CSC 241
Computer Systems
Laboratory Manual

WAKE FOREST
UNIVERSITY

Department of Computer Science

Table of Contents

Introduction

This laboratory manual contains lab exercises to be used in the Computer Systems course. The purpose of these labs is to introduce the student to several practical aspects of operating systems through the use of some facilities provided by the system.

The first three labs reinforce the concept of a process, a fundamental entity in Operating Systems design. Lab 1 introduces the concept of signals, an asynchronous mechanism for process communication. Lab 2 will show how processes are created in the Unix environment and what information is shared between a parent and a child process. Lab 3 introduces the concept of a shell and how it handles user's requests by creating processes.

Labs 4, 5, 6 and 7 deal with the critical section problem and the use of low-level synchronization primitives, semaphores, to coordinate the use of shared resources. Since the main objective of these labs is to introduce how to use these primitives and how to share resources among processes, a library, `ezipc.h` is provided which you can use. This way, the programming effort is invested in solving the problem and not in interpreting the complex data structures used by Unix to implement semaphores.

Lab 8 will reinforce the concept of scheduling by writing a simple CPU scheduler. Different scheduling algorithms are analyzed and performance of each algorithm is evaluated under different job mixes.

Finally, Lab 9 will be used to compare different virtual memory mechanisms. Different paging and segmentation algorithms are simulated and results compared with respect to performance.

Deliverables

All labs must be accompanied by a compressed (.zip) file with the following documents/files:

1- Source C code named `labX.c`

2- A README.txt file which contains the following lines & information:

    NAME: Your Name
    CSC-241 - Lab X
    brief explanation on the purpose of your lab
    compile & execution instructions

*Un-identifiable programs will not be graded.

3- Answers to the lab questions. **Must be typed**.

**Any submission that does not contain these documents and their relevant information will have points deducted.

Lab 1

Signals
Signals are used to notify a process of a particular event. Signals are sometimes compared to hardware interrupts which occur at the hardware level such as I/O or timer interrupts. Due to their similarities, signals have been described as software interrupts.

When a signal is sent to a process, a signal handler may be entered (depending on the current disposition of the signal). Throughout the development of Unix and its many flavors, the signal mechanism has gone through many changes. The POSIX standards provided a fairly well defined set of interfaces for using signals in code, and today the Linux implementation of signals is fully POSIX-compliant. Note that reliable signals require the use of the newer **`sigaction`** interface, as opposed to the traditional **`signal`** call.

Signals may be synchronous or asynchronous to the process depending on the source of the signal. Synchronous signals occur within the execution stream of the process when an irrecoverable error occurs, and immediate termination of the process is required (for example the execution of an illegal instruction). Asynchronous signals are external to the process. For example signals sent by another process, via the **`kill`** or **`sigsend`** system call.

Every signal has a unique name (for example `SIGINT`) and an id associated with it. Signals also have a default action that takes place when the signal is received. A process may define a signal handler function which will be executed when the signal is received. If no signal handler is defined the default action is taken.

Linux supports the standard signals listed below. Some signal numbers are architecture dependent as listed in the Value column.

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25		Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

The signals *SIGKILL* and *SIGSTOP* cannot be caught, blocked, or ignored.

There are other non-POSIX signals but they will not be discussed here. You may check the man pages if interested.

Description

`signal()` system call:

A process can change the disposition of a signal using `sigaction` or (less portably) `signal`. Using these system calls, a process can elect one of the following behaviors to occur on delivery of the signal:

- perform the default action

- ignore the signal

- catch the signal with a *signal handler*, a programmer-defined function that is automatically invoked when the signal is delivered.

The `signal(`*`signum, sighandler`*`)` system call installs a new signal handler for the signal with number *signum*. The signal handler is set to *sighandler* which may be a user specified function, or either SIGIGN or SIGDFL.

Upon arrival of a signal with number *signum* the following happens. If the corresponding handler is set to SIGIGN, then the signal is ignored. If the handler is set to SIGDFL, then the default action associated with the signal occurs. Finally, if the handler is set to a function *sighandler* then first either the handler is reset to SIGDFL or an implementation-dependent blocking of the signal is performed and next *sighandler* is called with argument *signum*.

Using a signal handler function for a signal is called "catching the signal". The signals SIGKILL and SIGSTOP cannot be caught or ignored.

The original Unix signal() would reset the handler to SIGDFL, and System V (and the Linux kernel and libc4,5) does the same. On the other hand, BSD does not reset the handler, but blocks new instances of this signal from occurring during a call of the handler. The glibc2 library follows the BSD behavior.

