Operating Systems [COMP2006]

OS Assignment 2018
Simple Data Sharing - Reader-Writers Problem

Eric Wojcik
Student Number: 19142124

Overview

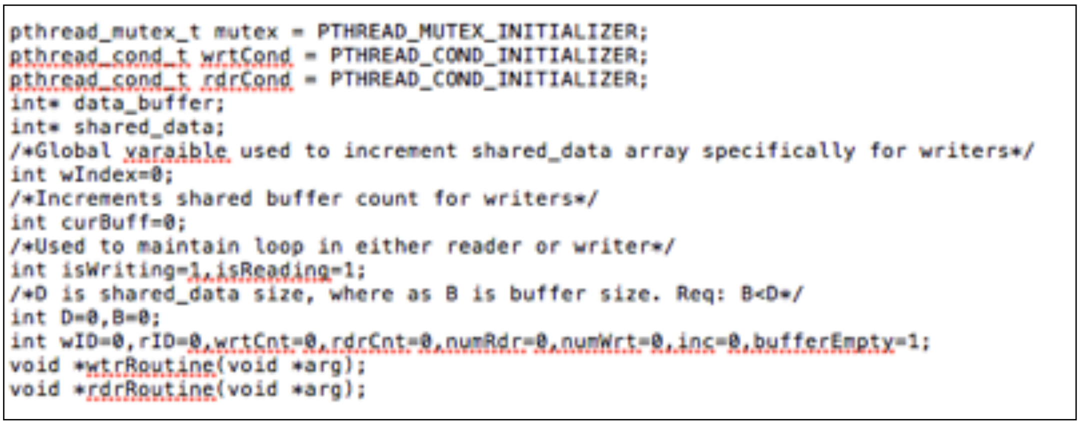
This report provides a brief discussion on how mutual exclusion is achieved in regards to a simple data sharing program. In particular, it will detail the use of mutual exclusion techniques in both process and thread implementations of the program. Further, it will explain the overall design of the code and the variables and data structures used.

1.0 Thread Implementation

Mutual exclusivity over shared resources in the multi-threaded variant of the program is detailed below.

The reader **(rdrRoutine())** first locks the mutex with *pthread_mutex_lock()* to obtain access to the shared variables and structures. It then checks if there are currently any writers writing or waiting to write. If so, the reader calls *pthread_cond_wait()* - with the reader condition variable *rdrCond* and *mutex* as arguments - which relinquishes control over the said lock and waits until the resources are available for use aka when the writer completes a write operation to the *data_buffer* and signals availability with *pthread_cond_signal()*. Once the writer is complete, and the shared resource is no longer being manipulated, the lock is obtained again as part of the *pthread_cond_wait()* function and *rdrCnt* is incremented. The *mutex* is then unlocked, allowing another reader to come in a read the resource concurrent to the first reader before being locked again so the *rdrCnt* can be decremented, indicating a reader has finished. If the final reader has left, *pthread_cond_signal()* is performed on the writer condition variable so that the writer is able to own mutex, before the mutex itself is unlocked by the reader.

For the writer **(wrtRoutine())**, a similar configuration is made. After the mutex lock is obtained, it instead checks if there are any queued writers or readers currently reading before calling *pthread_cond_wait()* on the writer condition variable *wrtCond*. After any writing is performed, and given there is more than 1 writer, *pthread_cond_signal()* is used to signal the next writer that it can leave the wait cycle. Otherwise, all the readers are woken up using *pthread_cond_broadcast()* on *rdrCond*.



```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t wrtCond = PTHREAD_COND_INITIALIZER;
pthread_cond_t rdrCond = PTHREAD_COND_INITIALIZER;
int* data_buffer;
int* shared_data;
/*Global varaible used to increment shared_data array specifically for writers*/
int wIndex=0;
/*Increments shared buffer count for writers*/
int curBuff=0;
/*Used to maintain loop in either reader or writer*/
int isWriting=1,isReading=1;
/*D is shared_data size, where as B is buffer size. Req: B<D*/
int D=0,B=0;
int wID=0,rID=0,wrtCnt=0,rdrCnt=0,numRdr=0,numWrt=0,inc=0,bufferEmpty=1;
void *wtrRoutine(void *arg);
void *rdrRoutine(void *arg);
```

Figure 1: Global variables used

Shared resources are global variables that are usable by both the reader and writer functions. Because of the concurrent nature of the program, said variables can be altered at the same time causing inconsistent results. The mutual exclusion lock is used to protect these variables from being altered at the same time by a) requiring a given thread obtain a mutex lock before accessing a global variable and b) blocking any other threads that try to acquire the mutex lock while it is in use. This behaviour therefore ensures that a race condition cannot occur.

2.0 Process Implementation

Mutual exclusivity over shared resources in the multi-process variant of the program is detailed below.

This implementation of the program utilised child processes to allocate duties to readers and writers [N readers or writers meant N child processes]. Overall, mutual exclusion is accomplished by using semaphores. This forces one child process to wait for another child process when they are accessing their critical section/shared resources. Achieved using *sem_wait()* and *sem_signal().*

In terms of the reader **(rdrRoutine()),** it would first use *sem_wait()* to acquire a lock on the *sems[0]/mutex* so it can access *readCount*, amongst other shared variables. *ReadCount* is incremented. It then check if the current reader is the first reader to access resource via *readCount*. If yes, then a *sem_wait()* is performed on *sems[1]/wrt* to prevent writing from occurring whilst the reader is accessing the resource. Immediately after, *sem_post()* is used on *sems[0]/mutex* to allow other readers to view the *data_buffer*. Buffer/array variables are incremented and reading is actually performed before *sem_wait()* is performed on *sems[0]/mutex* again. The *readCount* is decremented, and function checks if the current reader is the last reader. If true, *sem_unlock()* is used on *sems[1]/wrt* to allow the writer to perform operations on shared data. Finally mutex is also unlocked.

For the writer **(wrtRoutine()),** *sem_wait()* and *sem_post()* are only used once on *sems[1]/wrt* to allow the writer access to the shared resource *(data_buffer)* for modification.

As each process has its own instance of global variables, shared memory regions were created to simulate the effect of global variables and enable resources to be shared amongst reader and writer processes. The following POSIX functions were used to create, size, allocate and deallocate these shared regions:

- shm_open()
- ftruncate()
- mmap()
- shm_unlink()

- close()
- munmap()

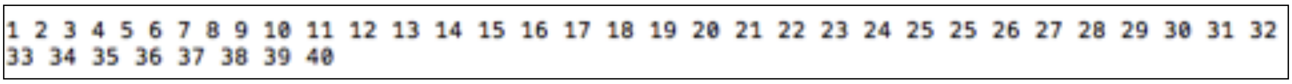
Semaphores [*mutex and wrt*] were created using *sem_init()*, and removed using *sem_close()* and *sem_destroy()*. Any zombie processes were handled with the *wait()* function, so that the parent process waits for all the child processes finish before terminating. This ensures no child processes remain.

