[Tutorial\_shell\_scripting](http://profile.iiita.ac.in/bibhas.ghoshal/lab_files/Tutorial_shell_scripting)

**Shell scripting**

Tutorial-0

Unix:UNIX is a popular multi-user, multitasking operating system (OS) developed at Bell Labs in the early 1970s.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

SHELL

Shell is a UNIX term for the interactive user interface with an operating system. The shell is the layer of programming that understands and executes the commands a user enters. In some systems, the shell is called a command interpreter. A shell usually implies an interface with a command syntax (think of the DOS operating system and its "C:>" prompts and user commands such as "dir" and "edit").

Shell Types

In Unix, there are two major types of shells −

Bourne shell − If you are using a Bourne-type shell, the $ character is the default prompt.

C shell − If you are using a C-type shell, the % character is the default prompt.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

SHELL SCRIPT

Usually shells are interactive that mean, they accept command as input from users and execute them. However some time we want to execute a bunch of commands routinely, so we have type in all commands each time in terminal.

As shell can also take commands as input from file we can write these commands in a file and can execute them in shell to avoid this repetitive work. These files are called Shell Scripts or Shell Programs. Shell scripts are similar to the batch file in MS-DOS. Each shell script is saved with .sh file extension eg. myscript.sh

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1) Function Call

#!/bin/bash

sum ()

{

x=$1

y=$2

k=0

k=$(( $x + $y ))

return $k

}

sum 7 5

echo "Sum of two no is:$?"

exit 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2) Command line argument

#!/bin/bash

x=$1

y=$2

k=$(( $x + $y ))

echo "sum is:$k"

exit 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

3) If-else(Conditional statement)

#!/bin/bash

big\_num()

{

x=$1

y=$2

z=$3

if [ $x -gt $y ] ; then

if [ $x -gt $z ] ; then

echo "Biggest no is:$x"

else

echo "Biggest no is:$z"

fi

else

if [ $y -gt $z ] ; then

echo "Biggest no is:$y"

else

echo "Biggest no is:$z"

fi

fi

}

big\_num 2 3 4

exit 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

4) while loop

#!/bin/bash

x=5

while [ $x -ge 1 ] ; do

echo $x

x=$(( $x - 1 ))

done

exit 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

5) For loop

#!/bin/bash

printf "enter the no which table u want "

read x

y=1

k=0

echo "Multiplication table of $x is as below"

for ((y=1;y<=10;y++)) do

k=$(( $x \* $y ))

echo "$x X $y=$k"

done

exit 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

6) Switch statement

#!/bin/bash

x=$1

case $x in

1)

echo "this is level 1"

;;

2)

echo "this is level 2"

;;

3)

echo "this is level 3"

;;

\*)

echo "this is my default level"

;;

esac

exit 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

7) Mathematical computation

#!/bin/bash

a=5.66

b=8.67

c=`echo $a + $b|bc`

echo $c

exit 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

8) List the content of current directory

#!/bin/bash

for x in `ls` ; do

echo $x

done

exit 0

[open\_read\_write\_with\_linux\_sys\_calls](http://profile.iiita.ac.in/bibhas.ghoshal/lab_files/io_system_calls1.c)

#include<stdio.h>

#include<fcntl.h>

int main(){

char buff[100],fn[10];

int fd,n;

printf("Enter the filename\n");

scanf("%s",fn);

fd = open(fn,O\_RDONLY);

n = read(fd,buff,100);

n = write(1,buff,n);

printf("%d\n",n);

close(fd);

}

[write\_lines\_of\_text\_sys\_call](http://profile.iiita.ac.in/bibhas.ghoshal/lab_files/io_system_calls2.c)

#include<stdio.h>

#include<fcntl.h>

#include<string.h>

int main(){

char \*buff,fn[10];

int fd,n,i;

printf("Enter the filename\n");

scanf("%s",fn);

printf("\nEnter text");

scanf("%s",buff);

fgets(buff,100,stdin);

fd = open(fn,O\_CREAT|O\_WRONLY);

n = write(fd,buff,strlen(buff));

close(fd);

}

[simulating "ls" command](http://profile.iiita.ac.in/bibhas.ghoshal/lab_files/simulating_ls.c)

#include<stdio.h>

#include<dirent.h>

int main(){

char d[10];

DIR \*p;

struct dirent \*d1;

printf("Enter the directory name\n");

scanf("%s",d);

p=opendir(d);

if(p==NULL){

perror("can't find directory");

exit(-1);

}

while(d1=readdir(p))

printf("%s\n",d1->d\_name);

}

[fork\_execute.c](http://profile.iiita.ac.in/bibhas.ghoshal/lab_assignments/sample_programs_process_mgmt/fork_execute.c)   [fork\_test.c](http://profile.iiita.ac.in/bibhas.ghoshal/lab_assignments/sample_programs_process_mgmt/fork_test.c)   [fork-01.c](http://profile.iiita.ac.in/bibhas.ghoshal/lab_assignments/sample_programs_process_mgmt/fork-01.c)  
[fork-02.c](http://profile.iiita.ac.in/bibhas.ghoshal/lab_assignments/sample_programs_process_mgmt/fork-02.c)   [fork-03.c](http://profile.iiita.ac.in/bibhas.ghoshal/lab_assignments/sample_programs_process_mgmt/fork-03.c)   [fork-04.c](http://profile.iiita.ac.in/bibhas.ghoshal/lab_assignments/sample_programs_process_mgmt/fork-04.c)  [fork-05.c](http://profile.iiita.ac.in/bibhas.ghoshal/lab_assignments/sample_programs_process_mgmt/fork-05.c)   
[orphan\_process.c](http://profile.iiita.ac.in/bibhas.ghoshal/lab_assignments/sample_programs_process_mgmt/orphan_process.c) [zombie\_process.c](http://profile.iiita.ac.in/bibhas.ghoshal/lab_assignments/sample_programs_process_mgmt/zombie_process.c)

#include<stdio.h>

#include<sys/types.h>

#include<sys/wait.h>

#include<stdlib.h>

#include<unistd.h>

int main(){

int pid;

int status;

pid =fork();

if(pid>0){

pid=wait(&status);

}

else{

execlp("ls","",NULL);

exit(0);

}

}

#include<stdio.h>

#include<sys/types.h>

#include<sys/wait.h>

#include<stdlib.h>

int main(){

int frv,pid;

int status;

frv =fork();

if(frv>0){

printf("I am in the parent, my process id = %d, my parent's process id = %d, the child's process id = %d and return value of the fork = %d\n",getpid(),getppid(),frv,frv);

pid = wait(&status);

printf("I am in the parent after the child exited and the exited child's process id = %d \n ",pid );