Synopsis

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

The most difficult part of issuing a `signal` is deciphering its prototype. In essence, the prototype declares `signal` to be a function that accepts two arguments, an integer `signum` value and a pointer to a function, which is called when the signal is received. If the invocation of `signal` is successful, it returns a pointer to a function that returns nothing (`void`). [Gray]

`alarm()` system call:

The `alarm()` system call sets a timer to deliver the signal `SIGALRM` to the calling process after the specified number of seconds. If an alarm has already been set with `alarm()` but has not been delivered, another call to `alarm()` will supersede the prior call. The request `alarm(0)` voids the current alarm and the signal `SIGALRM` will not be delivered.

The return value of alarm() is the amount of time left on the timer from a previous call to alarm(). If no alarm is currently set, the return value is 0.

Synopsis

```
#include <unistd.h>
unsigned alarm(unsigned seconds);
```

`pause()` system call:
The `pause()` system call forces a process to pause until a signal is received from either the `kill()` system call or an interval timer. Upon termination of a signal handler started during a `pause()`, the `pause()` call will return.

`kill()` system call
The `kill()` system call sends a signal to the processes specified by the *pid operand(s).*

Synopsis

```
kill [-s signal_name] pid ...
kill -l [exit_status]
kill -signal_name pid ...
kill -signal_number pid ...
```

Only the super-user may send signals to other users' processes.

The options are as follows:

-s signal_name
  A symbolic signal name specifying the signal to be sent instead of the default TERM.
-l [exit_status]
  if no operand is given, list the signal names; otherwise, write the signal name corresponding to exit_status.
-signal_name
  A symbolic signal name specifying the signal to be sent instead of the default TERM.
-signal_number
  A non-negative decimal integer, specifying the signal to be sent instead of the default TERM.

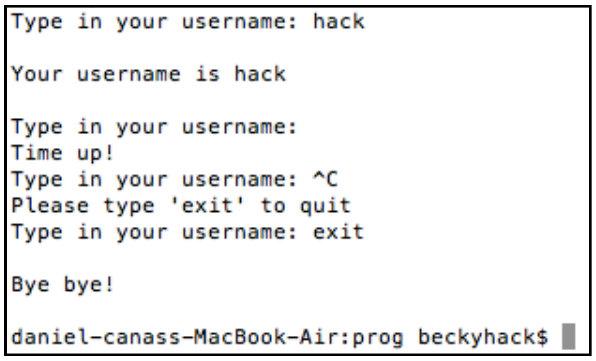
Program Guidlines

In this lab you will learn how to intercept signals using a signal handler. Be aware that there may be a race condition when processing a signal.

To illustrate the concept of signal handling, write program that prompts for a username, waits for user input, if none is given after a timeout of 5 seconds, print the message “Time up!”, reset the alarm and display the prompt again. To set the timeout value, use the `alarm()` system call. When the username input is received, the program echoes the input (“Your username is ________”) and then redisplays the username prompt. But if the input is the word “exit” then the program exits immediately.

Your program should also set an interrupt handler to catch the ^C (SIGINT) signal from the terminal or from another process, and will display a message stating “Please type ‘exit’ to quit” and redisplay the username prompt. **The program should not exit when control-C is pressed**.

Here is example output and user input for this program:



```
Type in your username: hack

Your username is hack

Type in your username:
Time up!
Type in your username: ^C
Please type 'exit' to quit
Type in your username: exit

Bye bye!

daniel-canass-MacBook-Air:prog beckyhack$
```

Questions
1- Explain one of the fundamental differences between `signal()` and `sigaction()` system calls.
2- What is the purpose of the SIGKILL signal if it cannot be intercepted?

***NOTE*: Please refer to page 3 of the Lab Manual for submission requirements.

Simple signal handling program

```
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>
// Prototypes
void alarmHandler(int);
void quitHandler(int);
main () {

// set signal handlers
signal ( SIGALRM, alarmHandler);
signal ( SIGINT,  quitHandler );
 alarm(10);
 pause();
 printf("\nNormal Termination\n");
exit(0);
}
void alarmHandler( int theAlarm ) {
 printf("\nRRRRIIIIIIIINNNNNNNNNGGGGGG !!!!! \n");
return;
}
void quitHandler ( int theInt) {
 printf("\nNot QUITTING!!!!\n");
return; }
```

Lab 2

Process Creation in UNIX
Processes are essential resources in the UNIX system. There are user processes and system processes. Some processes like Web Server, print servers and others are called daemons. In this lab we will deal only with user processes. A process is defined by an entry in the Process Control Table (or PCB). A new entry is created every time a process is created. In addition, a process is assigned a *unique process id*. This is a number assigned by the operating system by incrementing one to the last id assigned. If it reaches its maximum, it will wrap-around and start from one, until an available id is found. If there are no more id's available, then the maximum number of active processes has been reached and the process is not created.