3.0 Testing

3.1 Issues/Known errors

Both implementations of the the reader-writer solution do not work as intended. Although the cause of the issues have yet to be determined, it is most likely to do with synchronicity, or general design flaws. For the process implementation, an input with 2 readers and 2 writers produce almost desirable results (with the exception of the last element of the buffer cycle storing a invalid value), however even this result can be inconsistent. Besides this, there are no know faults with the program. All allocated memory has been deallocated accordingly. Despite this, there is a possibility of memory leaks occurring upon failure of shared memory operations (such as *mmap()* or *shm_open()*).

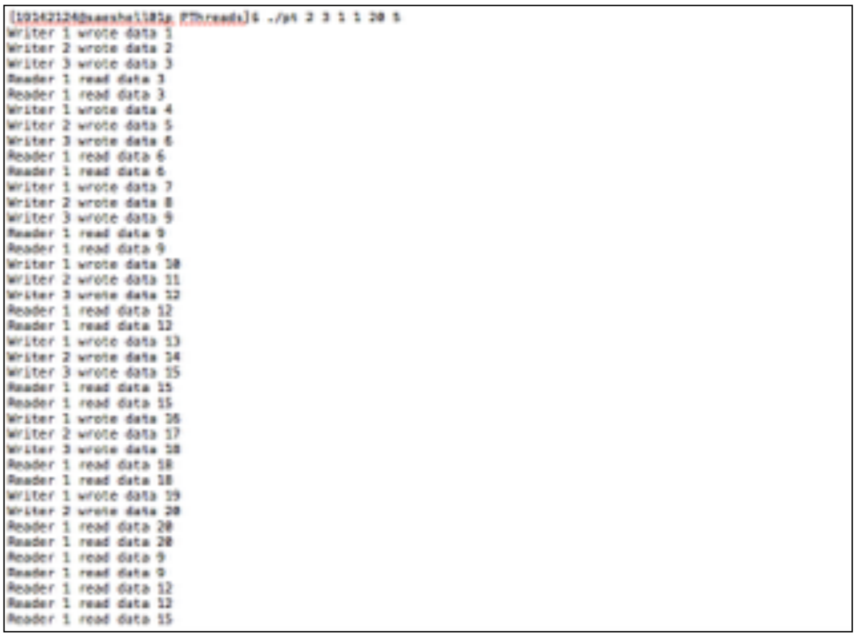
3.2 Input Data



```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 25 26 27 28 29 30 31 32
33 34 35 36 37 38 39 40
```

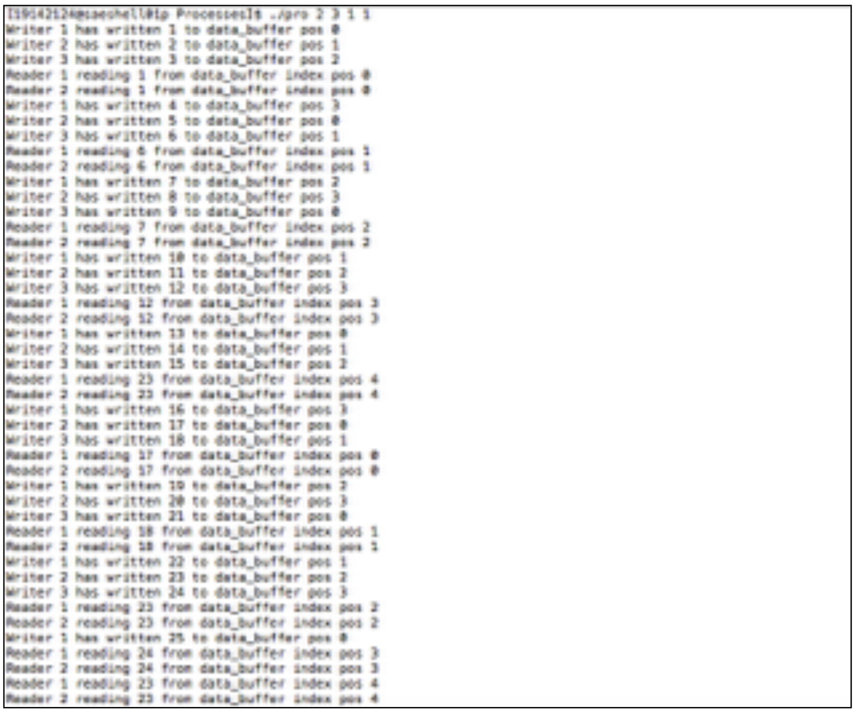
Figure 2: Shared_data file

3.3 Output



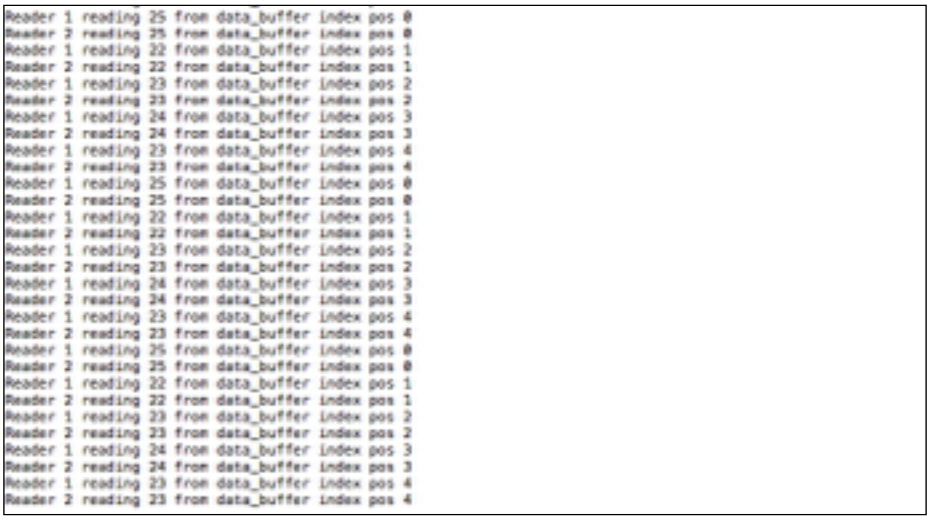
```
[19142124@saeshell01p PThreads]$ ./pt 2 3 1 1 20 5
Writer 1 wrote data 1
Writer 2 wrote data 2
Writer 3 wrote data 3
Reader 1 read data 3
Reader 1 read data 3
Writer 1 wrote data 4
Writer 2 wrote data 5
Writer 3 wrote data 6
Reader 1 read data 6
Reader 1 read data 6
Writer 1 wrote data 7
Writer 2 wrote data 8
Writer 3 wrote data 9
Reader 1 read data 9
Reader 1 read data 9
Writer 1 wrote data 10
Writer 2 wrote data 11
Writer 3 wrote data 12
Reader 1 read data 12
Reader 1 read data 12
Writer 1 wrote data 13
Writer 2 wrote data 14
Writer 3 wrote data 15
Reader 1 read data 15
Reader 1 read data 15
Writer 1 wrote data 16
Writer 2 wrote data 17
Writer 3 wrote data 18
Reader 1 read data 18
Reader 1 read data 18
Writer 1 wrote data 19
Writer 2 wrote data 20
Reader 1 read data 20
Reader 1 read data 20
Reader 1 read data 9
Reader 1 read data 9
Reader 1 read data 12
Reader 1 read data 12
Reader 1 read data 15
```