}

else{

printf("I am in the child, my process id = %d, and my parents's process id = %d and return value of the fork = %d\n",getpid(),getppid(),frv);

exit(0);

}

}

/\* ----------------------------------------------------------------- \*/

/\* PROGRAM fork-01.c \*/

/\* This program illustrates the use of fork() and getpid() system \*/

/\* calls. Note that write() is used instead of printf() since the \*/

/\* latter is buffered while the former is not. \*/

/\* ----------------------------------------------------------------- \*/

#include <stdio.h>

#include <string.h>

#include <sys/types.h>

#define MAX\_COUNT 200

#define BUF\_SIZE 100

void main(void)

{

pid\_t pid;

int i;

char buf[BUF\_SIZE];

fork();

pid = getpid();

for (i = 1; i <= MAX\_COUNT; i++) {

sprintf(buf, "This line is from pid %d, value = %d\n", pid, i);

write(1, buf, strlen(buf));

// printf("%s",buf);

}

}

/\* ----------------------------------------------------------------- \*/

/\* PROGRAM fork-02.c \*/

/\* This program runs two processes, a parent and a child. Both of \*/

/\* them run the same loop printing some messages. Note that printf()\*/

/\* is used in this program. \*/

/\* ----------------------------------------------------------------- \*/

#include <stdio.h>

#include <sys/types.h>

#define MAX\_COUNT 200

void ChildProcess(void); /\* child process prototype \*/

void ParentProcess(void); /\* parent process prototype \*/

void main(void)

{

pid\_t pid;

pid = fork();

if (pid == 0)

ChildProcess();

else

ParentProcess();

}

void ChildProcess(void)

{

int i;

for (i = 1; i <= MAX\_COUNT; i++)

printf(" This line is from child, value = %d\n", i);

printf(" \*\*\* Child process is done \*\*\*\n");

}

void ParentProcess(void)

{

int i;

for (i = 1; i <= MAX\_COUNT; i++)

printf("This line is from parent, value = %d\n", i);

printf("\*\*\* Parent is done \*\*\*\n");

}

/\* ----------------------------------------------------------------- \*/

/\* PROGRAM fork-03.c \*/

/\* This program creates two child processes displaying some lines.\*/

/\* The parent waits until all child processes are done and exits. \*/

/\* ----------------------------------------------------------------- \*/

#include <stdio.h>

#include <string.h>

#include <sys/types.h>

#define MAX\_COUNT 200

#define BUF\_SIZE 100

void ChildProcess(char [], char []); /\* child process prototype \*/

void main(void)

{

pid\_t pid1, pid2, pid;

int status;

int i;

char buf[BUF\_SIZE];

printf("\*\*\* Parent is about to fork process 1 \*\*\*\n");

if ((pid1 = fork()) < 0) {

printf("Failed to fork process 1\n");

exit(1);

}

else if (pid1 == 0)

ChildProcess("First", " ");

printf("\*\*\* Parent is about to fork process 2 \*\*\*\n");

if ((pid2 = fork()) < 0) {

printf("Failed to fork process 2\n");

exit(1);

}

else if (pid2 == 0)

ChildProcess("Second", " ");

sprintf(buf, "\*\*\* Parent enters waiting status .....\n");

write(1, buf, strlen(buf));

pid = wait(&status);

sprintf(buf, "\*\*\* Parent detects process %d was done \*\*\*\n", pid);

write(1, buf, strlen(buf));

pid = wait(&status);

printf("\*\*\* Parent detects process %d is done \*\*\*\n", pid);

printf("\*\*\* Parent exits \*\*\*\n");

exit(0);

}

void ChildProcess(char \*number, char \*space)

{

pid\_t pid;

int i;

char buf[BUF\_SIZE];

pid = getpid();

sprintf(buf, "%s%s child process starts (pid = %d)\n",

space, number, pid);

write(1, buf, strlen(buf));

for (i = 1; i <= MAX\_COUNT; i++) {

sprintf(buf, "%s%s child's output, value = %d\n", space, number, i);

write(1, buf, strlen(buf));

}

sprintf(buf, "%s%s child (pid = %d) is about to exit\n",

space, number, pid);

write(1, buf, strlen(buf));

exit(0);

}

/\* ----------------------------------------------------------------- \*/

/\* PROGRAM fork-04.c \*/

/\* This one is identical to fork-03.c, except that the parent \*/

/\* process also does some computation. \*/

/\* ----------------------------------------------------------------- \*/

#include <stdio.h>

#include <string.h>

#include <sys/types.h>

#define MAX\_COUNT 200

#define BUF\_SIZE 100

void ChildProcess(char [], char []); /\* child process prototype \*/

void ParentProcess(void); /\* parent process prototype \*/

void main(void)

{

pid\_t pid1, pid2, pid;

int status;

int i;

char buf[BUF\_SIZE];

printf("\*\*\* Parent is about to fork process 1 \*\*\*\n");

if ((pid1 = fork()) < 0) {

printf("Failed to fork process 1\n");

exit(1);

}

else if (pid1 == 0)

ChildProcess("First", " ");

printf("\*\*\* Parent is about to fork process 2 \*\*\*\n");

if ((pid2 = fork()) < 0) {

printf("Failed to fork process 2\n");

exit(1);

}

else if (pid2 == 0)

ChildProcess("Second", " ");

ParentProcess();

sprintf(buf, "\*\*\* Parent enters waiting status .....\n");

write(1, buf, strlen(buf));

pid = wait(&status);

sprintf(buf, "\*\*\* Parent detects process %d was done \*\*\*\n", pid);

write(1, buf, strlen(buf));

pid = wait(&status);

printf("\*\*\* Parent detects process %d is done \*\*\*\n", pid);

printf("\*\*\* Parent exits \*\*\*\n");

exit(0);

}

/\* ----------------------------------------------------------------- \*/

/\* FUNCTION ParentProcess : \*/

/\* This function computes and displays all Armstrong numbers in \*/

/\* the range of 0 and 9999. A four digit number is an Armstrong \*/

/\* number if and only if the sum of individual digits raised to the \*/

/\* fourth power is equal to the number itself. \*/

/\* ----------------------------------------------------------------- \*/

#define QUAD(x) (x\*x\*x\*x)

void ParentProcess(void)

