Q ){
```

```
    while( !isEmpty(Q) ) dequeue(Q);


}


////////////////////////////////////////////////////////////////////////////////  MAIN  //////////////
////////////////////////////////////////////////////////////////////////////


main (void)


{
    int fdIN;
    int fdOUT;
    int clients;
    float clock = 0;
    int i = 0;
    int timeQuantum;
    int finalTurnAround = 0;


    Input input;
    Queue Ready;
    Queue IO;


    IO.head = NULL;
    IO.tail = NULL;
    IO.sz = 0;


    Ready.head = NULL;
    Ready.tail = NULL;
    Ready.sz = 0;
```

```
if ((mkfifo("commFIFO",0666)<0 && errno != EEXIST))        // creates common FIFO
{
   perror("Can't create Common FIFO\n");
   exit(-1);
}


printf("\nPlease enter the amount of clients: ");
scanf("%d", &clients);
printf("Please enter the size of Time Quantum: ");
scanf("%d", &timeQuantum);

printf("\nServer is ready...\n\n");

if((fdIN=open("commFIFO", O_RDONLY))<0) /*Opening commFIFO*/
   printf("Can't open common FIFO to read\n");

for(i = 0; i < clients; i++)       //fills for every client
{
   read(fdIN, &input, sizeof(input));
   printf("\t   Client %d\n", i + 1);
   printf("\n---------------------------------\n");
   printf("Private FIFO name: %s", input.privateFIFO);
   enqueue(&Ready, input);
   printf("\t\t\tSize of Ready queue: %d\n", Ready.sz);
   printf("---------------------------------\n\n");
}
```

```
close(fdIN);
unlink("commFIFO");

int count = 0;
int count2 = 0;
float utlize = 0;
float percentage = 0;

clock = 0;
printf("\n\nStarting...\n");
while( !isEmpty(&Ready) || !isEmpty(&IO))
{
      sleep(1);

      if (!isEmpty(&Ready))
      {
            if(first(&Ready).burst[first(&Ready).pointer] >= 1)          // will not finish in time
              {
                 printf("\nClock: %.0f", clock);
                 printFirst(&Ready);
                 count++;

                    if(count%timeQuantum == 0 && count != first(&Ready).burst[first(&Ready).pointer])
                    {
                          Input temp = dequeue(&Ready);       // creates a temp node
                          temp.burst[temp.pointer] = temp.burst[temp.pointer] - timeQuantum;
                          enqueue(&Ready, temp);              // puts it back in line
                          count = 0;
```

```
                }
                if(count == first(&Ready).burst[first(&Ready).pointer])
                {
                    Input input = dequeue(&Ready);       // creates a temp node
                    input.burst[input.pointer] = input.burst[input.pointer] - count;
                    input.pointer++;
                    if(input.pointer == input.max)
                    {
                       clients--;
                       percentage = (clock - utlize)/clock;
                       printf("\n\n\tProcess %s has finished.\n\tCompletion Time: %.0f\n\tCPU
Utilization: %.2f\n\tProcess Dequeued\n", input.privateFIFO, clock + input.counter, percentage);
                       printf("\tSize of Ready queue: %d\n", Ready.sz);
                       printf("\tSize of IO queue: %d\n\t\t\t\t\t", IO.sz);
                       input.taround = clock + input.counter;
                       input.cpu = percentage;
                       fdOUT = open(input.privateFIFO, O_WRONLY); //Open privFIFO
                       write(fdOUT, &input, sizeof(input)); //Write to privFIFO
                       count = 0;
                       if(clients == 0)
                       {
                          printf("\n\n----------\nTOTAL CPU UTILIZATION: %.2f\n----------
----------", percentage);
                          break;
                       }
                    }
                    else{
                    enqueue(&IO, input);
                    count = 0;
```

```
                    }
                count = 0;


            }
        }

}
else{
   utlize++;
   printf("\nClock: %.0f", clock);
   printf("\t\tReady:          ");
   }

if(!isEmpty(&IO))
{
      if(first(&IO).burst[first(&IO).pointer] >= 1)
      {
         printFirstIO(&IO);
         count2++;
      }
      if(count2 == first(&IO).burst[first(&IO).pointer])
      {
         Input temp = dequeue(&IO);
         count2++;
         temp.counter++;
         temp.pointer++;
         enqueue(&Ready, temp);
         count2 = 0;
      }
```

```
            }
             else
            {
             printf("\t\tI/O:");
             }

       clock++;



       //printf("\n\n\tClient containing %s has finished.\n\tCompletion/Turn Around Time: %d\n\tClient
Dequeued\n", input.privateFIFO, clock);
       //fdOUT = open(input.privateFIFO, O_WRONLY); //Open privFIFO
       //write(fdOUT, &input, sizeof(input)); //Write to privFIFO
       //printf("\tSize of Ready Queue %d\n", Ready.sz);




    }


    destroyQueue(&Ready);


    printf("\n\n\n");



}
```