Figure 3: Pthread Implementation



```
[19142124@saeshell01p Processes]$ ./pro 2 3 1 1
Writer 1 has written 1 to data_buffer pos 0
Writer 2 has written 2 to data_buffer pos 1
Writer 3 has written 3 to data_buffer pos 2
Reader 1 reading 1 from data_buffer index pos 0
Reader 2 reading 1 from data_buffer index pos 0
Writer 1 has written 4 to data_buffer pos 3
Writer 2 has written 5 to data_buffer pos 0
Writer 3 has written 6 to data_buffer pos 1
Reader 1 reading 6 from data_buffer index pos 1
Reader 2 reading 6 from data_buffer index pos 1
Writer 1 has written 7 to data_buffer pos 2
Writer 2 has written 8 to data_buffer pos 3
Writer 3 has written 9 to data_buffer pos 0
Reader 1 reading 7 from data_buffer index pos 2
Reader 2 reading 7 from data_buffer index pos 2
Writer 1 has written 10 to data_buffer pos 1
Writer 2 has written 11 to data_buffer pos 2
Writer 3 has written 12 to data_buffer pos 3
Reader 1 reading 12 from data_buffer index pos 3
Reader 2 reading 12 from data_buffer index pos 3
Writer 1 has written 13 to data_buffer pos 0
Writer 2 has written 14 to data_buffer pos 1
Writer 3 has written 15 to data_buffer pos 2
Reader 1 reading 23 from data_buffer index pos 4
Reader 2 reading 23 from data_buffer index pos 4
Writer 1 has written 16 to data_buffer pos 3
Writer 2 has written 17 to data_buffer pos 0
Writer 3 has written 18 to data_buffer pos 1
Reader 1 reading 17 from data_buffer index pos 0
Reader 2 reading 17 from data_buffer index pos 0
Writer 1 has written 19 to data_buffer pos 2
Writer 2 has written 20 to data_buffer pos 3
Writer 3 has written 21 to data_buffer pos 0
Reader 1 reading 18 from data_buffer index pos 1
Reader 2 reading 18 from data_buffer index pos 1
Writer 1 has written 22 to data_buffer pos 1
Writer 2 has written 23 to data_buffer pos 2
Writer 3 has written 24 to data_buffer pos 3
Reader 1 reading 23 from data_buffer index pos 2
Reader 2 reading 23 from data_buffer index pos 2
Writer 1 has written 25 to data_buffer pos 0
Reader 1 reading 24 from data_buffer index pos 3
Reader 2 reading 24 from data_buffer index pos 3
Reader 1 reading 23 from data_buffer index pos 4
Reader 2 reading 23 from data_buffer index pos 4
```

Figure 4: Process Implementation pt.1



```
Reader 1 reading 25 from data_buffer index pos 0
Reader 2 reading 25 from data_buffer index pos 0
Reader 1 reading 22 from data_buffer index pos 1
Reader 2 reading 22 from data_buffer index pos 1
Reader 1 reading 23 from data_buffer index pos 2
Reader 2 reading 23 from data_buffer index pos 2
Reader 1 reading 24 from data_buffer index pos 3
Reader 2 reading 24 from data_buffer index pos 3
Reader 1 reading 23 from data_buffer index pos 4
Reader 2 reading 23 from data_buffer index pos 4
Reader 1 reading 25 from data_buffer index pos 0
Reader 2 reading 25 from data_buffer index pos 0
Reader 1 reading 22 from data_buffer index pos 1
Reader 2 reading 22 from data_buffer index pos 1
Reader 1 reading 23 from data_buffer index pos 2
Reader 2 reading 23 from data_buffer index pos 2
Reader 1 reading 24 from data_buffer index pos 3
Reader 2 reading 24 from data_buffer index pos 3
Reader 1 reading 23 from data_buffer index pos 4
Reader 2 reading 23 from data_buffer index pos 4
Reader 1 reading 25 from data_buffer index pos 0
Reader 2 reading 25 from data_buffer index pos 0
Reader 1 reading 22 from data_buffer index pos 1
Reader 2 reading 22 from data_buffer index pos 1
Reader 1 reading 23 from data_buffer index pos 2
Reader 2 reading 23 from data_buffer index pos 2
Reader 1 reading 24 from data_buffer index pos 3
Reader 2 reading 24 from data_buffer index pos 3
Reader 1 reading 23 from data_buffer index pos 4
Reader 2 reading 23 from data_buffer index pos 4
```

Figure 5: Process Implementation pt.2

4.0 READ-ME

4.1 Purpose

This program implements the reader-writer problem using both a thread and process implementation. The user can enter any number of readers or writers [max threads being 200 each], and can also specify sleep() values. After reading a supplied shared_data file, a writer writes one bit of data from the shared_data file to the data_buffer (which is a shared resource). A reader reads from this data_buffer one by one. After the threads or processes terminate, the total number of data written or read is output to a file "sim_out".

4.2 File Hierarchy

```
>PThreads
  -pt (Executable)
  -pthread.c
  -pthread.h
  -fileio.c
  -Makefile
  -shared_data (Test file)
  -sim_out (Write-out file)
```

```
>Processes
  -pro (Executable)
  -processes.c
  -processes.h
  -fileio.c
  -Makefile
  -shared_data (Test file)
  -sim_out (Write-out file)
```

4.3 How-to-run

Compile:

```
make
```

Running program:

[For Pthreads]

```
./pt  RDR  WRT  S-R  S-W  D  B
```

[For Processes]

```
./pro  RDR  WRT  S-R  S-W
```

Legend

RDR = readers
WRT = writers
S-R = sleep value for readers
S-W = sleep value for writers
D = size of shared_data
B = size of data_buffer