{

int a, b, c, d;

int abcd, a4b4c4d4;

int count = 0;

char buf[BUF\_SIZE];

sprintf(buf, "Parent is about to compute the Armstrong numbers\n");

write(1, buf, strlen(buf));

for (a = 0; a <= 9; a++)

for (b = 0; b <= 9; b++)

for (c = 0; c <= 9; c++)

for (d = 0; d <= 9; d++) {

abcd = a\*1000 + b\*100 + c\*10 + d;

a4b4c4d4 = QUAD(a) + QUAD(b) + QUAD(c) + QUAD(d);

if (abcd == a4b4c4d4) {

sprintf(buf, "From parent: "

"the %d Armstrong number is %d\n",

++count, abcd);

write(1, buf, strlen(buf));

}

}

sprintf(buf, "From parent: there are %d Armstrong numbers\n", count);

write(1, buf, strlen(buf));

}

void ChildProcess(char \*number, char \*space)

{

pid\_t pid;

int i;

char buf[BUF\_SIZE];

pid = getpid();

sprintf(buf, "%s%s child process starts (pid = %d)\n",

space, number, pid);

write(1, buf, strlen(buf));

for (i = 1; i <= MAX\_COUNT; i++) {

sprintf(buf, "%s%s child's output, value = %d\n",

space, number, i);

write(1, buf, strlen(buf));

}

sprintf(buf, "%s%s child (pid = %d) is about to exit\n",

space, number, pid);

write(1, buf, strlen(buf));

exit(0);

/\* ---------------------------------------------------------------- \*/

/\* PROGRAM fork-05.c \*/

/\* This is a modification to fork-04.c. Now wait() is treated \*/

/\* as a procedure, since the returned process ID is unimportant to \*/

/\* this program. \*/

/\* ---------------------------------------------------------------- \*/

#include <stdio.h>

#include <string.h>

#include <sys/types.h>

#define MAX\_COUNT 100

#define BUF\_SIZE 100

void ChildProcess(char [], char []);

void main(void)

{

pid\_t pid1, pid2, pid;

int status;

int i;

char buf[BUF\_SIZE];

printf("\*\*\* Parent is about to fork process 1 \*\*\*\n");

if ((pid1 = fork()) < 0) {

printf("Failed to fork process 1\n");

exit(1);

}

else if (pid1 == 0)

ChildProcess("First", " ");

printf("\*\*\* Parent is about to fork process 2 \*\*\*\n");

if ((pid2 = fork()) < 0) {

printf("Failed to fork process 2\n");

exit(1);

}

else if (pid2 == 0)

ChildProcess("Second", " ");

sprintf(buf, "\*\*\* Parent enters waiting status .....\n");

write(1, buf, strlen(buf));

wait(&status);

sprintf(buf, "\*\*\* Parent detects a child process was done \*\*\*\n");

write(1, buf, strlen(buf));

wait(&status);

printf("\*\*\* Parent detects another child process was done \*\*\*\n");

printf("\*\*\* Parent exits \*\*\*\n");

exit(0);

}

void ChildProcess(char \*number, char \*space)

{

pid\_t pid;

int i;

char buf[BUF\_SIZE];

pid = getpid();

sprintf(buf, "%s%s child process starts (pid = %d)\n",

space, number, pid);

write(1, buf, strlen(buf));

for (i = 1; i <= MAX\_COUNT; i++) {

sprintf(buf, "%s%s child's output, value = %d\n",

space, number, i);

write(1, buf, strlen(buf));

}

sprintf(buf, "%s%s child (pid = %d) is about to exit\n",

space, number, pid);

write(1, buf, strlen(buf));

exit(0);

}

#include <sys/wait.h>

#include<stdio.h>

void ChildProc(void)

{

int i;

for(i = 0; i < 100; i++)

{

printf("Child : %d from process <%d> : Parent <%d>\n",i, getpid(), getppid());

}

}

void ParentProc(void)

{

int i;

for(i = 0; i < 20; i++)

{

printf("Parent : %d from process <%d> : Parent <%d>\n",i, getpid(), getppid());

}

}

int main(void)

{

pid\_t child;

child = fork();

if(child == 0)

{

ChildProc();

}

else

{

ParentProc();

//waitpid(child,NULL,0);

}

}

[thread\_example.c](http://profile.iiita.ac.in/bibhas.ghoshal/lab_assignments/thread_example.c)  [print-rand-matrix.c](http://profile.iiita.ac.in/bibhas.ghoshal/lab_assignments/print-rand-matrix.c)  [c-timer.c](http://profile.iiita.ac.in/bibhas.ghoshal/lab_assignments/c-timer.c)

#include<stdio.h>

#include<pthread.h>

void \*thread\_fn(void \*arg){

long id =(long) arg;

printf("Starting thread %ld\n", id);

sleep(5);

printf("Exiting thread %ld\n", id);

return NULL;

}

int main(){

pthread\_t t1,t2;

pthread\_create(&t1,NULL,thread\_fn,(void \*)1);

pthread\_create(&t2,NULL,thread\_fn,(void \*)2);

pthread\_join(t1,NULL);

printf("Exiting main\n");

return 0;

}

#include <stdlib.h>

#include <unistd.h>

#include <stdio.h>

#include <math.h>

#include <sys/time.h>

char \*Usage = "print-rand-matrix -r rows -c cols\n";

#define ARGS "r:c:"

int Rows;

int Cols;

int main(int argc, char \*\*argv)

{

int c;

int i;

int j;

double r;

struct timeval tm;

unsigned long seed;

while((c = getopt(argc,argv,ARGS)) != EOF) {

switch(c) {

case 'r':

Rows = atoi(optarg);

break;

case 'c':

Cols = atoi(optarg);

break;

default:

fprintf(stderr,"unrecognized command %c\n",

(char)c);

fprintf(stderr,"usage: %s",Usage);

exit(1);

}

}

if(Rows <= 0) {

fprintf(stderr,"must enter rows\n");

fprintf(stderr,"usage: %s",Usage);

exit(1);

}

if(Cols <= 0) {

fprintf(stderr,"must enter columns\n");

fprintf(stderr,"usage: %s",Usage);

exit(1);

}

gettimeofday(&tm,NULL);

seed = tm.tv\_sec + tm.tv\_usec;

srand48(seed);

printf("%d %d\n",Rows,Cols);

for(i=0; i < Rows; i++) {

printf("# Row %d\n",i);

for(j=0; j < Cols; j++) {

r = drand48();

printf("%f\n",r);

