Operating System Individual Coursework

CS2AO17: Operating System

University of Reading

Department of Computer Science

Shavin Croos

Module Code:	**CS2AO17**
Assignment report title:	**Operating System Technical Report**
Student Number:	**27015244**
Date (when the work completed):	**03/03/2021**
Actual hrs spent for the assignment:	**24+ Hours**

Assignment evaluation (3 key points):

1)The page limit for this report was not enough for the code placed into the report.

2) When asking for code excerpts, it means to have snippets of code in the report, not to place entire pieces of code.

3) Gave intuition to learn about how these codes worked.

Contents

Introduction…… 2

Lab 1 - Linux introduction, Shell programming..... 2

Question 1 - Shell Programming…….. 2

Question 2 - The Purpose of Bash……... 2

Question 3 - Enjoy life! You're still young!..... 2

Lab 2 - System calls…… 3

Question 4 - Parent & Child Processes…….. 3

Lab 3 - Pipes and Forks…… 4

Question 5 - Pipes setup, Process & Execution…… 4

Question 6 - Pipes & Processes…… 4

Lab 4 - Concurrency…… 5

Question 7 - Performance Analysis…….. 5

Question 8 - Pthreads Utilisation…….. 6

Question 9 - Mutual Exclusion……... 6

Lab 5 - Inter Process Communication……... 6

Question 10 - Normal & Urgent Messages…….. 6

Question 11 - Improved kirk.c ..…… 6

Lab 6 - Memory management …….. 7

Question 12 - 1MB Page Size…… 7

Question 13 - Large Files being Mapped…… 8

Bibliography…….. 9

Repository…….. 9

Appendices……... 10

Appendix A - Q6.c: Piping and processing…… 10

Appendix B - Q8.c: Pthread usage…… 11

Appendix C - Q9.c: Mutual Exclusions…… 12

Appendix D - kirk2.c: Inter Process Communication…… 14

Appendix E - spock2.c: Non-Urgent Receiver ……..…… 15

Appendix F - starfleet.c: Urgent Receiver .....…… 16

Appendix G - kirk3.c: Signal Handlers …… 17

Appendix H - mmapdemo2.c: Larger Files being Mapped …… 18

Introduction

This lab report looks into the use of the Linux commands and scripts that have helped me to hone in the skills in building and executing operating systems. Such skills include system calls, CPU scheduling, process management and file systems. The details that this report will show will explain what problems are overcome and how these labs help me to build the necessary skills to finish the tasks. This report contains excerpts of code in the Appendix as I had to exceed the page limit to ensure that all the relevant code is there and this was approved by Dr Julian Kunkel.

Lab 1 - Linux introduction, Shell programming

Question 1 - Shell Programming

By adding the -o flag within the strace command that was used in E5, as shown in **Figure 1**, the output was sent to a file named “mycat.out”, which contained the summarised details of the output obtained from using the strace command. At the bottom of **Figure 1**, there was a table that showed summarised details of 15 system calls in total, of which there had been 3 read system calls, 6 write system calls, 3 open system calls and 3 close system calls executed. The further details of how these system calls were being managed during the runtime had been specified above the table.

```
at815244@nxnode2:~/cs2ao17_Practicals/cs2ao17/lab_1$ strace -C -e trace=open,close,read,write -o mycat.out ./mycat cmd2.sh
# Passing arguments to a shell script
echo 1. argument: $1
echo 2. argument: $2
echo 3. argument: $3
echo Total number of arguments: $#
echo All arguments: $*
at815244@nxnode2:~/cs2ao17_Practicals/cs2ao17/lab_1$ cat mycat.out
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
close(3)                                = 0
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0`\t\2\0\0\0\0\0"..., 832) = 832
close(3)                                = 0
open("cmd2.sh", O_RDONLY)               = 3
read(3, "# Passing arguments to a shell s"..., 1048576) = 159
write(1, "# Passing arguments to a shell s"..., 38) = 38
write(1, "echo 1. argument: $1\n", 21)  = 21
write(1, "echo 2. argument: $2\n", 21)  = 21
write(1, "echo 3. argument: $3\n", 21)  = 21
write(1, "echo Total number of arguments: "..., 35) = 35
write(1, "echo All arguments: $*\n", 23) = 23
read(3, "", 1048576)                    = 0
close(3)                                = 0
+++ exited with 0 +++
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
  0.00    0.000000           0         3           read
  0.00    0.000000           0         6           write
  0.00    0.000000           0         3           open
  0.00    0.000000           0         3           close
------ ----------- ----------- --------- --------- ----------------
100.00    0.000000                    15           total
```

Figure 1 - Strace command being executed in Question 1

Question 2 - The Purpose of Bash

Bash, which stands for Bourne Again Shell, is a shell interpreter that takes the input from a file called a shell script and executes the command. This means that it takes the commands in the form of plain text and calls the operating system services to do something and return the outcome of the constructed code. (Hiwarale, 2019)

Question 3 - Enjoy life! You’re still young!