Description
UNIX uses the system call `fork()` to create a process. The `fork()` system call will make a copy of the process requesting the service. This call has the unique feature that it returns twice, once in the calling process and once in the newly created process.

A process is created by invoking the `fork` system call as depicted below.

```
procid  =  fork();
```

The `fork()` replicates the parent process. Both the parent and the child processes resume execution at the instruction following the `fork()`. In the parent process *procid* will receive the *process id* of its child, while in the child process the function `fork()` will return a value of zero. This makes it possible for a process to identify itself as either a parent or a child process.

Execution rule:

Parent does not wait for children to terminate.

The following code illustrates a typical way of using the `fork()` construct.

```
procid = fork();
if ( procid == 0)
       { do child processing }
else
   { do parent processing }
```

The child process inherits the environment of the parent process. Even though a new process is created with new stack and data segments, files opened by the parent process are accessible to the child process. This is the only sharing which occurs among the processes.

Degree of sharing:

Files and associated buffers currently in use by the parent are shared with the child.
The child receives a copy of the variables declared by the parent.

A mechanism is available to avoid this file sharing. The interested reader is referred to [BAC-86]

Program Guidlines

In this lab you will learn how to create processes and see how information is shared among them. You should be very careful when creating processes because a small mistake can bring the entire system to a halt. A line of code like:

```
      while(1) pid = fork();
```

will very quickly reach the maximum number of processes and will bring the system to a halt. Why?

Write a C program the will accomplish the following:

-open a text file named `lab2.txt` (be sure to include error handling)
-create a process that will read and display each line of the file until the end-of-file (EOF) is reached and then exits.
-the parent process will also read and display each line of the same file and exits when the end-of-file is reached.
-output lines must be preceded by the process id of the process.

Sample `lab2.txt` text file:

1 this is line one of the file
2 this is line two of the file
3 this is line three of the file
4 this is line fourth of the file
...
100 this is the hundredth and last line of the file

Questions:
1- What is the maximum number of processes supported by your UNIX System?
2- On your UNIX system what data structure(s) is used to implement the PCB?
3- Investigate the `sleep()` system call. What does it do?
4- What happens if you insert the code:
```
 sleep(2);
```
before the read instruction in the child process? Explain your answer.
5- What is the effect if the same sleep system call is inserted before the read instruction, but this time in the parent process? Explain your answer.
6- Consider the following C program:
```
      #include  <stdio.h>
      #include  <stdlib.h>
      int pid;
      main () {
      printf("hello ");
      pid=fork();
      if ( pid==0 ) {
        printf(" hello world\n");
        exit (0);
       }
       printf("world\n");
      }
```

What is the output produced by the program? Explain your answer.

7- This following illustrates how to wait for all child processes to terminate, and how to handle the returned value of a process. The `wait()` system call waits for a process to terminate. It blocks the calling process until a child processes terminates. The `wait()` returns a `pid_t` type (usually a `long int`) with the terminated child process id and receives as an argument the terminating status set by the child process.
The terminating status variable values is a 32 bit integer and has two different formats, one if the process terminated normally, and a second when the process terminated due to an uncaught signal.
Describe the format of both possible values of the terminating status word. You may use internet resources to answer this question.