}

}

exit(0);

}

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <sys/time.h>

double CTimer() {

struct timeval tm;

gettimeofday(&tm,NULL);

return((double)tm.tv\_sec + (double)(tm.tv\_usec/1000000.0));

}

[Lecture notes on IPC](http://profile.iiita.ac.in/bibhas.ghoshal/lab_assignments/lecture_notes_IPC_mechanism.pdf)    
[shared memory](http://profile.iiita.ac.in/bibhas.ghoshal/lab_assignments/shm.c)    
[pipe](http://profile.iiita.ac.in/bibhas.ghoshal/lab_assignments/pipe.c)  [pipe\_example.c](http://profile.iiita.ac.in/bibhas.ghoshal/lab_assignments/pipe_example.c)    
[read-write1.c](http://profile.iiita.ac.in/bibhas.ghoshal/lab_assignments/read-write-1.c)  [read-write2.c](http://profile.iiita.ac.in/bibhas.ghoshal/lab_assignments/read-write-2.c)

/\*

SHARING MEMORY BETWEEN PROCESSES

In this example, we show how two processes can share a common

portion of the memory. Recall that when a process forks, the

new child process has an identical copy of the variables of

the parent process. After fork the parent and child can update

their own copies of the variables in their own way, since they

dont actually share the variable. Here we show how they can

share memory, so that when one updates it, the other can see

the change.

\*/

#include <stdio.h>

#include <sys/ipc.h>

#include <sys/shm.h> /\* This file is necessary for using shared

memory constructs

\*/

main()

{

int shmid,status;

int \*a, \*b;

int i;

/\*

The operating system keeps track of the set of shared memory

segments. In order to acquire shared memory, we must first

request the shared memory from the OS using the shmget()

system call. The second parameter specifies the number of

bytes of memory requested. shmget() returns a shared memory

identifier (SHMID) which is an integer. Refer to the online

man pages for details on the other two parameters of shmget()

\*/

shmid = shmget(IPC\_PRIVATE, 2\*sizeof(int), 0777|IPC\_CREAT);

/\* We request an array of two integers \*/

/\*

After forking, the parent and child must "attach" the shared

memory to its local data segment. This is done by the shmat()

system call. shmat() takes the SHMID of the shared memory

segment as input parameter and returns the address at which

the segment has been attached. Thus shmat() returns a char

pointer.

\*/

if (fork() == 0) {

/\* Child Process \*/

/\* shmat() returns a char pointer which is typecast here

to int and the address is stored in the int pointer b. \*/

b = (int \*) shmat(shmid, 0, 0);

for( i=0; i< 10; i++) {

sleep(5);

printf("\t\t\t Child reads: %d,%d\n",b[0],b[1]);

}

/\* each process should "detach" itself from the

shared memory after it is used \*/

shmdt(b);

}

else {

/\* Parent Process \*/

/\* shmat() returns a char pointer which is typecast here

to int and the address is stored in the int pointer a.

Thus the memory locations a[0] and a[1] of the parent

are the same as the memory locations b[0] and b[1] of

the parent, since the memory is shared.

\*/

a = (int \*) shmat(shmid, 0, 0);

a[0] = 0; a[1] = 1;

for( i=0; i< 10; i++) {

sleep(1);

a[0] = a[0] + a[1];

a[1] = a[0] + a[1];

printf("Parent writes: %d,%d\n",a[0],a[1]);

}

wait(&status);

/\* each process should "detach" itself from the

shared memory after it is used \*/

shmdt(a);

/\* Child has exited, so parent process should delete

the cretaed shared memory. Unlike attach and detach,

which is to be done for each process separately,

deleting the shared memory has to be done by only

one process after making sure that noone else

will be using it

\*/

shmctl(shmid, IPC\_RMID, 0);

}

}

/\*

POINTS TO NOTE:

In this case we find that the child reads all the values written

by the parent. Also the child does not print the same values

again.

1. Modify the sleep in the child process to sleep(2). What

happens now?

2. Restore the sleep in the child process to sleep(1) and modify

the sleep in the parent process to sleep(2). What happens now?

Thus we see that when the writer is faster than the reader, then

the reader may miss some of the values written into the shared

memory. Similarly, when the reader is faster than the writer, then

the reader may read the same values more than once. Perfect

inter-process communication requires synchronization between the

reader and the writer. You can use semaphores to do this.

Further note that "sleep" is not a synchronization construct.

We use "sleep" to model some amount of computation which may

exist in the process in a real world application.

Also, we have called the different shared memory related

functions such as shmget, shmat, shmdt, and shmctl, assuming

that they always succeed and never fail. This is done to

keep this proram simple. In practice, you should always check for

the return values from this function and exit if there is

an error.

\*/

/\*

COMMUNICATING VIA A PIPE

Consider an application that requires one process to write a set of

values, that are to be read by the other. Pipe is a system

construct which facilitates such communication. A pipe is also a

shared buffer. When a process tries to read from an empty pipe, it

waits until someone has written something into the pipe. When the

pipe is full, any process attempting to write into the pipe is made

to wait.

\*/

#include <stdio.h>

#include <unistd.h> /\* Include this file to use pipes \*/

#define BUFSIZE 80 /\* We will write lines of 80 chars into the pipe.

The pipe has a large capacity and can accomodate

many such lines.

\*/

main()

{

int fd[2];

int n=0;

int i;

char line[BUFSIZE];

/\* A pipe is treated as a file by the system. You must have used

fopen() to open a file. fopen() returns a "file pointer" which

is used in fprintf(), fscanf(), fclose() etc. However, when we

wish to perform reads and writes in blocks from a file, we use

the system call "open" to open a file. Internally files are

always opened using the "open" call. For each process the system

maintains a "file descriptor table" (FDT) containing an entry

for each file opened by that process. When a new file is opened,

a new entry is created in the FDT, and the entry number is

returned as an integer called "file descriptor".

Unlike in a file, we may want to both read and write from a

pipe at the same time. Hence when a pipe is created, two file

descriptors are created -- one for reading the pipe and one for

writing into the pipe. The pipe() system call requires an

array of two integers as parameter. The system returns the file

descriptors through this array.

\*/

pipe(fd); /\* fd[0] is for reading,

fd[1] is for writing

\*/

/\* To illustrate the working of the pipe, we will make the child

process write the integer n into the pipe and make the parent

to read from the pipe. We put sleep in the writer process

(in this case the child) to show that the reader process waits

for the writer to write into the pipe.