Code 1 shows the age2.sh script. age2.sh is a simple age checker that returns the appropriate statement, through the use of a simple if-else statement, back to the user, depending on the user’s input.

```
#!/bin/bash
# Shell script to check if the person is younger than 20, if true, then
person is young.
echo "Enter your name and age: "
read answer
read answer2
# Reads input from answer2 and compares with the value of the if statement.
If answer less than 20, then prints first statement, else will print second
statement.
if [[ "$answer2" -lt 20 ]]
then
# Echo commands which prints the name and age of the user respectively
depending on the if statement input comparison.
      echo ""
      echo "$answer, you are still young! Enjoy life!"
else
      echo ""
      echo "$answer, you are $answer2 and getting old, you are now
working!"
fi
```

Code 1 - age2.sh Script

Lab 2 - System calls

Question 4 - Parent & Child Processes

Written in **Code 2** is a simple example of system calls in action. In this code, 2 child variables are initialised and are assigned to the fork() function. A for loop is then created next, which is initialised with a counter starting from 1 and is incremented each time by 1 until the counter reaches 10. In this for loop, due to each fork() assignment to the child variables, the “child process” and “parent process” print messages are constantly being alternated between the two. The process id and the counter number are also printed whilst this happens. For each increment, there is a 1 second delay before the loop is repeated again until the counter value of 10 is reached.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main (){
// child processes initialised and forked at once.
  int kid1 = fork();
  int kid2 = fork();
// iteration of the process ids creation for 10 increments.
  for (int counter = 1; counter <= 10; counter++){
    if (kid1 == 0){
    printf("child %d process: counter = %d\n", getpid(), counter);
  } else if (kid2 == 0){
    printf("parent %d process: counter = %d\n", getppid(), counter);
  }
```