5.0 Source Code

pthread.c

```
/***********************************************************************
* Author: Eric Wojcik
* Student ID: 19142124
* COMP2006 – Operating Systems Assignment
* Reader–Writer solution (first problem) using pthreads and mutex
* Last Modified: 6/5/2018
*******************************************************************************
****/
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include "pthread.h"

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t wrtCond = PTHREAD_COND_INITIALIZER;
pthread_cond_t rdrCond = PTHREAD_COND_INITIALIZER;
int* data_buffer;
int* shared_data;
/*Global varaible used to increment shared_data array specifically for writers*/
int wIndex=0;
/*Increments shared buffer count for writers*/
int curBuff=0;
/*Used to maintain loop in either reader or writer*/
int isWriting=1,isReading=1;
/*D is shared_data size, where as B is buffer size. Req: B<D*/
int D=0,B=0;
int wID=0,rID=0,wrtCnt=0,rdrCnt=0,numRdr=0,numWrt=0,inc=0,bufferEmpty=1;
void *wtrRoutine(void *arg);
void *rdrRoutine(void *arg);

int main(int argc, char* argv[])
{
  int i,err;
  int* wArr;
  pthread_t* readers;
  pthread_t* writers;
  storage A;
  FILE *simOut=NULL;

  /*Command line validation*/
  validCom(argc,argv);
  numRdr = atoi(argv[1]);
  numWrt = atoi(argv[2]);
  A.sleepR = atoi(argv[3]);
  A.sleepW = atoi(argv[4]);
  D = atoi(argv[5]);
  B = atoi(argv[6]);

  if(B>D)
  {
```

```
  printf(″B value needs to be greater than D value!¥n″);
  exit(1);
}

/*Setting up*/
data_buffer = (int*)malloc(sizeof(int)*B);
shared_data = (int*)malloc(sizeof(int)*D);
A.wrtStore = (int*)malloc(sizeof(int)*numWrt);
A.rdrStore = (int*)malloc(sizeof(int)*numRdr);
readers = (pthread_t*)malloc(sizeof(pthread_t)*numRdr);
writers = (pthread_t*)malloc(sizeof(pthread_t)*numWrt);
arrayInit(data_buffer, B);
arrayInit(shared_data, D);
arrayInit(A.wrtStore, numWrt);
arrayInit(A.rdrStore, numRdr);
eraseFile(simOut);
readData(shared_data,D);

/*Creating pthreads for both readers and writers. Upon creation of a thread,
it will execute a corresponding routine function*/
for(i=0;i<numRdr;i++)
{
  if((err=(pthread_create(&readers[i],NULL,rdrRoutine,&A)))!=0)
  {
    fprintf (stderr, ″Error = %d (%s)¥n″, err, strerror (err));
    exit (1);
  }
}
for(i=0;i<numWrt;i++)
{
  if((err=(pthread_create(&writers[i],NULL,wtrRoutine,&A)))!=0)
  {
    fprintf (stderr, ″Error = %d (%s)¥n″, err, strerror (err));
    exit (1);
  }
}
for(i=0;i<numRdr;i++)
{
  pthread_join(readers[i],NULL);
}
for(i=0;i<numWrt;i++)
{
  pthread_join(writers[i],NULL);
}
/*Printing results to file ″simOut″*/
for(i=0;i<numWrt;i++)
{
  writeOut((int)writers[i],i+1,A.wrtStore[i],simOut,″writer″,″writing″,″to″);
}
for(i=0;i<numRdr;i++)
{
  writeOut((int)readers[i],i+1,A.rdrStore[i],simOut,″reader″,″reading″,″from″);
}

/*Performing cleanup*/
freeFunc(A.wrtStore, A.rdrStore,shared_data,data_buffer,readers,writers);
```

```
  return 0;
}


/
********************************************************************************
****
* Handles routine for any amount of reader threads
* >input:= arg         Void pointer that can take any type
* >output:= NULL
********************************************************************************
****/
void *rdrRoutine(void *arg)
{
  int pid,buffIndex,rBuffer;
  pid=rID;
  buffIndex=0,rBuffer=0;
  storage *A = arg;
  inc=1;
  while(isReading==1)
  {
    if(bufferEmpty != 1)
    {
      pthread_mutex_lock(&mutex);
      /*Checks if there are any writers waiting to write,writing, or both*/
      if(wrtCnt==1)
      {
        pthread_cond_wait(&rdrCond,&mutex);
      }
      rdrCnt++;
      pthread_mutex_unlock(&mutex);
      rBuffer = buffIndex;
      buffIndex++;
      if(buffIndex==B)
      {
        buffIndex = 0;
      }
      pthread_mutex_lock(&mutex);
      if(inc==D)
      {
        isReading=0;
      }
      else
      {
        /*Reading occurs*/
        printf("Reader %d read data %d¥n",pid+1,data_buffer[rBuffer]);
        rdrCnt--;
        A->rdrStore[pid]++;
      }
      if(rdrCnt==0)
      {
        pthread_cond_signal(&wrtCond);
      }
      else
      {
        pthread_cond_signal(&rdrCond);
      }
      inc++;
      pthread_mutex_unlock(&mutex);
      sleep(A->sleepR);
    }
  }
  pthread_exit(NULL);
}

/
************************************************************************
****
* Handles routine for any amount of reader threads
* >input:= arg        Void pointer that can take any type
* >output:= NULL
************************************************************************
****/
void *wtrRoutine(void *arg)
{
  int i,j,pid,temp,buffIndex;
  storage *A = arg;
  pid=wID++;
  while(isWriting==1)
  {
    pthread_mutex_lock(&mutex);
    wrtCnt++;
    /*Check if there is writer queued or reader(s) reading*/
    if((wrtCnt>1)||(rdrCnt>0))
    {
      pthread_cond_wait(&wrtCond,&mutex);
    }
    curBuff = buffIndex;
    buffIndex++;
    pthread_mutex_unlock(&mutex);
    pthread_mutex_lock(&mutex);
    if(buffIndex==B)
    {
      buffIndex = 0;
    }
    if(wIndex>=D)
    {
      isWriting=0;
    }
    else
    {
      /*Writing occurs*/
      data_buffer[curBuff]=shared_data[wIndex];
      bufferEmpty = 0;
      printf(″Writer %d wrote data %d¥n″,pid+1,data_buffer[curBuff]);
      A->wrtStore[pid]++;
      wIndex++;
    }
    wrtCnt−−;
    if(wrtCnt>0)
    {
      pthread_cond_signal(&wrtCond);
    }
    else
    {
        pthread_cond_broadcast(&rdrCond);
    }
    pthread_mutex_unlock(&mutex);
    sleep(A->sleepW);
  }
  pthread_exit(NULL);
}


/
****************************************************************************
****
* Clears file so that it is ready for use
* >input:= simOut   Pointer to FILE simOut
****************************************************************************
****/
void eraseFile(FILE *simOut)
{
  simOut = fopen("sim_out", "w");
  fclose(simOut);
}

/
****************************************************************************
****
* Generic function that writes out results of both readers OR writers to a file
* called 'sim_out'
* >input:= pID         Thread id number
* >input:= rwNum       Current reader/writer number
* >input:= numData     Pieces of data written or read
* >input:= simOut      Pointer to FILE simOut
* >input:= rw1         Phrase 'reader' or 'writer'
* >input:= rw2         Phrase 'read' or 'written'
* >input:= rw3         Phrase 'from' or 'to'
* >output:= shared_data
****************************************************************************
****/
void writeOut(int pID, int rwNum, int numData, FILE *simOut, char* rw1, char* rw2,char*
rw3)
{
  simOut = fopen("sim_out", "a");
  if(simOut!=NULL)
  {
    fprintf(simOut,"%s-%d [pid:%d] has finished %s %d pieces of data %s the data-
buffer¥n",rw1,rwNum,pID,rw2,numData,rw3);
  }
  else /*Check if file was opened properly*/
  {
    perror("Error ");
    exit(1);
  }
  fclose(simOut);
}
```

```
/
**************************************************************************
****
* Frees any allocated memory and destroys mutexes and conditional variables
* >input:= wrtStore  Contains integer array used to count number of bits written
by a given writer
* >input:= rdrStore  Contains integer array used to count number of bits read
by a given reader
* >input:= sd        Shared_data array
* >input:= db        Data_buffer array
**************************************************************************
****/
void freeFunc(int* wrtStore, int* rdrStore, int* sd, int* db, pthread_t* r, pthread_t* w)
{
  pthread_mutex_destroy(&mutex);
  pthread_cond_destroy(&wrtCond);
  pthread_cond_destroy(&rdrCond);
  free(r);
  free(w);
  free(wrtStore);
  free(rdrStore);
}

/
**************************************************************************
****
* Validates the command line parameters, make sure there are correct amount
* >input:= argc      Number of command line parameters (including exec name)
* >input:= argv      Char array containing text obtained from command line
**************************************************************************
****/
void validCom(int argc, char* argv[])
{
  int i;
  if(argc<5)
  {
    printf("Too few arguments. Need to have r,w,t1,t2,d,b¥n");
    exit(1);
  }
  else if(argc>7)
  {
    printf("Too many arguments. Need to have r,w,t1,t2,d,b¥n");
    exit(1);
  }
  for(i=1;i<5;i++)
  {
    if(atoi(argv[i])<0)
    {
      printf("Argument %d needs to be a positive value¥n",i);
      exit(1);
    }
  }
}
```

```
/
****************************************************************************
****
* Generic array initialisation function that sets all values of array to 0
* >input:= array      Any integer array
* >input:= numInc      Any max value
* >output:= array
****************************************************************************
****/
int* arrayInit(int* array, int numInc)
{
  int i;
  for(i=0;i<numInc;i++)
  {
    array[i]=0;
  }
  return array;
}
```