To write a block of bytes into a pipe (or more generally into

a file) the write() system call is used. Similarly read() is

used to read a block of bytes from a file. Refer to the online

man pages for these calls.

\*/

if (fork() == 0) {

close(fd[0]); /\* The child will not read and

hence we close fd[0]

\*/

for (i=0; i < 10; i++) {

sprintf(line,"%d",n); /\* Since write() accepts only

arrays of bytes, we

first write the integer n

into the char array "line"

\*/

write(fd[1], line, BUFSIZE);

printf("Child writes: %d\n",n);

n++;

sleep(2);

}

}

else {

close(fd[1]); /\* The parent will not write and

hence we close fd[1]

\*/

for (i=0; i < 10; i++) {

printf("\t\t\t Parent trying to read pipe\n");

read(fd[0], line, BUFSIZE);

sscanf(line,"%d",&n); /\* Read the integer from the

line of characters read

from the pipe

\*/

printf("\t\t\t Parent reads: %d\n",n);

}

}

}

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

int main(){

int pipefd[2];

int pid;

char recv[32];

pipe(pipefd);

switch(pid=fork()){

case -1:perror("fork");

exit(1);

case 0: /\* in child process \*/

close(pipefd[0]); /\* close unnecessary pipefd \*/

FILE \*out = fdopen(pipefd[1],"w"); /\* open file descriptor as a stream \*/

fprintf(out,"Hello World\n"); /\* write to out stream \*/

break;

default: /\* in parent process \*/

close(pipefd[1]); /\* close unnecessary pipefd \*/

FILE \*in = fdopen(pipefd[0],"r"); /\* open file descriptor as a stream \*/

fscanf(in,"%s",recv); /\* read from in stream \*/

printf("%s",recv);

break;

}

}

/\* A simple readers/writers program using a one-word shared memory. \*/

#include <sys/types.h>

#include <unistd.h>

#include <stdio.h>

#include <sys/mman.h>

#define SIZE sizeof(int) /\* size of [int] integer \*/

#define run\_length 10 /\* number of iterations in test run \*/

int main (void)

{ pid\_t pid; /\* variable to record process id of child \*/

caddr\_t shared\_memory; /\* shared memory base address \*/

int i\_parent, i\_child; /\* index variables \*/

int value; /\* value read by child \*/

/\* set up shared memory segment \*/

shared\_memory=mmap(0, SIZE, PROT\_READ | PROT\_WRITE,

MAP\_ANONYMOUS | MAP\_SHARED, -1, 0);

if (shared\_memory == (caddr\_t) -1)

{ perror ("error in mmap while allocating shared memory\n");

exit (1);

}

if ((pid = fork()) < 0) /\* apply fork and check for error \*/

{ perror ("error in fork");

exit (1);

}

if (0 == pid)

{ /\* processing for child \*/

printf ("The child process begins.\n");

for (i\_child = 0; i\_child < run\_length; i\_child++)

{ sleep(1); /\* wait for memory to be updated \*/

value = \*shared\_memory;

printf ("Child's report: current value = %2d\n", value);

}

printf ("The child is done\n");

}

else

{ /\* processing for parent \*/

printf ("The parent process begins.\n");

for (i\_parent = 0; i\_parent < run\_length; i\_parent++)

{ \*shared\_memory = i\_parent \* i\_parent;/\* square into shared memory \*/

printf ("Parent's report: current index = %2d\n", i\_parent );

sleep(1); /\* wait for child to read value \*/

}

wait(pid);

printf ("The parent is done\n");

}

exit (0);

}

/\* A simple readers/writers program using a shared buffer and spinlocks \*/

#include <sys/types.h>

#include <unistd.h>

#include <stdio.h>

#include <sys/mman.h>

#define BUF\_SIZE 5 /\* logical size of buffer \*/

#define SHARED\_MEM\_SIZE (BUF\_SIZE+2)\*sizeof(int) /\* size of shared memory \*/

#define run\_length 10 /\* number of iterations in test run \*/

int main (void)

{ pid\_t pid; /\* variable to record process id of child \*/

caddr\_t shared\_memory; /\* shared memory base address \*/

int \*in; /\* pointer to logical 'in' address for writer \*/

int \*out; /\* pointer to logical 'out' address for reader \*/

int \*buffer; /\* logical base address for buffer \*/

int i\_child, j\_child; /\* index variables \*/

int value; /\* value read by child \*/

/\* set up shared memory segment \*/

shared\_memory=mmap(0, SHARED\_MEM\_SIZE, PROT\_READ | PROT\_WRITE,

MAP\_ANONYMOUS | MAP\_SHARED, -1, 0);

if (shared\_memory == (caddr\_t) -1)

{ perror ("error in mmap while allocating shared memory\n");

exit (1);

}

/\* shared memory segment will be organized as follows:

0 n-1 n n+1

---------------------------------------------------------------

| | | |

---------------------------------------------------------------

^ ^ ^

buffer in out

\*/

buffer = (int\*) shared\_memory; /\* logical buffer starts at shared segment \*/

in = (int\*) shared\_memory + BUF\_SIZE\*sizeof(int);

out = (int\*) shared\_memory + (BUF\_SIZE+1)\*sizeof(int);

\*in = \*out = 0; /\* initial starting points \*/

if (-1 == (pid = fork())) /\* check for error in spawning child process \*/

{ perror ("error in fork");

exit (1);

}

if (0 == pid)

{ /\* processing for child == reader \*/

printf ("The reader process begins.\n");

for (i\_child = 0; i\_child < run\_length; i\_child++)

{ while (\*in == \*out) ; /\* spinlock waiting for data \*/

value = buffer[\*out];

\*out = (\*out + 1) % BUF\_SIZE;

printf ("Reader's report: item %2d == %2d\n", i\_child, value);

}

printf ("Reader done.\n");

}

else

{ /\* processing for parent == writer \*/

printf ("The writer process begins.\n");

for (j\_child = 0; j\_child < run\_length; j\_child++)

{ while ((\*in + 1) % BUF\_SIZE == \*out);/\* spinlock waiting for space \*/

buffer[\*in] = j\_child\*j\_child; /\* put data in buffer \*/

\*in = (\*in + 1) % BUF\_SIZE;

printf ("Writer's report: item %2d put in buffer\n", j\_child);

}

wait (pid);

printf ("Writer done.\n");

}

exit (0);