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

int main () {
int save_status, i, status;
pid_t child_pid;
for ( i=0;i<3; i++ ) {
if ( fork() == 0 ) {
 sleep (1);// code for child process
 exit( i ) ;
}
}
 while ((child_pid =wait(&save_status ))  != -1 )  {
  status = save_status >> 8;
  printf("Child pid: %d with status %d\n",child_pid,status);
 }
}
```

***NOTE*: Please refer to page 3 of the Lab Manual for submission requirements.

Lab 3

Simple Shell
The Linux shell is a command interpreter. It allows users to execute commands without knowledge of the internals of the system. A shell can execute commands directly from the terminal input window or they can read shell commands from a file. These files are known as shell script files.

There are many shells available to the user. Some of the most common ones are: `sh`, `bash` and `csh`. They very much perform the same functions and differ basically in the way the can be programmed through script files. This lab concentrates on command line input to the shell.

Commands from the terminal window are read by the shell and executed on behalf of the user. Commands can be programs, like `g++`, or be Linux system commands, like `ls` , `pwd`, etc. System commands are usually stored in the `/usr/bin` directory but may vary on different Linux flavors. You may find that some of the commands do not appear as programs in the `/usr/bin` directory. This is because the shell will have this commands built in for efficiency reasons.

To execute a command the shell creates a process to execute the code of the particular command. For example to execute the `ls` command, the shell will create a process to execute the code for `ls`. The shell will wait for the command to terminate and will the prompt the user for another command. Output for the command got to the standard output file. The shell uses three predefined files for input and output. The `stdin`, `stdout` and `stderr`, for input, output end error messages. By default, all files are associated with the terminal window.

Description
To execute a command the shell requires three systems calls, `fork`, `exec` and `wait`. The `fork` system call is used to create a process (see Lab 2). `exec()` is used to overwrite the code of the child process with the code of the command to be executed, and the `wait()` is used to wait for the child process to terminate (See question 3, lab 2). This way the child process does not have to execute the same code as the parent process. The `exec()` has many variations, `execl`, `execv` ,`execle`, `execve`, `execlp` and `execvp.` The difference is in the way the arguments are sent.

When `exec()` is executed, the program file given by the first argument will be loaded into the caller's address space and over-write the program there. Then, the arguments will be provided to the program and execution starts. Once the specified program file starts its execution, the original program in the caller's address space is gone and is replaced by the new program. `exec()` returns a negative value if the execution fails. The most common failure is that the file could not be found .

`exec()` system call
Only the execlp() and `execvp()` system call will be discussed here. The execlp () library function is used when the number of arguments to be passed to the program to be executed is known in advance.

Synopsis

```
#include <unistd.h>
int execlp ( const char *file, const char *arg, . . . );
int execvp ( const char *file, char * const argv[] );
```

When using the `execlp` the initial argument, file, is a pointer to the file that contains the program code to be executed. If this file reference begins with a /, it is assumed that the reference is an absolute path to the file. If no / is found, each of the directories specified in the `PATH` variable is searched, and the first valid program reference found will be the one executed. It is a good practice to fully specify the program to be executed in all situations to prevent a program with the same name, found in a prior PATH string directory, from being inadvertently executed.

The ellipses in the execlp function prototype can be thought of as argument 0(*arg0*) through argument *n* (*argn*). These arguments are pointers to the null-terminated string that would be normally passed by the system to the program if it were invoked on the command line. This is, argument 0, by convention, should be the name of the program that is executing. This is usually the same as the value of `file`, although the program referenced by `file` may include an absolute path, while the value in argument 0 most often would not. Argument 1 would be the first parameter to be passed to the program (which, using `arvg` notation, would be `argv[1]`), argument 2 would be the second, and so on. The last argument to the `execp` library call must be `NULL` that is for portability reasons, cast to a character pointer.

Example:

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

int main ( int argc, char *argv[ ] ) {
 if( argc > 1 ) {
        execlp("/bin/cat", "cat", argv[1], (char *) NULL);
        perror("exec failure ");
        return 1;
 }
  printf("Usage: %s text_file\n", *argv);
//  cerr << "Usage: " << *argv << " text_file" << endl;
 return 2;
}
```

If the number of arguments for the program to be executed is dynamic, then the `execvp` call can be used. As with `execlp` call, the initial argument to `execvp` is a pointer to the file that contains the program code to be executed. However *unlike* `execlp`, there is only one additional argument that `execvp` requires. This second argument, defined as:

```
char *conts argv[ ]
```

specifies that a reference to an array of pointers to character strings should be passed. The format of this array parallels that of `argv` and, in many cases, is `argv`. If the reference is not the `argv` values for the current program, the programmer is responsible for constructing and initializing the new *argv-like* array. If the second approach is taken, the last element of the new `argv` array should contain a `NULL` address value.

`wait()` system call
`wait()` has also several variations, `wait()` and `waitpid()`.