```
  // sleep(1) delays the printed results for 1 second each on the terminal.
    sleep(1);
  }
  return 0;
}
```

Code 2 - Q4.c

Lab 3 - Pipes and Forks

Question 5 - Pipes setup, Process & Execution

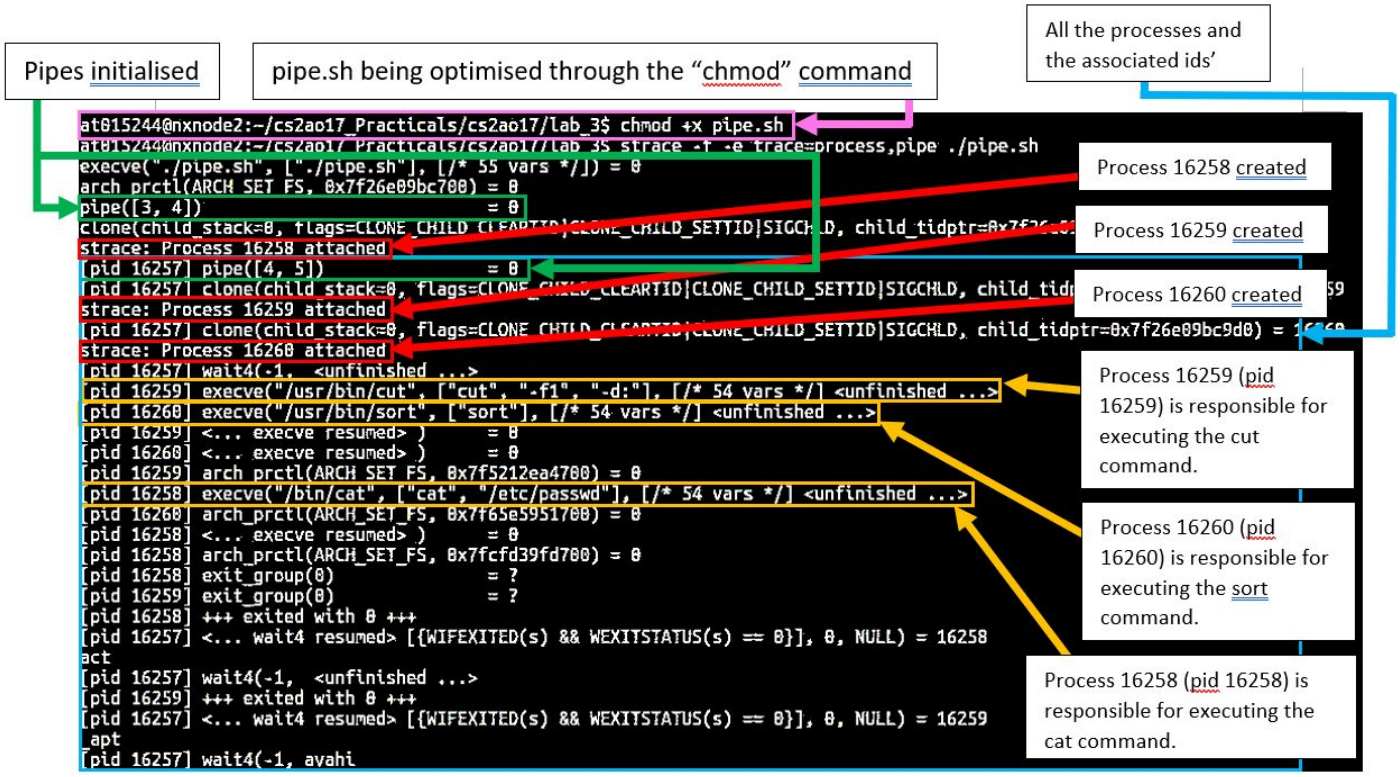

Figure 2 - Strace output with the use of process and pipe on the pipeline "cat /etc/passwd|cut -f1 -d:|sort"

Question 6 - Pipes & Processes

In the code of **Appendix A**, 2 pipes are initialised in the form of arrays, where the value of '2' is the array size which shows an input and output for each pipe. The first pipe is initialised and the first switch case is called, where an initial error check is carried out. During the switch case, when the fork() function is 0, which means that the function is in it's child process, then the second pipe is initialised, along with the second switch case. This is where the main pipeline "cat /etc/group|sort|cut -f3 -d:" is broken down and executed in sections. The cat command is executed by case 0 and the resulting output is piped through to the sort command. Next, sort command is executed and it's output is piped through the cut command and processed, which finally terminates the command. These commands can be seen in **Figure 3** and the result of the strace command.

Figure 3 - Strace output with the use of process and pipe on the pipeline "cat /etc/group|sort|cut -f3 -d:"

Lab 4 - Concurrency

Question 7 - Performance Analysis

N value (number of processes)	Average CPU% usage (approx. 1.d.p)
2	100.0
4	100.0
8	99.7
16	83.7

Taking note of the CPU usage, through the use of the top command, and executing the code needed for the question, the use of 2 processes and 4 processes resulted in the CPU usage being at 100%. The CPU usage only dropped to 99.7% when 8 processors were in use, but the load was evenly distributed across the CPU. Finally, 83.7% of the CPU was used when 16 processors were used, but the distribution of the load across the CPU was quite uneven.

The CPU usage was at 100% for the thread values of 2 and 4, as the amount of the data needed to be handled by the CPU was higher than expected. This could be due to other processes that were running at the same time in the background unrelated to the running of the coded program used in the question. Another reason why the CPU usage is at 100% could be due to the use of an older CPU (Intel Core i7-8750H). Furthermore, this high reading could also be caused by the fact I was using NoMachine (an old virtual machine), rather than VirtualBox's one. However, the CPU usage did decrease generally as the number of processes increased, since more threads were being used, meaning that the computing resources were being divided according to the number of forks carried out.

Question 8 - Pthreads Utilisation

In **Appendix B**, the number of threads defined in the code has been set to 5 and we keep the running total as a global variable. The code then initialises these threads and their associated arguments. During the for loop, the number of threads is printed and is incremented until the value is less than the value of the arguments set initially. In the loop, the function ‘perform work’ is called, which carries out the computation inside and returns the total, until the thread processing is complete. The processed threads are returned as an output in main, where it is printed to the user.

Question 9 - Mutual Exclusion

As **Appendix C** is similar to **Appendix B**, but uses mutex exclusion in the code. Again, the number of threads defined in the code is set to 5 and the global variable is the running total. The code initialises the threads to perform the calculations. Next, the for loop is created and during this loop, the appropriate number of processes for these respective threads to calculate on are created, which leaves the child processes left to execute. After this the process ID is printed out and calls ‘perform work’ to carry out the computation. This allows for different packages of data to be manipulated on each thread and then frees it whilst waiting for the other threads to complete their tasks. Once the other threads complete the assigned tasks and are joined, the outputs are returned and printed to the user.

Lab 5 - Inter Process Communication

Question 10 - Normal & Urgent Messages

Based on the kirk.c file and by adapting it, **Appendix D** demonstrates direct messaging between a client and two receivers. In order to check if the message is typed in lowercase, a simple if statement is initialised to check whether the user’s input as a string is typed in lowercase characters or not. If the whole string consists of lowercase characters, the message is sent Spock only. On the other hand, if the whole string is typed in uppercase characters, then the message is sent to Starfleet only. In order for the code to identify which receivers should get the appropriate messages, ‘msgsend’ has a double equals operand, which dictates that ‘-1’ will send messages to Spock only and ‘0’ will send messages to Starfleet only. The message id and key is also repeated in **Appendix F** and **Appendix E**, as well as **Appendix D**. **Figure 4** illustrates the result of the text chat between Spock, Starfleet and kirk.

Figure 4 - Sending normal and urgent messages to Spock and Starfleet respectively

Question 11 - Improved kirk.c

The adapted version of kirk.c requires the use of signals in the code, which is looked at in detail in **Appendix G**. The signal handler ‘sigint_handler’ is initialised as a void function to check if this message is returned when the user terminates the program. The signals are held inside a constructor named ‘sa’, which allows them to access the signal handler stored in the struct, via ‘sa.sa handler’, and assign it to the key signal handler; ‘sigint_handler’. ‘sa.sa _flags’ verifies if the termination of the program is true and if so, the process will be reset and restarted. The function ‘sigemptyset’ empties the sets and does not prevent any other signals from reaching the program. When ^C is typed, the program calls the key signal handler and returns the message inside. **Figure 5** demonstrates the signal handler in action.