}

[Synchronizing Threads](http://www.csc.villanova.edu/~mdamian/threads/posixsem.html#exercise_1) [read-write3.c](http://profile.iiita.ac.in/bibhas.ghoshal/lab_assignments/read-write-3.c)  [read-write4.c](http://profile.iiita.ac.in/bibhas.ghoshal/lab_assignments/read-write-4.c)

**Synchronizing Threads with POSIX Semaphores**

1. Why semaphores?
2. Posix semaphores are easy to use
   * [sem\_init](http://www.csc.villanova.edu/~mdamian/threads/posixsem.html#init)
   * [sem\_wait](http://www.csc.villanova.edu/~mdamian/threads/posixsem.html#wait)
   * [sem\_post](http://www.csc.villanova.edu/~mdamian/threads/posixsem.html#post)
   * [sem\_getvalue](http://www.csc.villanova.edu/~mdamian/threads/posixsem.html#getvalue)
   * [sem\_destroy](http://www.csc.villanova.edu/~mdamian/threads/posixsem.html#destroy)
3. [Activities](http://www.csc.villanova.edu/~mdamian/threads/posixsem.html#exercise_1)   [1](http://www.csc.villanova.edu/~mdamian/threads/posixsem.html#exercise_1)   [2](http://www.csc.villanova.edu/~mdamian/threads/posixsem.html#exercise_2)

Now it is time to take a look at some code that does something a little unexpected. The program [badcnt.c](http://www.csc.villanova.edu/~mdamian/threads/badcnt.txt) creates two new threads, both of which increment a global variable called cnt exactly NITER, withNITER = 1,000,000. But the program produces unexpected results.

Activity 1. Create a directory called posixsem in your class Unix directory. Download in this directory the code [badcnt.c](http://www.csc.villanova.edu/~mdamian/threads/badcnt.txt) and compile it using

gcc badcnt.c -o xbadcnt -lpthread

Run the executable badcnt and observe the ouput.

Quite unexpected! Since cnt starts at 0, and both threads increment it NITER times, we should see cnt equal to 2\*NITER at the end of the program. What happens?

Threads can greatly simplify writing elegant and efficient programs. However, there are problems when multiple threads share a common address space, like the variable cnt in our earlier example.

To understand what might happen, let us analyze this simple piece of code:

THREAD 1 THREAD 2

a = data; b = data;

a++; b--;

data = a; data = b;

Now if this code is executed serially (for instance, THREAD 1 first and then THREAD 2), there are no problems. However threads execute in an arbitrary order, so consider the following situation:

|  |  |  |
| --- | --- | --- |
| **Thread 1** | **Thread 2** | **data** |
| a = data; | --- | 0 |
| a = a+1; | --- | 0 |
| --- | b = data;  // 0 | 0 |
| --- | b = b - 1; | 0 |
| data = a;  // 1 | --- | 1 |
| --- | data = b;  // 1 | 1 |

So data could end up +1, 0, -1, and there is **NO WAY** to know which value! It is completely non-deterministic!

The solution to this is to provide functions that will block a thread if another thread is accessing data that it is using.

Pthreads may use semaphores to achieve this.

**Posix semaphores**

All POSIX semaphore functions and types are prototyped or defined in semaphore.h. To define a semaphore object, use

sem\_t *sem\_name*;

To initialize a semaphore, use [sem\_init](http://pubs.opengroup.org/onlinepubs/009695399/functions/sem_init.html):

int [sem\_init](http://pubs.opengroup.org/onlinepubs/009695399/functions/sem_init.html)(sem\_t \*sem, int pshared, unsigned int value);

* sem points to a semaphore object to initialize
* pshared is a flag indicating whether or not the semaphore should be shared with fork()ed processes. LinuxThreads does not currently support shared semaphores
* value is an initial value to set the semaphore to

Example of use:

sem\_init(&sem\_name, 0, 10);

To wait on a semaphore, use [sem\_wait](http://pubs.opengroup.org/onlinepubs/009695399/functions/sem_wait.html):

int [sem\_wait](http://pubs.opengroup.org/onlinepubs/009695399/functions/sem_wait.html)(sem\_t \*sem);

Example of use:

sem\_wait(&sem\_name);

* If the value of the semaphore is negative, the calling process blocks; one of the blocked processes wakes up when another process calls sem\_post.

To increment the value of a semaphore, use [sem\_post](http://pubs.opengroup.org/onlinepubs/009695399/functions/sem_post.html):

int [sem\_post](http://pubs.opengroup.org/onlinepubs/009695399/functions/sem_post.html)(sem\_t \*sem);

Example of use:

sem\_post(&sem\_name);

* It increments the value of the semaphore and wakes up a blocked process waiting on the semaphore, if any.

To find out the value of a semaphore, use

int [sem\_getvalue](http://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_getvalue.html)(sem\_t \*sem, int \*valp);

* gets the current value of sem and places it in the location pointed to by valp

Example of use:

int value;

sem\_getvalue(&sem\_name, &value);

printf("The value of the semaphors is %d\n", value);

To destroy a semaphore, use

int [sem\_destroy](http://pubs.opengroup.org/onlinepubs/7908799/xsh/sem_destroy.html)(sem\_t \*sem);

* destroys the semaphore; no threads should be waiting on the semaphore if its destruction is to succeed.

Example of use:

sem\_destroy(&sem\_name);

**Using semaphores - a short example**

Consider the problem we had before and now let us use semaphores:

Declare the semaphore global (outside of any funcion):

sem\_t mutex;

Initialize the semaphore in the main function:

sem\_init(&mutex, 0, 1);

|  |  |  |
| --- | --- | --- |
| **Thread 1** | **Thread 2** | **data** |
| sem\_wait (&mutex); | --- | 0 |
| --- | sem\_wait (&mutex); | 0 |
| a = data; | /\* blocked \*/ | 0 |
| a = a+1; | /\* blocked \*/ | 0 |
| data = a; | /\* blocked \*/ | 1 |
| sem\_post (&mutex); | /\* blocked \*/ | 1 |
| /\* blocked \*/ | b = data; | 1 |
| /\* blocked \*/ | b = b - 1; | 1 |
| /\* blocked \*/ | data = b; | 2 |
| /\* blocked \*/ | sem\_post (&mutex); | 2 |
| **[data is fine. The data race is gone.]** | | |

Activity 2. Use the example above as a guide to fix the program [badcnt.c](http://www.csc.villanova.edu/~mdamian/threads/badcnt.c), so that the program always produces the expected output (the value 2\*NITER). Make a copy of badcnt.c into goodcnt.c before you modify the code.