---

pthread.h

```
int* readData(int* shared_data, int arrNum);
void eraseFile(FILE *simOut);
void writeOut(int pID, int rwNum, int numData, FILE *simOut, char* rw1, char* rw2,char* rw3);
void freeFunc(int* wrtStore, int* rdrStore, int* sd, int* db, pthread_t* r, pthread_t* w);
void validCom(int argc, char* argv[]);
int* arrayInit(int* array, int numInc);
typedef struct storage
{
  int sleepR;
  int sleepW;
  int* wrtStore;
  int* rdrStore;
}storage;
```

---

fileio.c (used by both pthread and processes programs)

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

 /
****************************************************************************
****
 * Read the contents of the input file 'shared_data into a shared_data integer
 * array, which is accesible soley by the writer.
 * >input:= shared_data   Integer array to read data into
 * >input:= arrNum         Size of shared_data file (D)
 * >output:= shared_data

****************************************************************************
****/
int* readData(int* shared_data, int arrNum)
{
  int count, n;
  FILE *input = NULL;
  input = fopen("shared_data", "r");
  count = 0;
  if (input==NULL)
  {
    perror("File could not be opened");
    exit(1);
  }

  /*Reads an individual int and scans to a temp int 'n'. Stores 'n' in
  corresponding element position in shared_data array. Repeats until all read*/
  while(fscanf(input, " %d", &n)==1)
  {
    if(count<=arrNum)
    {
      shared_data[count]=n;
    }
    count++;
  }

  if(ferror(input))
  {
    perror("File could not be read");
    exit(1);
  }

  fclose(input); /*Closes the file*/
  return shared_data;
}
```

Makefile (Pthreads)

```
CC = gcc
CFLAGS = -Wall -ansi -pedantic -std=gnu99
OBJ = pthread.c fileio.c
LIBS = -pthread
EXEC = pt

$(EXEC): $(OBJ)
	$(CC) $(OBJ) -o $(EXEC) $(LIBS)

pthread.o: pthread.c pthread.h
	$(CC) -c pthread.c $(CFLAGS)

fileio.o: fileio.c fileio.h
	$(CC) -c fileio.c $(CFLAGS)

clean:
	rm -f $(EXEC) $(OBJ);
```

---

processes.c

```
/
*************************************************************************
****
* Author: Eric Wojcik
* Student ID: 19142124
* COMP2006 - Operating Systems Assignment
* Reader-Writer solution (first problem) using processes and semaphores
* Last Modified: 6/5/2018
*************************************************************************
****/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include "processes.h"
#include <sys/wait.h>

int main(int argc, char* argv[])
{
  int i,j,numRdr,numWrt,sleepR,sleepW;
  int* shared_data;
  int* wrtPIDS;
  int* rdrPIDS;
  FILE *simOut=NULL;

  /*File Descriptors*/
  int
dFD,bFD,wincFD,wbuffcFD,buffEmptyFD,rdrcFD,semFD,databuffFD,rstoreFD,wstoreFD,wi
nFD,iswrtFD;
  /*Pointers for shared memory*/
  int
*dPT,*bPT,*wincPT,*wbuffcPT,*buffEmptyPT,*rdrcPT,*databuffPT,*rstorePT,*wstoreP
T,*winPT,*iswrtPT;
  /*Mutex for readCount, mutex for resource and pointer to shared sem space*/
  sem_t mutex,wrt,*sems;
  pid_t pid;

  /*Command line validation*/
  validCom(argc,argv);

  numRdr = atoi(argv[1]);
  numWrt = atoi(argv[2]);
  sleepR = atoi(argv[3]);
  sleepW = atoi(argv[4]);

  /*Creating shared memory regions and allocating the sizes for them*/

createMemory(&dFD,&bFD,&wincFD,&wbuffcFD,&buffEmptyFD,&rdrcFD,&semFD,&databu
ffFD,&rstoreFD,&wstoreFD,numRdr,numWrt,&winFD,&iswrtFD);
  /*Mapping pointers to the shared memory regions*/
```

```
mapMemAddr(&dFD,&bFD,&wincFD,&wbuffcFD,&buffEmptyFD,&rdrcFD,&semFD,&databuf
fFD,&rstoreFD,&wstoreFD,numRdr,numWrt,
&dPT,&bPT,&wincPT,&wbuffcPT,&buffEmptyPT,&rdrcPT,&databuffPT,&rstorePT,&wstore
PT,&sems,&winPT,&iswrtPT,&winFD,&iswrtFD);