```
at015244@nxnode2:~/cs2ao17_Practicals/cs2ao17/lab_5$ ./kirk3
Enter lines of text, ^C to quit:
^C
Signal Terminated! Emergency transporter sent!
at015244@nxnode2:~/cs2ao17_Practicals/cs2ao17/lab_5$
```

Figure 5 - Signal termination in kirk

Lab 6 - Memory management

Question 12 - 1MB Page Size

vaddr.c is altered in Code 3 in order to provide the address that uses a 1MB page size. The offset for the address in vaddr.c was initially set up to 4KB, but to accommodate an address that uses a 1MB page size, the size of 1MB in bytes had to be determined, which was 1000000 bytes. Next, the power of 2 that gave 1MB in bytes had to be found, which was done by calculating the log of base 2 to 1000000 bytes. This returned the approximate value of 20. When inputting this value as the power of 2, the result returned was 1048576, which was the size of 1MB in bytes. The address 1048576 was typed in and the page number and offset values were returned, as shown in **Figure 6**.

```c
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    unsigned long page, offset, address;
    if(argc != 2)  exit(1);
    address= atoll(argv[1]);

    page = address >> 20;          /*calculating pages number*/
    offset = address & 0xfff;      /*calculating remaining offset*/

    printf("The address %lu contains: \n", address);
    printf("page number = %lu\n",page);
    printf("offset = %lu\n", offset);
    return 0;
}
```

Code 3 - vaddr2.c

```
at015244@nxnode2:~/cs2ao17_Practicals/cs2ao17/lab_6$ ./vaddr2 1048576
The address 1048576 contains:
page number = 1
offset = 0
at015244@nxnode2:~/cs2ao17_Practicals/cs2ao17/lab_6$
```

Figure 6 - Inputted 1MB address and returned page and offset value

Question 13 - Large Files being Mapped

Appendix G demonstrates the use of chunking. First of all, the size of the chunk is set and the chunk was obtained by dividing the offset retrieved by the defined chunk size. The offset is provided from the user input, which designates the chunk to be mapped, which tells us that 1MB is the size that each chunk will be at. The memory is obtained from the offset that is inputted. Dividing the offset by the chunk size will allow us to understand the number of chunks that are required to meet the set byte size. The corresponding ASCII character is obtained through the char pointer 'data'. In the last step, by combining mmap with the size of the chunk and beginning at the chunk offset, the byte can be found by determining the offset modulus chunk size. **Figure 7** illustrates the mmapdemo2.c strace output.

One benefit of using MMAP is when MMAP is reading from and writing to a memory-mapped file, it avoids the unnecessary copy from happening when using the read or write system calls. Aside from any potential page faults, reading from and writing to a memory-mapped file does not incur any system call or context switch overhead, meaning it is as simple as accessing memory. (Zhang, 2020)