To compile a program that uses pthreads *and* posix semaphores, use

gcc -o xfilename filename.c -lpthread -lrt

/\* A readers/writers program using a shared buffer and semaphores \*/

#include <sys/types.h>

#include <unistd.h>

#include <stdio.h>

#include <sys/mman.h>

#include <sys/sem.h>

#define BUF\_SIZE 5 /\* logical size of buffer \*/

#define SHARED\_MEM\_SIZE (BUF\_SIZE+2)\*sizeof(int) /\* size of shared memory \*/

#define run\_length 10 /\* number of iterations in test run \*/

#define buf\_used 0 /\* semaphore array index to check buffer elts used \*/

#define buf\_space 1 /\* semaphore array index to check buffer elts empty \*/

int sem\_init(void)

{ /\* procedure to create and initialize semaphores and return semaphore id,

assuming two semaphores defined in the given array of semaphores \*/

int semid;

/\* create new semaphore set of 2 semaphores \*/

if ((semid = semget (IPC\_PRIVATE, 2, IPC\_CREAT | 0600)) < 0)

{ perror ("error in creating semaphore");/\* 0600 = read/alter by user \*/

exit (1);

}

/\* initialization of semaphores \*/

/\* BUF\_SIZE free spaces in empty buffer \*/

if (semctl (semid, buf\_space, SETVAL, BUF\_SIZE) < 0)

{ perror ("error in initializing first semaphore");

exit (1);

}

/\* 0 items in empty buffer \*/

if (semctl (semid, buf\_used, SETVAL, 0) < 0)

{ perror ("error in initializing second semaphore");

exit (1);

}

return semid;

}

void P(int semid, int index)

{/\* procedure to perform a P or wait operation on a semaphore of given index \*/

struct sembuf sops[1]; /\* only one semaphore operation to be executed \*/

sops[0].sem\_num = index;/\* define operation on semaphore with given index \*/

sops[0].sem\_op = -1; /\* subtract 1 to value for P operation \*/

sops[0].sem\_flg = 0; /\* type "man semop" in shell window for details \*/

if (semop (semid, sops, 1) == -1)

{ perror ("error in semaphore operation");

exit (1);

}

}

void V(int semid, int index)

{/\* procedure to perform a V or signal operation on semaphore of given index \*/

struct sembuf sops[1]; /\* define operation on semaphore with given index \*/

sops[0].sem\_num = index;/\* define operation on semaphore with given index \*/

sops[0].sem\_op = 1; /\* add 1 to value for V operation \*/

sops[0].sem\_flg = 0; /\* type "man semop" in shell window for details \*/

if (semop (semid, sops, 1) == -1)

{ perror ("error in semaphore operation");

exit (1);

}

}

int main (void)

{ pid\_t pid; /\* variable to record process id of child \*/

/\* shared memory elements \*/

caddr\_t shared\_memory; /\* shared memory base address \*/

int \*in; /\* pointer to logical 'in' address for writer \*/

int \*out; /\* pointer to logical 'out' address for reader \*/

int \*buffer; /\* logical base address for buffer \*/

/\* semaphore elements \*/

int semid; /\* identifier for a semaphore set \*/

/\* local variables \*/

int i\_child, j\_child; /\* index variables \*/

int value; /\* value read by child \*/

/\* set up shared memory segment \*/

shared\_memory=mmap(0, SHARED\_MEM\_SIZE, PROT\_READ | PROT\_WRITE,

MAP\_ANONYMOUS | MAP\_SHARED, -1, 0);

if (shared\_memory == (caddr\_t) -1)

{ perror ("error in mmap while allocating shared memory\n");

exit (1);

}

/\* set up pointers to appropriate places in shared memory segment \*/

buffer = (int\*) shared\_memory; /\* logical buffer starts at shared segment \*/

in = (int\*) shared\_memory + BUF\_SIZE\*sizeof(int);

out = (int\*) shared\_memory + (BUF\_SIZE+1)\*sizeof(int);

\*in = \*out = 0; /\* initial starting points \*/

/\* create and initialize semaphore \*/

semid = sem\_init();

if (-1 == (pid = fork())) /\* check for error in spawning child process \*/

{ perror ("error in fork");

exit (1);

}

if (0 == pid)

{ /\* processing for child == reader \*/

printf ("The reader process begins.\n");

for (i\_child = 0; i\_child < run\_length; i\_child++)

{ P(semid, buf\_used); /\* wait semaphore for something used \*/

value = buffer[\*out];

\*out = (\*out + 1) % BUF\_SIZE;

printf ("Reader's report: item %2d == %2d\n", i\_child, value);

V(semid, buf\_space); /\* signal semaphore for space available \*/

if ((i\_child % 3) == 1)

sleep(1); /\* take time to process every third element \*/

}

printf ("Reader done.\n");

}

else

{ /\* processing for parent == writer \*/

printf ("The writer process begins.\n");

for (j\_child = 0; j\_child < run\_length; j\_child++)

{ P(semid, buf\_space);/\* wait semaphore for space available \*/

buffer[\*in] = j\_child\*j\_child; /\* put data in buffer \*/

\*in = (\*in + 1) % BUF\_SIZE;

printf ("Writer's report: item %2d put in buffer\n", j\_child);

V(semid, buf\_used); /\* signal semaphore for something used \*/

if ((j\_child % 4) == 0)

sleep(1); /\* take time to generate every fourth element \*/

}

wait (pid);

printf ("Writer done.\n");

/\* Remove the semaphore from the system and destroy the set of

semaphores and data structure associated with it. \*/

if (semctl (semid, 0, IPC\_RMID) < 0)

{ perror ("error in removing semaphore from the system");

exit (1);

}

printf ("Semaphore cleanup complete.\n");

}

exit (0);

}

/\* A readers/writers program for multiple readers and multiple writers \*/

#include <sys/types.h>

#include <unistd.h>

#include <stdio.h>

#include <sys/mman.h>

#include <sys/sem.h>

#define NUM\_READERS 5 /\* number of reader processes to be spawned \*/

#define NUM\_WRITERS 6 /\* number of writer processes to be spawned \*/

#define BUF\_SIZE 5 /\* logical size of buffer \*/

#define SHARED\_MEM\_SIZE (BUF\_SIZE+2)\*sizeof(int) /\* size of shared memory \*/