  /*Initialise sempahores with non-zero value meaning it can be shared between
processes*/
  if((sem_init(&mutex,1,1)==1)||(sem_init(&wrt,1,1)==1))
  {
    perror("Semaphore initialisation failed");
    exit(1);
  }

  /*Allocating sempahores to shared memory space via the sems array*/
  sems[0]=mutex;
  sems[1]=wrt;

  /*Remaining set-up*/
  shared_data = (int*)malloc(sizeof(int)*D);
  wrtPIDS = (int*)malloc(sizeof(int)*numWrt);
  rdrPIDS = (int*)malloc(sizeof(int)*numRdr);
  *winPT = 0;
  *iswrtPT= 1;
  *wbuffcPT = 0;
  *buffEmptyPT=0;
  *rdrcPT=0;
  shared_data = arrayInit(shared_data,D);
  wstorePT = arrayInit(wstorePT,numWrt);
  rstorePT = arrayInit(rstorePT,numRdr);
  wrtPIDS = arrayInit(wrtPIDS,numWrt);
  rdrPIDS = arrayInit(rdrPIDS,numRdr);
  readData(shared_data,D);
  eraseFile(simOut);

  /*Creating reader and writer processes*/
  for(i=0;i<numWrt;i++)
  {
    if((pid=fork())==0)
    {

wrtRoutine(i,iswrtPT,winPT,wbuffcPT,buffEmptyPT,wstorePT,databuffPT,shared_data,se
ms,sleepW);
      exit(0);
    }
    else if(pid<0)
    {
      perror("Could not create child process!");
    }
  }
  for(i=0;i<numRdr;i++)
  {
    if((pid=fork())==0)
    {
      rdrRoutine(i,buffEmptyPT,rstorePT,databuffPT,sems,rdrcPT,sleepR);
      exit(0);
    }
    else if(pid<0)
    {
      perror("Could not create child process!");
    }
  }
  /*Parent waiting for all child processes to finish to avoid zombie processes*/
  while((pid=wait(0))>0);
  /*Processes Complete*/
  for(i=0;i<numWrt;i++)
  {
    writeOut(0,i+1,wstorePT[i],simOut,"writer","writing","to");
  }
  for(i=0;i<numRdr;i++)
  {
    writeOut(0,i+1,rstorePT[i],simOut,"reader","reading","from");
  }
  /*Cleaning up process*/
  if(pid>0)
  {
    /*Cleaning up semaphores*/
    sem_close(&(sems[0]));
    sem_close(&(sems[1]));
    sem_destroy(&(sems[0]));
    sem_destroy(&(sems[1]));

    /*Cleaning up file descriptors*/
    close(dFD);
    close(bFD);
    close(wincFD);
    close(wbuffcFD);
    close(buffEmptyFD);
    close(rdrcFD);
    close(semFD);
    close(databuffFD);
    close(rstoreFD);
    close(wstoreFD);
    close(iswrtFD);

    /*Clearing out the shared memory regions*/
    shm_unlink("/D_val");
    shm_unlink("/B_val");
    shm_unlink("/winnFD");
    shm_unlink("/iswrting");
    shm_unlink("wincFD");
    shm_unlink("/wrt_buffcnt");
    shm_unlink("/buff_empty");
    shm_unlink("/read_cnt");
    shm_unlink("/sems");
    shm_unlink("/db");
    shm_unlink("/rdr_store");
    shm_unlink("/wrt_store");

    /*Clearing out mapped memory regions*/
    munmap(dPT,sizeof(int));
    munmap(bPT, sizeof(int));
    munmap(wincPT, sizeof(int));
    munmap(iswrtPT, sizeof(int));
    munmap(wbuffcPT, sizeof(int));
    munmap(buffEmptyPT, sizeof(int));
    munmap(rdrcPT, sizeof(int));
    munmap(sems, sizeof(sem_t)*2);
    munmap(databuffPT, sizeof(int)*B);
    munmap(rstorePT, sizeof(int)*numRdr);
    munmap(wstorePT, sizeof(int)*numWrt);
  }
  free(shared_data);
  free(wrtPIDS);
  free(rdrPIDS);
  return 0;
}


/
***************************************************************************
****
* Generic array initialisation function that sets all values of array to 0
* >input:= array      Any integer array
* >input:= numInc     Any max value
* >output:= array
***************************************************************************
****/
int* arrayInit(int* array, int numInc)
{
  int i;
  for(i=0;i<numInc;i++)
  {
    array[i]=0;
  }
  return array;
}

/
***************************************************************************
****
 * Validates the command line parameters, make sure there are correct amount
 * >input:= argc      Number of command line parameters (including exec name)
 * >input:= argv      Char array containing text obtained from command line
 *
***************************************************************************
****/
void validCom(int argc, char* argv[])
{
  int i;
  if(argc<5)
  {
    printf("Too few arguments. Need to have r,w,t1,t2,d,b¥n");
    exit(1);
  }
  else if(argc>7)
  {
    printf("Too many arguments. Need to have r,w,t1,t2,d,b¥n");
    exit(1);
  }
  for(i=1;i<5;i++)
  {
    if(atoi(argv[i])<0)
    {
      printf(″Argument %d needs to be a positive value¥n″,i);
      exit(1);
    }
  }
}


/
*******************************************************************************
****
* Handles routine for any amount of reader processes
* >input:= pid            Numerical identifier for current reader
* >input:= buffEmptyPT    Pointer to variable used to indicate if buffer is empty
* >input:= rstorePT       Pointer to array that stores the amount of bits read by any
given reader
* >input:= databuffPT     Pointer to array that serves as the shared buffer
* >input:= sems           Array of semaphores, contains semaphores wrt and mutex
* >output:= NULL
*******************************************************************************
****/
void rdrRoutine(int pid, int* buffEmptyPT, int* rstorePT, int* databuffPT, sem_t*
sems,int* readCount,int sleepR)
{
  int i,isReading,bufferIndex,curIndex,rinc,localRCnt;
  isReading = 1,bufferIndex=0,curIndex=0,rinc=0,localRCnt=0;
  for(i=0;i<D;i++)
  {
    if(isReading==1)
    {
      sem_wait(&(sems[0]));
      *readCount = *readCount + 1;
      /*If first reader will lock out writer*/
      if(*readCount==1)
      {
        sem_wait(&(sems[1]));
      }
      sem_post(&(sems[0]));
      /*Releases mutex to allow another reader to read at same time*/
      curIndex=bufferIndex;
      bufferIndex++;
      if(bufferIndex==B)
      {
        bufferIndex=0;
      }
      if(rinc==D)
      {
        isReading = 0;
        localRCnt=0;
      }
      else
      {
        printf("Reader %d reading %d from data_buffer index pos
%d¥n",pid+1,databuffPT[curIndex],curIndex);
        localRCnt=1;
      }
      sem_wait(&(sems[0]));
      rstorePT[pid]=rstorePT[pid]+localRCnt;
      *readCount = *readCount-1;
      /*Checks if the last reader is done, will then unlock writer*/
      if(*readCount==0)
      {
        sem_post(&(sems[1]));
      }
      rinc++;
      sem_post(&(sems[0]));
      sleep(sleepR);
    }
  }
}