```
at015244@nxnode2:~/cs2ao17_Practicals/cs2ao17/lab_6$ strace -C -e trace=open,close,read,write ./mmapdemo2 53
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
close(3)                                = 0
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0`\t\2\0\0\0\0\0"..., 832) = 832
close(3)                                = 0
open("mmapdemo2.c", O_RDONLY)           = 3
write(1, "byte at offset 0 is '47'\n", 25byte at offset 0 is '47'
) = 25
+++ exited with 0 +++
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
  0.00    0.000000           0         1           read
  0.00    0.000000           0         1           write
  0.00    0.000000           0         3           open
  0.00    0.000000           0         2           close
------ ----------- ----------- --------- --------- ----------------
100.00    0.000000                     7           total
```

Figure 7 - System call strace carried out on mmapdemo2.c

Bibliography

H, Uday. (2020). *Bash Scripting: Everything you need to know about Bash-shell programming, 2019.* Available at: https://medium.com/sysf/bash-scripting-everything-you-need-to-know-about-bash-shell-programming-cd08595f2fba (Accessed: 3 March 2021)

Z, Milly. (2020). *Cons and Pros of MMAP, 2016*. Available at: https://millyz.github.io/ta/os3150_2016/mem2-lab/mem2/part2_3.html (Accessed: 3 March 2021)

Repository

https://csgitlab.reading.ac.uk/at015244/cs2ao17.git

Appendices

Appendix A - Q6.c: Piping and processing

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

int main(void) {
  int pfds[2];
  int sndpfds[2];

  pipe(pfds);

  switch (fork()) {
    case -1:
    perror("Fork");
    exit(1);

    case 0:
    pipe(sndpfds);
      switch (fork()) {
      case -1:
      perror("Fork");
      exit(2);

      case 0:
      dup2(sndpfds[1], STDOUT_FILENO);
      close(sndpfds[0]);
      close(sndpfds[1]);
      execlp("/usr/bin/cat", "/usr/bin/cat", "/etc/group", NULL);
      exit(3);

      default:
      dup2(sndpfds[0], STDIN_FILENO);
      dup2(pfds[1], STDOUT_FILENO);
      close(sndpfds[0]);
      close(sndpfds[1]);
      execlp("/usr/bin/sort", "/usr/bin/sort", NULL);
      exit(3);
    }
    default:
    dup2(pfds[0], STDIN_FILENO);
    close(pfds[0]);
    close(pfds[1]);
    execlp("/usr/bin/cut", "/usr/bin/cut", "-f3", "-d:", NULL);
    exit(5);
  }
  return 0;
}
```

Appendix B - Q8.c: Pthread usage

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <math.h>
#include <pthread.h>

#define NUM_THREADS 5     /* define number of threads*/

float total=0;    /* Set global variable */

void *compute(void *threadid)     /* compute function just does something.
*/
{
    int i;
    float oldtotal=0, result=0;

    /* for a large number of times */
    for(i=0;i<2000000000;i++)
      result = sqrt(1000.0) * sqrt(1000.0);

    /* Print the result - should be no surprise */
    printf("Result is %f\n",result);

    /* to keep a running total in the global variable total */
    oldtotal = total;
    total = oldtotal + result;

    /* Print running total so far. */
    printf("Total is %f\n",total);

    pthread_exit(NULL);
} // code to be continued

int main()
{
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    float result=0;

    printf("\n");

    /* to loop and create the required number of processes */
    /* NOTE carefully how only the child process is left to run */
    for(t=0;t<NUM_THREADS;t++)
    {
```

```
          /* give a message about the thread ID */
            printf("IN MAIN: creating thread %ld\n", t);