#define reader\_length 12 /\* number of iterations for reader in test run \*/

#define writer\_length 10 /\* number of iterations for writer in test run \*/

#define buf\_used 0 /\* semaphore array index to check buffer elts used \*/

#define buf\_space 1 /\* semaphore array index to check buffer elts empty \*/

#define mutex 2 /\* semaphore index for mutual exclusion to buffer\*/

int sem\_create(int num\_semaphores)

{ /\* procedure to create specified number of semaphores \*/

int semid;

/\* create new semaphore set of semaphores \*/

if ((semid = semget (IPC\_PRIVATE, num\_semaphores, IPC\_CREAT | 0600)) < 0)

{ perror ("error in creating semaphore");/\* 0600 = read/alter by user \*/

exit (1);

}

return semid;

}

void sem\_init(int semid, int index, int value)

{ /\* procedure to initialize specified semaphore to given value \*/

if (semctl (semid, index, SETVAL, value) < 0)

{ perror ("error in initializing first semaphore");

exit (1);

}

}

void P(int semid, int index)

{/\* procedure to perform a P or wait operation on a semaphore of given index \*/

struct sembuf sops[1]; /\* only one semaphore operation to be executed \*/

sops[0].sem\_num = index;/\* define operation on semaphore with given index \*/

sops[0].sem\_op = -1; /\* subtract 1 to value for P operation \*/

sops[0].sem\_flg = 0; /\* type "man semop" in shell window for details \*/

if (semop (semid, sops, 1) == -1)

{ perror ("error in semaphore operation");

exit (1);

}

}

void V(int semid, int index)

{/\* procedure to perform a V or signal operation on semaphore of given index \*/

struct sembuf sops[1]; /\* define operation on semaphore with given index \*/

sops[0].sem\_num = index;/\* define operation on semaphore with given index \*/

sops[0].sem\_op = 1; /\* add 1 to value for V operation \*/

sops[0].sem\_flg = 0; /\* type "man semop" in shell window for details \*/

if (semop (semid, sops, 1) == -1)

{ perror ("error in semaphore operation");

exit (1);

}

}

int main (void)

{ pid\_t pid; /\* variable to record process id of child \*/

/\* shared memory elements \*/

caddr\_t shared\_memory; /\* shared memory base address \*/

int \*in; /\* pointer to logical 'in' address for writer \*/

int \*out; /\* pointer to logical 'out' address for reader \*/

int \*buffer; /\* logical base address for buffer \*/

/\* semaphore elements \*/

int semid; /\* identifier for a semaphore set \*/

/\* record of spawned processes \*/

pid\_t proc[NUM\_READERS+NUM\_WRITERS];

/\* local variables \*/

int p\_count, i; /\* index variables \*/

int value; /\* value read by child \*/

/\* set up shared memory segment \*/

shared\_memory=mmap(0, SHARED\_MEM\_SIZE, PROT\_READ | PROT\_WRITE,

MAP\_ANONYMOUS | MAP\_SHARED, -1, 0);

if (shared\_memory == (caddr\_t) -1)

{ perror ("error in mmap while allocating shared memory\n");

exit (1);

}

/\* set up pointers to appropriate places in shared memory segment \*/

buffer = (int\*) shared\_memory; /\* logical buffer starts at shared segment \*/

in = (int\*) shared\_memory + BUF\_SIZE\*sizeof(int);

out = (int\*) shared\_memory + (BUF\_SIZE+1)\*sizeof(int);

\*in = \*out = 0; /\* initial starting points \*/

/\* create and initialize semaphores \*/

semid = sem\_create(3);

sem\_init(semid, buf\_used, 0);

sem\_init(semid, buf\_space, BUF\_SIZE);

sem\_init(semid, mutex, 1);

/\* spawn writer processes \*/

for (p\_count = 1; p\_count <= NUM\_WRITERS; p\_count++)

{ if (-1 == (pid = fork())) /\* spawn child process \*/

{ perror ("error in fork");

exit (1);

}

if (0 == pid)

{ /\* processing for parent == writer \*/

printf ("The writer process %d begins.\n", p\_count);

for (i = 0; i < writer\_length; i++)

{value = 100\*p\_count + i;/\* writer == first digit of value \*/

P(semid, buf\_space);/\* wait semaphore for space available \*/

P(semid, mutex); /\* wait semaphore for buffer access \*/

buffer[\*in] = value;/\* put data in buffer \*/

\*in = (\*in + 1) % BUF\_SIZE;

V(semid, mutex); /\* signal semaphore for buffer access \*/

V(semid, buf\_used); /\* signal semaphore for something used \*/

/\*printf ("Writer %d: item %2d put %d in buffer\n", j\_child);\*/

}

printf ("Writer %d done.\n", p\_count);

exit(0);

}

else proc[p\_count-1] = pid;

}

/\* spawn reader processes \*/

for (p\_count = 1; p\_count <= NUM\_READERS; p\_count++)

{ if (-1 == (pid = fork())) /\* spawn child process \*/

{ perror ("error in fork");

exit (1);

}

if (0 == pid)

{ /\* processing for child == reader \*/

printf ("The reader process %d begins.\n", p\_count);

for (i = 0; i < reader\_length; i++)

{P(semid, buf\_used); /\* wait semaphore for something used \*/

P(semid, mutex); /\* wait semaphore for buffer access \*/

value = buffer[\*out];/\* take data from buffer \*/

\*out = (\*out + 1) % BUF\_SIZE;

V(semid, mutex); /\* signal semaphore for buffer access \*/

V(semid, buf\_space); /\* signal semaphore for space available \*/

printf ("Reader %d: item %2d == %2d\n", p\_count, i, value);

if ((i+p\_count)%5 == 0) /\* pause somewhere in processing \*/

sleep(1); /\* to make output more interesting \*/

}

printf ("Reader %d done.\n", p\_count);

exit(0);

}

else proc[p\_count+NUM\_READERS-1] = pid;

}

/\* wait for all children to finish \*/

printf("All child processes spawned by parent\n");

printf("Parent waiting for children to finish\n");

for (p\_count = 0; p\_count<NUM\_WRITERS+NUM\_READERS; p\_count++)

waitpid(proc[p\_count], NULL, 0);

/\* Remove the semaphore from the system and destroy the set of

semaphores and data structure associated with it. \*/

if (semctl (semid, 0, IPC\_RMID) < 0)

{ perror ("error in removing semaphore from the system");

exit (1);

}

printf ("Semaphore cleanup complete.\n");

exit (0);

}