/
**************************************************************************
****
* Handles routine for any amount of writer processes
* >input:= pid            Numerical identifier for current writer
* >input:= stop            This variable determines when a writer needs to stop writing
* >input:= wincPT          Variable used to increment through the shared_data array. This
is in shared memory so that writers can share the load
* >input:= buffEmptyPT     Pointer to variable used to indicate if buffer is empty
* >input:= wstorePT        Pointer to array that stores the amount of bits written by any
given writer
* >input:= databuffPT      Pointer to array that serves as the shared buffer
* >input:= shared_data     Array of ints, stores shared_data read in from the shared_data
file. Contents are written to data_buffer
* >input:= sems            Array of semaphores, contains semaphores wrt and mutex
* >output:= NULL
**************************************************************************
****/
void wrtRoutine(int pid, int* stop, int* wincPT, int* wbuffcPT,int* buffEmptyPT,int*
wstorePT, int* databuffPT,
int* shared_data,sem_t* sems,int sleepW)
{
  int i,bufferIndex,localWCnt,process;
  bufferIndex=0,localWCnt=0;
  for(i=0;i<D;i++)
  {
    if(*stop==1)
    {
      sem_wait(&(sems[1]));
      bufferIndex = *wbuffcPT;
      *wbuffcPT=*wbuffcPT+1;
      if(*wbuffcPT==B-1)
      {
        *wbuffcPT=0;
      }
      if(*wincPT==D)
      {
        *stop = 0;
        localWCnt=0;
      }
      else
      {
        printf(″Writer %d has written %d to data_buffer pos
%d¥n″,pid+1,shared_data[*wincPT],bufferIndex);
        databuffPT[bufferIndex]=shared_data[*wincPT];
        *buffEmptyPT=0;
        localWCnt=1;
      }
      *wincPT=*wincPT+1;
      wstorePT[pid]=wstorePT[pid]+localWCnt;
      sem_post(&(sems[1]));
      sleep(sleepW);
    }
  }
}

/
***************************************************************************
****
 * Clears file so that it is ready for use
 * >input:= simOut   Pointer to FILE simOut

***************************************************************************
****/
void eraseFile(FILE *simOut)
{
  simOut = fopen(″sim_out″, ″w″);
  fclose(simOut);
}

/
***************************************************************************
****
 * Generic function that writes out results of both readers OR writers to a file
 * called ’sim_out’
 * >input:= pID        Thread id number
 * >input:= rwNum       Current reader/writer number
 * >input:= numData    Pieces of data written or read
 * >input:= simOut     Pointer to FILE simOut
 * >input:= rw1        Phrase ’reader’ or ’writer’
 * >input:= rw2        Phrase ’read’ or ’written’
 * >input:= rw3        Phrase ’from’ or ’to’
 * >output:= shared_data
 *
***************************************************************************
****/
void writeOut(int pID, int rwNum, int numData, FILE *simOut, char* rw1, char* rw2,char*
rw3)
{
  simOut = fopen(″sim_out″, ″a″);
  if(simOut!=NULL)
  {
    fprintf(simOut,"%s-%d [pid:%d] has finished %s %d pieces of data %s the data-
buffer¥n",rw1,rwNum,pID,rw2,numData,rw3);
  }
  else /*Check if file was opened properly*/
  {
    perror("Error ");
    exit(1);
  }
  fclose(simOut);
}

/
**************************************************************************
****
 * Maps each supplied pointer to a memory address via the use of a file descriptor.
 * >input:= dFD             File descriptor for variable D (size of shared_data)
 * >input:= bFD             File descriptor for variable B (size of data_buffer)
 * >input:= wincFD           File descriptor for writer increment variable used to go through
shared_data
 * >input:= wbuffcFD          File descriptor for variable used to increment through buffer
(allows writers to share load)
 * >input:= buffEmptyFD    File descriptor for variable used to determine if buffer empty
 * >input:= rdrcFD           File descriptor for variable read count
 * >input:= semFD            File descriptor for semaphores
 * >input:= databuffFD      File descriptor for data_buffer
 * >input:= rstoreFD         File descriptor for reader array storage (for amount read)
 * >input:= wstoreFD          File descriptor for writer array storage (for amount written)
 * >input:= winFD            File descriptor for writer variable win
 * >input:= iswrtFD          File descriptor for is writing
 * >input:= numRdr           Integer representing number of readers
 * >input:= numWrt           Integer representing number of writers
 * >input:= dPT             Points to new memory address for variable D
 * >input:= bPT             Points to new memory address for variable B
 * >input:= wincPT           Points to new memory address for variable writer increment
 * >input:= wbuffcPT          Points to new memory address for variable writer buffer
increment
 * >input:= buffEmptyPT    Points to new memory address for variable empty buffer
 * >input:= rdrcPT           Points to new memory address for variable reader count
 * >input:= databuffPT      Points to new memory address for data_buffer array
 * >input:= rstorePT         Points to new memory address for reader storage array
 * >input:= wstorePT          Points to new memory address for writer storage array
 * >input:= sems             Points to new memory address for semaphore storage array
 * >input:= winPT            Points to new memory address for variable writer increment
 * >input:= iswrtPT          Points to new memory address for variable is writing

**************************************************************************
****/
void mapMemAddr(int* dFD, int* bFD, int* wincFD, int* wbuffcFD, int* buffEmptyFD,
int* rdrcFD, int* semFD, int* databuffFD,
int*rstoreFD, int* wstoreFD, int numRdr, int numWrt, int** dPT,int** bPT,int**
wincPT,int** wbuffcPT,int** buffEmptyPT,
int** rdrcPT,int(**databuffPT),int(**rstorePT),int(**wstorePT),sem_t** sems,int**
winPT,int** iswrtPT,int* winFD, int* iswrtFD)
{
  *dPT = (int*) mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED,
*dFD, 0);
  *bPT = (int*) mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED,
*bFD, 0);
  *wincPT = (int*) mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED,
*wincFD, 0);
  *winPT = (int*) mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED,
*winFD, 0);
  *iswrtPT = (int*) mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED,
*iswrtFD, 0);
  *wbuffcPT = (int*) mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE,
MAP_SHARED, *wbuffcFD, 0);
  *buffEmptyPT = (int*) mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE,
MAP_SHARED, *buffEmptyFD, 0);
  *rdrcPT = (int*) mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED,
*rdrcFD, 0);
  *sems = mmap(NULL, sizeof(sem_t)*2, PROT_READ | PROT_WRITE, MAP_SHARED,
*semFD, 0);
  *databuffPT = (int*) mmap(NULL, sizeof(int)*B, PROT_READ | PROT_WRITE,
MAP_SHARED, *databuffFD, 0);
  *rstorePT = (int*) mmap(NULL, sizeof(int)*numRdr, PROT_READ | PROT_WRITE,
MAP_SHARED, *rstoreFD, 0);
  *wstorePT = (int*) mmap(NULL, sizeof(int)*numWrt, PROT_READ | PROT_WRITE,
MAP_SHARED, *wstoreFD, 0);