             /* call the function to do some computation */
             rc = pthread_create(&threads[t], NULL, compute, (void *)t);
             if (rc){
               printf("ERROR: return code from pthread_create() is %d\n",
rc);
               exit(-1);
             }
             /* After computation, quit. OR process creation will bom! */
             break;
    }
    pthread_exit(NULL);
 }
```

Appendix C - Q9.c: Mutual Exclusions

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <math.h>
#include <pthread.h>
#define NUM_THREADS 5  /* define the total number of processes we want */

float total=0;   /* Set global variable */
pthread_mutex_t mutexsum;

void *compute(void *threadid)  /* compute function just does something. */
{
    int i;
    float oldtotal=0, result=0;

    /* for a large number of times */
    for(i=0;i<2000000000;i++)
      result = sqrt(1000.0) * sqrt(1000.0);

    /* Print the result - should be no surprise */
    printf("Result is %f\n",result);

    /* to keep a running total in the global variable total */
    pthread_mutex_lock (&mutexsum);
    oldtotal = total;
    total = oldtotal + result;
    pthread_mutex_unlock (&mutexsum);

    /* Print running total so far. */
```

```
    printf("Total is %f\n",total);

    pthread_exit((void*) 0);
} // code to be continued
int main()
{
    pthread_t threads[NUM_THREADS];
    pthread_attr_t attr;
    void *status;
    int rc;
    long t;

    printf("\n");

    pthread_mutex_init(&mutexsum, NULL);

    /* performing dot multiplication through the initialisation of the
       pthreads */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    /* to loop and create the required number of processes */
    /* NOTE carefully how only the child process is left to run */
    for(t=0;t<NUM_THREADS;t++)
    {
            /* give a message about the proc ID */
            printf("IN MAIN: creating thread %ld\n", t);

            /* call the function to do some computation */
            rc = pthread_create(&threads[t], &attr, compute, (void *)t);
            if (rc){
            printf("ERROR: return code from pthread_create() is %d\n", rc);
               exit(-1);
             }
    }
    pthread_attr_destroy(&attr);

    for(t=0; t<NUM_THREADS; t++)
    {
      pthread_join(threads[t], &status);
    }

    pthread_mutex_destroy(&mutexsum);
    pthread_exit(NULL);
 }
```

Appendix D - kirk2.c: Inter Process Communication

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct my_msgbuf {
    long mtype;
    char mtext[200];
};

int main(void)
{
    struct my_msgbuf buf;
    int msqid;
    key_t key;

    if ((key = ftok("kirk2.c", 'B')) == -1) {
        perror("ftok");
        exit(1);
    }

    if ((msqid = msgget(key, 0644 | IPC_CREAT)) == -1) {
        perror("msgget");
        exit(1);
    }

    printf("Enter lines of text, ^D to quit:\n");

    buf.mtype = 1; /* we don't really care in this case */

    while(fgets(buf.mtext, sizeof buf.mtext, stdin) != NULL) {
      if(*buf.mtext >= 'a' && *buf.mtext <= 'z'){
        int len = strlen(buf.mtext);

        /* ditch newline at end, if it exists */
        if (buf.mtext[len-1] == '\n') buf.mtext[len-1] = '\0';

        if (msgsnd(msqid, &buf, len+1, 0) == -1) /* +1 for '\0' */
            perror("msgsnd");
    } else {
      int len = strlen(buf.mtext);

      /* ditch newline at end, if it exists */
      if (buf.mtext[len-1] == '\n') buf.mtext[len-1] = '\0';
```

```
      if (msgsnd(msqid, &buf, len+1, 0) == 0) /* +1 for '\0' */
          perror("msgsnd");
    }
  }

    if (msgctl(msqid, IPC_RMID, NULL) == -1) {
        perror("msgctl");
        exit(1);
    }

    return 0;
}
```

Appendix E - spock2.c: Non-Urgent Receiver

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct my_msgbuf {
    long mtype;
    char mtext[200];
};

int main(void)
{
    struct my_msgbuf buf;
    int msqid;
    key_t key;

    if ((key = ftok("kirk2.c", 'B')) == -1) {  /* same key as kirk2.c */
        perror("ftok");
        exit(1);
    }

    if ((msqid = msgget(key, 0644)) == -1) { /* connect to the queue */
        perror("msgget");
        exit(1);
    }

    printf("spock2: ready to receive messages, captain.\n");

    for(;;) { /* Spock never quits! */
        if (msgrcv(msqid, &buf, sizeof buf.mtext, 0, 0) == -1) {
            perror("msgrcv");
```

```
            exit(1);
        }
        printf("kirk2: \"%s\"\n", buf.mtext);
    }

    return 0;
}
```