Synopsis

```
#include <sys/wait.h>
pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

The `wait()` and `waitpid()` functions obtain status information pertaining to one of the caller's child processes. Various options permit status information to be obtained for child processes that have terminated or stopped. If status information is available for two or more child processes, the order in which their status is reported is unspecified.

The `wait()` function suspends execution of the calling process until status information for one of the terminated child processes of the calling process is available, or until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process. If status information is available prior to the call to `wait()`, return shall be immediate.

The `waitpid()` function is equivalent to `wait()` if the *pid* argument is -1 and the options argument is 0. Otherwise, its behavior shall be modified by the values of the pid and options arguments.

The *pid* argument specifies a set of child processes for which status is requested. The `waitpid()` function shall only return the status of a child process from this set:

If pid is equal to -1, status is requested for any child process. In this respect, `waitpid()` is then equivalent to `wait()`.

If *`pid`* is greater than 0, it specifies the process ID of a single child process for which status is requested.

If *pid* is 0, status is requested for *any* child process whose process group ID is equal to that of the calling process.

If *pid* is less than -1, status is requested for any child process whose process group ID is equal to the absolute value of *pid*.

If `wait()` or `waitpid()` returns because the status of a child process is available, these functions shall return a value equal to the process ID of the child process for which status is reported.

If the value of the argument `stat_loc` is not a null pointer, information shall be stored in the location pointed to by `stat_loc`. This value is the one returned by the terminating process (i.e *`exit(x);`*)

Program Guidlines

The purpose of this lab is to write a simple shell program. Your shell program should use the same style as the `Bourne` shell for running programs. In particular, when the user types a line such as:

```
command  [identifier[identifier]]
```

your shell should parse the command line to build `argv`. Use the execvp() system call to execute the file that contains the command. This way the `exec()` will search the `PATH` directory paths automatically.

The shell must also interpret the "&" operator as a command terminator. A command terminated with "&" should be executed concurrently with the shell rather than the shell waiting for the command to terminate before it prompts the user for another command (background execution).

If a command is given that execvp() cannot execute (such as an erroneous command), an appropriate error message must be outputted and the shell should be reprompted.

Your program should terminate once the ^C signal is used or when the user types in "exit".

Questions

1- How can the shell identify if the command was executed successfully or not?
2- Under what circumstances will the `exec()` return to the calling process?
***NOTE*: Please refer to page 3 of the Lab Manual for submission requirements.

Lab 4

Sleeping Barbers Problem
The sleeping barbers problem is a classical Operating Systems problem. You should be able to identify it with some of the problems discussed in class.In this lab you will write a program to solve the Sleeping Barber's problem. The original Sleeping Barber problem was proposed by Dijkstra.

Description
You will again use the IPC implementation under UNIX and the ezipc.h/ezipcc.h library to solve the problem. If ezipc is not used, points will be taken off. You may use SHOW() and SET() only for debugging purposes.

<u>Program Guidlines</u>
In this lab you will write a program to simulate the Sleeping Barber's problem.

There is a barbershop with **n** chairs for waiting customers, one barber's chair and one barber. If a customer enters the store and there are no free chairs, the customer leaves. If a customer enters the store and the barber is sleeping, the customer wakes up the barber and gets a haircut. Otherwise, a customer enters the store, takes a seat and waits. If the barber finishes a haircut and there are waiting customers, the barber cuts the hair of the next customer. Otherwise, the barber goes to sleep in his chair.

Before a customer leaves, he/she must go to the cashier and pay for the haircut. The cashier is able to service only one customer at a time Using semaphores and shared memory, write the functions to control the action of customers and the barber. Your program consists of a barber process (the main program) and a process for each customer. In the program you must display actions taken by the customer and the barber. (See sample output below)

The program should be controlled by the following parameters:

maximim_number_customers: since you don't want to enter an infinite loop generating customers, you will only generate maximim_number_customers. When that limit is reached, no more customer process should be forked.

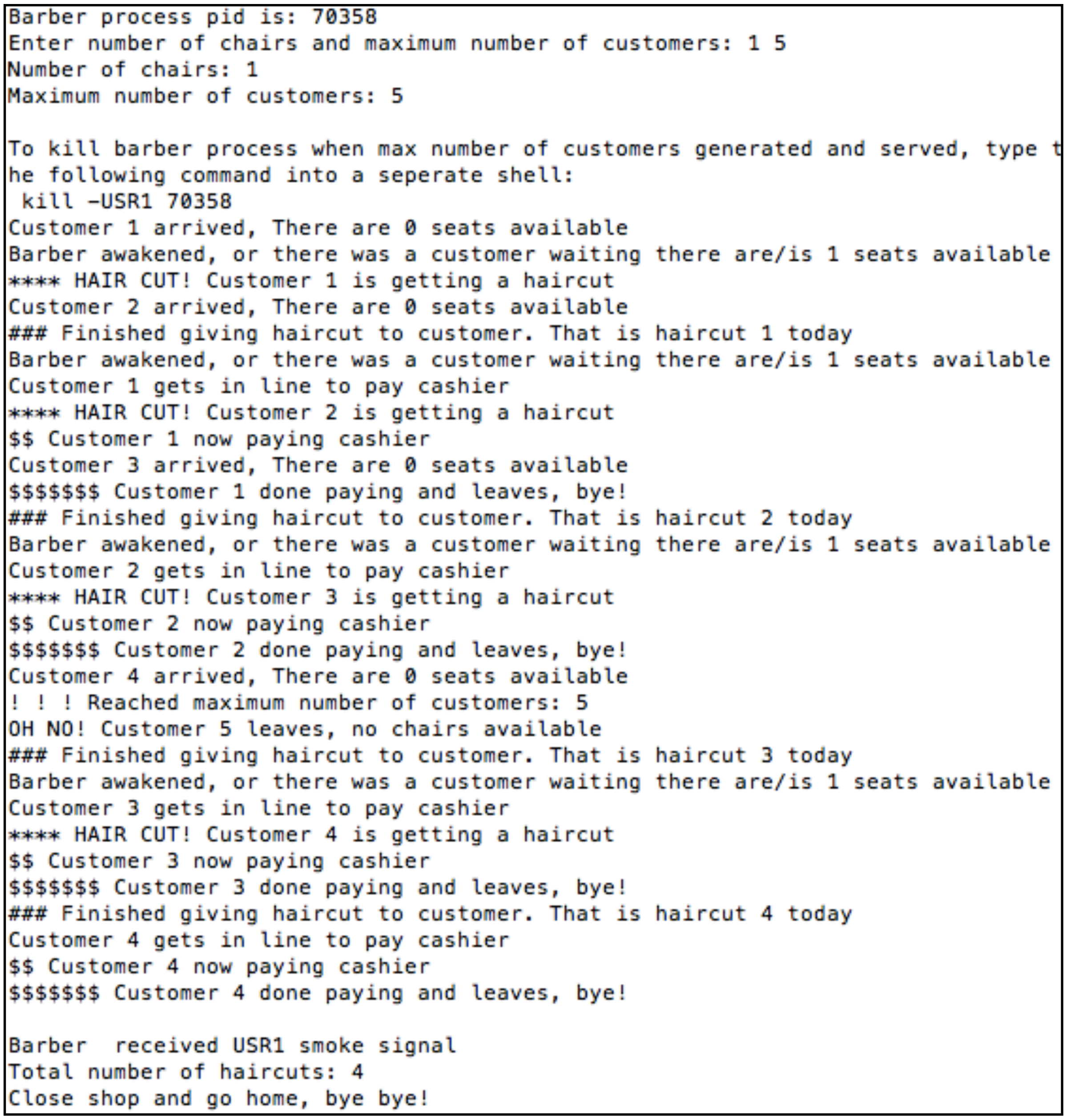
number_of_chairs: this is the maximum number of chairs in the waiting room.

The program must prompt for maximim_number_customers and number_of_chairs.

Since the barber does not know how many customers were generated, it will go to sleep and never wake up. That means that the process will never end. To terminate the barber process send a SIGUSR1 signal from a separate command line. This will wake up the barber due to the signal and the barber should print out the number of customers served and then close the shop and go home. Be sure to print out the barber process's ID so you know where to send the signal. For example, if the barber process's ID is 6004, to send this signal you would open up a new command line for your VM and type the following: kill -SIGUSR1 6004

The ezipc tool has a function SHOW(*semaphore*) where it takes in a semaphore variable name and returns the current value of that semaphore. This may be valuable for debugging your program, but it cannot be used if your final submitted program.

Here is an example output with 1 chair and 5 customers:



```
Barber process pid is: 70358
Enter number of chairs and maximum number of customers: 1 5
Number of chairs: 1
Maximum number of customers: 5