  if((*dPT==MAP_FAILED)||(*bPT==MAP_FAILED)||(*wincPT==MAP_FAILED)||
(*wbuffcPT==MAP_FAILED)||(*buffEmptyPT==MAP_FAILED)||
  (*rdrcPT==MAP_FAILED)||(*sems==MAP_FAILED)||(databuffPT==MAP_FAILED)||
(rstorePT==MAP_FAILED)||(wstorePT==MAP_FAILED)
  ||(*winPT==MAP_FAILED)||(*iswrtPT==MAP_FAILED))
  {
    perror("Failed to map to memory space");
    exit(1);
  }
}



/
***************************************************************************
****
 * Creates a shared memory region for each file descriptor so that they can be
accessed by
 * all processes and child processes.
 * >input:= dFD            File descriptor for variable D (size of shared_data)
 * >input:= bFD            File descriptor for variable B (size of data_buffer)
 * >input:= wincFD          File descriptor for writer increment variable used to go through
shared_data
 * >input:= wbuffcFD        File descriptor for variable used to increment through buffer
(allows writers to share load)
 * >input:= buffEmptyFD     File descriptor for variable used to determine if buffer empty
 * >input:= rdrcFD          File descriptor for variable read count
 * >input:= semFD           File descriptor for semaphores
 * >input:= databuffFD      File descriptor for data_buffer
 * >input:= rstoreFD        File descriptor for reader array storage (for amount read)
 * >input:= wstoreFD        File descriptor for writer array storage (for amount written)
 * >input:= winFD          File descriptor for writer variable win
 * >input:= iswrtFD        File descriptor for is writing
 * >input:= numRdr          Integer representing number of readers
 * >input:= numWrt          Integer representing number of writers

*************************************************************************
****/
void createMemory(int* dFD, int* bFD, int* wincFD, int* wbuffcFD, int* buffEmptyFD,
int* rdrcFD, int* semFD, int* databuffFD,
 int*rstoreFD, int* wstoreFD, int numRdr, int numWrt,int* winFD,int*iswrtFD)
{

  /*Creating shared memory region for every variable/struture*/
  *dFD = shm_open("/D_val", O_CREAT | O_RDWR,0666);
  *bFD = shm_open("/B_val", O_CREAT | O_RDWR,0666);
  *winFD = shm_open("/winnFD", O_CREAT | O_RDWR,0666);
  *iswrtFD = shm_open("/iswrting", O_CREAT | O_RDWR,0666);
  *wincFD = shm_open("wincFD", O_CREAT | O_RDWR,0666);
  *wbuffcFD = shm_open("/wrt_buffcnt", O_CREAT | O_RDWR,0666);
  *buffEmptyFD = shm_open("/buff_empty", O_CREAT | O_RDWR,0666);
  *rdrcFD = shm_open("/read_cnt", O_CREAT | O_RDWR, 0666);
  *semFD = shm_open("/sems", O_CREAT | O_RDWR, 0666);
  *databuffFD = shm_open("/db", O_CREAT | O_RDWR,0666);
  *rstoreFD = shm_open("/rdr_store", O_CREAT | O_RDWR,0666);
  *wstoreFD = shm_open("/wrt_store", O_CREAT | O_RDWR,0666);

  if((*dFD==-1)||(*bFD==-1)||(*wincFD==-1)||(*wbuffcFD==-1)||(*buffEmptyFD==-1)||
(*rdrcFD==-1)||(*semFD==-1)||(*databuffFD==-1)
||(*rstoreFD==-1)||(*wstoreFD==-1)||(*winFD==-1)||(*iswrtFD==-1))
  {
    perror("Failed to create memory space ");
    exit(1);
  }
  /*Set size of shared memory objects*/
  if(ftruncate(*dFD, sizeof(int))==-1)
  {
    perror("Failed to configure size for D");
    exit(1);
  }
  if(ftruncate(*bFD, sizeof(int))==-1)
  {
    perror("Failed to configure size for B");
    exit(1);
  }
  if(ftruncate(*wincFD, sizeof(int))==-1)
  {
    perror("Failed to configure size for writer inc");
  }
  if(ftruncate(*winFD, sizeof(int))==-1)
  {
    perror("Failed to configure size for writer inc");
    exit(1);
  }
  if(ftruncate(*iswrtFD, sizeof(int))==-1)
  {
    perror("Failed to configure size of writer stop value");
    exit(1);
  }
  if(ftruncate(*wbuffcFD, sizeof(int))==-1)
  {
    perror("Failed to configure size for writer buffer count");
    exit(1);
  }
  if(ftruncate(*buffEmptyFD, sizeof(int))==-1)
  {
    perror("Failed to configure size for buffer empty");
    exit(1);
  }
  if(ftruncate(*semFD, sizeof(sem_t)*2)==-1)
  {
    perror("Failed to configure size for mutex and wrt semaphores");
    exit(1);
  }
  if(ftruncate(*rdrcFD, sizeof(int))==-1)
  {
    perror("Failed to configure size for reader  count");
    exit(1);
  }
  if(ftruncate(*databuffFD, sizeof(int)*D)==-1)
  {
    perror("Failed to configure size for data_buffer");
    exit(1);
  }
  if(ftruncate(*rstoreFD, sizeof(int)*numRdr)==-1)
  {
    perror("Failed to configure size for reader data count array");
    exit(1);
  }
  if(ftruncate(*wstoreFD, sizeof(int)*numWrt)==-1)
  {
    perror("Failed to configure size for writer data count");
    exit(1);
  }
}
```

Makefile (Processes)

```
GCC = gcc
CFLAGS = -Wall -ansi -pedantic
OBJ = processes.c fileio.c
LIBS = -pthread -lrt -std=gnu99
EXEC = pro

$(EXEC): $(OBJ)
	$(CC) $(OBJ) -o $(EXEC) $(LIBS)

pthread.o: processes.c processes.h
	$(CC) -c pthread.c $(CFLAGS)

fileio.o: fileio.c fileio.h
```

```
	$(CC) −c fileio.c $(CFLAGS)

clean:
	rm −f $(EXEC) $(OBJ);
```