Appendix F - starfleet.c: Urgent Receiver

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct my_msgbuf {
    long mtype;
    char mtext[200];
};

int main(void)
{
    struct my_msgbuf buf;
    int msqid;
    key_t key;

    if ((key = ftok("kirk2.c", 'B')) == -1) {  /* same key as kirk2.c */
        perror("ftok");
        exit(1);
    }

    if ((msqid = msgget(key, 0644)) == -1) { /* connect to the queue */
        perror("msgget");
        exit(1);
    }

    printf("starfleet: ready to receive URGENT messages, captain.\n");

    for(;;) { /* Spock never quits! */
        if (msgrcv(msqid, &buf, sizeof buf.mtext, 0, 0) == 0) {
            perror("msgrcv");
            exit(1);
        }
        printf("kirk2: \"%s\"\n", buf.mtext);
    }
    return 0;
```

```
}
```

Appendix G - kirk3.c: Signal Handlers

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct my_msgbuf {
    long mtype;
    char mtext[200];
};

void sigint_handler(int sig)
{
  write(0, "\nSignal Terminated! Emergency transporter sent!\n", 47);
  write(0, "\n", 2);
}

int main(void)
{
    void sigint_handler(int sig);
    struct sigaction sa;

    sa.sa_handler = sigint_handler;
    sa.sa_flags = 0; // or SA_RESTART;
    sigemptyset(&sa.sa_mask);

    if (sigaction(SIGINT, &sa, NULL) == -1){
      perror("sigaction");
      exit(1);
    }

    struct my_msgbuf buf;
    int msqid;
    key_t key;

    if ((key = ftok("kirk2.c", 'B')) == -1) {
        perror("ftok");
        exit(1);
    }
```

```
    if ((msqid = msgget(key, 0644 | IPC_CREAT)) == -1) {
        perror("msgget");
        exit(1);
    }

    printf("Enter lines of text, ^D to quit:\n");

    buf.mtype = 1; /* we don't really care in this case */

    while(fgets(buf.mtext, sizeof buf.mtext, stdin) != NULL) {
      if(*buf.mtext >= 'a' && *buf.mtext <= 'z'){
        int len = strlen(buf.mtext);

        /* ditch newline at end, if it exists */
        if (buf.mtext[len-1] == '\n') buf.mtext[len-1] = '\0';

        if (msgsnd(msqid, &buf, len+1, 0) == -1) /* +1 for '\0' */
            perror("msgsnd");
    } else {
      int len = strlen(buf.mtext);

      /* ditch newline at end, if it exists */
      if (buf.mtext[len-1] == '\n') buf.mtext[len-1] = '\0';

      if (msgsnd(msqid, &buf, len+1, 0) == 0) /* +1 for '\0' */
          perror("msgsnd");
    }
  }

    if (msgctl(msqid, IPC_RMID, NULL) == -1) {
        perror("msgctl");
        exit(1);
    }

    return 0;
}
```

Appendix H - mmapdemo2.c: Larger Files being Mapped

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

```c
#define CHUNK_SIZE (1024*1024)

int main(int argc, char *argv[])
{
	int fd, offset, chomp, bit;
	char *data;
	struct stat sbuf;

	if (argc != 2) {
		fprintf(stderr, "usage: mmapdemo2 offset\n");
		exit(1);
	}

	if ((fd = open("mmapdemo2.c", O_RDONLY)) == -1) {
		perror("open");
		exit(1);
	}

	if (stat("mmapdemo2.c", &sbuf) == -1) {
		perror("stat");
		exit(1);
	}

	chomp = offset/CHUNK_SIZE;

	offset = atoi(argv[0]);
	if (offset < 0 || offset > sbuf.st_size-1) {
		fprintf(stderr, "mmapdemo2: offset must be in the range
0-%ld\n", sbuf.st_size-1);
		exit(1);
	}

	if ((data = mmap((caddr_t)0, sbuf.st_size, PROT_READ, MAP_SHARED, fd,
chomp*CHUNK_SIZE)) == (caddr_t)(0)) {
		perror("mmap");
		exit(1);
	}

	bit = offset % CHUNK_SIZE;

	printf("byte at offset %d is '%d'\n", offset, (int)data[bit]);

	return 0;
}
```