To kill barber process when max number of customers generated and served, type t
he following command into a seperate shell:
 kill -USR1 70358
Customer 1 arrived, There are 0 seats available
Barber awakened, or there was a customer waiting there are/is 1 seats available
**** HAIR CUT! Customer 1 is getting a haircut
Customer 2 arrived, There are 0 seats available
### Finished giving haircut to customer. That is haircut 1 today
Barber awakened, or there was a customer waiting there are/is 1 seats available
Customer 1 gets in line to pay cashier
**** HAIR CUT! Customer 2 is getting a haircut
$$ Customer 1 now paying cashier
Customer 3 arrived, There are 0 seats available
$$$$$$$ Customer 1 done paying and leaves, bye!
### Finished giving haircut to customer. That is haircut 2 today
Barber awakened, or there was a customer waiting there are/is 1 seats available
Customer 2 gets in line to pay cashier
**** HAIR CUT! Customer 3 is getting a haircut
$$ Customer 2 now paying cashier
$$$$$$$ Customer 2 done paying and leaves, bye!
Customer 4 arrived, There are 0 seats available
! ! ! Reached maximum number of customers: 5
OH NO! Customer 5 leaves, no chairs available
### Finished giving haircut to customer. That is haircut 3 today
Barber awakened, or there was a customer waiting there are/is 1 seats available
Customer 3 gets in line to pay cashier
**** HAIR CUT! Customer 4 is getting a haircut
$$ Customer 3 now paying cashier
$$$$$$$ Customer 3 done paying and leaves, bye!
### Finished giving haircut to customer. That is haircut 4 today
Customer 4 gets in line to pay cashier
$$ Customer 4 now paying cashier
$$$$$$$ Customer 4 done paying and leaves, bye!

Barber  received USR1 smoke signal
Total number of haircuts: 4
Close shop and go home, bye bye!
```

Questions
1- Identify the critical section(s) of the problem.
2- What code modifications are necessary if the owner hires another barber?

***NOTE*: Please refer to page 3 of the Lab Manual for submission requirements.

Lab 5

Conway's Problem
In this lab you will write a program to solve Conway's problem.

Description
You will again use the IPC implementation under UNIX and the ezipc.h/ezipcc.h library to solve the problem. If ezipc is not used, points will be taken off. You may use SHOW() and SET() only for debugging purposes.

Purpose
Conway's Problem: Write a program to read 80-column cards and write them as 125-character lines with the following changes. After every card image the string <EOL> is inserted. Every adjacent pair of asterisks ** is replaced by a **#**.

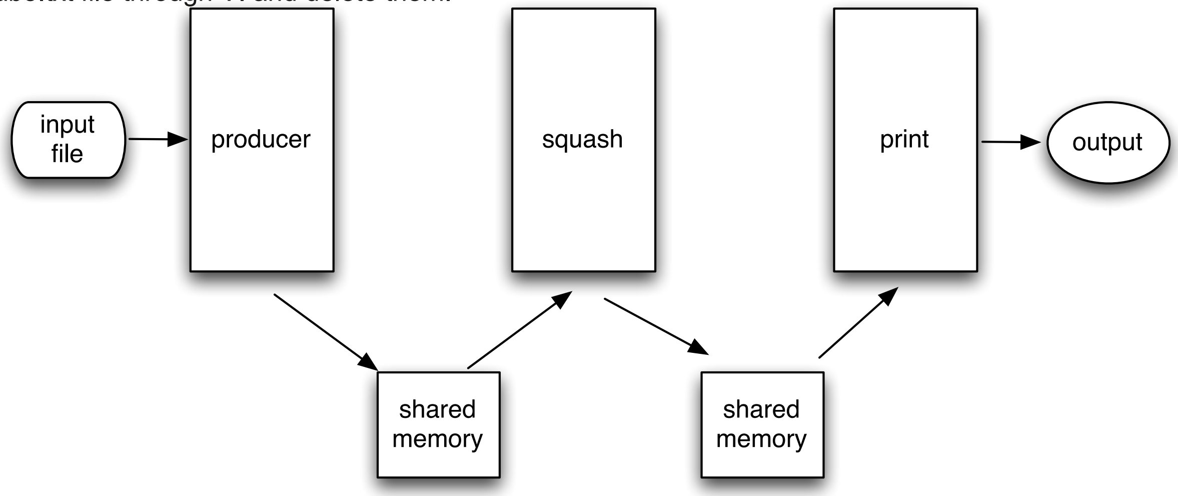
Of course Conway's problem can be solved by a single sequential program. However, it is difficult to be sure that you have taken care of all special cases such as pairs of asterisks at the end of a card and so on.

The problem has an elegant solution with three concurrent processes. One process `producer` reads cards and inserts the string <EOL> when necessary and passes characters through a one-character buffer to a process `squash`. `squash`, which knows nothing about 80-column cards, simply looks for double asterisks and passes a stream of modified characters to a process `print`. `print` takes the characters and prints them as 125-character lines.

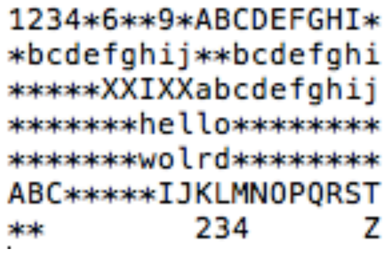
Program Guidlines
Write a program to solve Conway's problem with three separate processes for the producer, squasher and printer as described above using three concurrent processes *with the following modifications*:

- Your program should read in the card text from a file called "lab5.txt"
- Each *card* is 20 characters long (not 80 column cards).
- To simulate card input read from a file where each line has 20 characters.
- `Printer process`*:* prints 25-character lines to the screen (not 125 character lines).

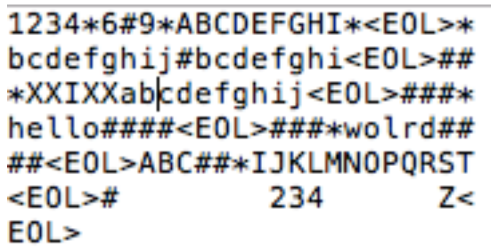
*Note that each shared memory should only have space for 1 character at a time.
***Attention Windows users!* If you are copying text you created in Windows over to your VM to run your program with, sometimes Windows puts an extra garbage character at the end of each line which can result in strange behavior in your output. Ensure these garbage characters are removed by going into the lab5.txt file through VI and delete them.



Here is an example, given the text file provided on the left, your program would be expected to output the text on the right:



```
1234*6**9*ABCDEFGHI*
*bcdefghij**bcdefghi
*****XXIXXabcdefghij
*******hello********
*******wolrd********
ABC*****IJKLMNOPQRST
**        234      Z
```



```
1234*6#9*ABCDEFGHI*<EOL>*
bcdefghij#bcdefghi<EOL>##
*XXIXXabcdefghij<EOL>###*
hello####<EOL>###*wolrd##
##<EOL>ABC##*IJKLMNOPQRST
<EOL>#        234      Z<
EOL>
```

Questions

1- Identify the critical section of the problem

***NOTE*: Please refer to page 3 of the Lab Manual for submission requirements.

Lab 6

Santa Claus Problem
The Santa Claus problem is a not so classical Operating Systems problem. You should be able to identify it with some of the problems discussed in class. This problem was originally presented by John Trono of St. Michael's College in Vermont [Trono].

Description
You will again use the IPC implementation under UNIX and the ezipc.h/ezipcc.h library to solve the problem. If ezipc is not used, points will be taken off. You may use SHOW() and SET() only for debugging purposes.

Program Guidlines

In this lab you will write a program to solve the Santa Claus problem.

Santa Claus sleeps in his shop up at the North Pole, and can only be wakened by either all nine reindeer being back from their year long vacation on the beaches of some tropical island in the South Pacific, or by some elves who are having some difficulties making the toys.

One elf's problem is never serious enough to wake up Santa (otherwise, he may **never** get any sleep), so, the elves visit Santa in a group of three. When three elves are having their problems solved, any other elves wishing to visit Santa must wait for those elves to return.

If Santa wakes up to find three elves waiting at his shop's door, along with the last reindeer having come back from the tropics, Santa has decided that the elves can wait until after Christmas, because it is more important to get his sleigh ready as soon as possible. (It is assumed that the reindeer don't want to leave the tropics, and therefore they stay there until the last possible moment).

They might not even come back, but since Santa is footing the bill for their year in paradise ... This could also explain the quickness in their delivering of presents, since the reindeer can't wait to get back to where it is warm. The penalty for the last reindeer to arrive is that it must get Santa while the others wait in a warming hut before being harnessed to the sleigh.

Here are some addition specifications:
- After the ninth reindeer arrives, Santa must invoke prepareSleigh, and then all nine reindeer must invoke getHitched.
- After the third elf arrives, Santa must invoke `helpElves`. Concurrently, all three elves should invoke `getHelp`.
- All three elves must invoke `getHelp` before any additional elves enter (increment the elf counter).

Santa should run in a loop so he can help many sets of elves. We can assume that there are exactly 9 reindeer, but there may be any number of elves.

Questions
1- Identify the critical section of the problem.
***NOTE*: Please refer to page 3 of the Lab Manual for submission requirements.

Note:
The programs should have the proper output, to show how each process is behaving. It should display every action, for example, elf arrives, santa is awakened, reindeer arrives, santa is helping elves, elves invokes `getHelp`, etc.

Lab 7

Unisex Bathrooms

The Unisex Bathroom is an interesting problem in resource allocation that is useful for illustrative purposes in the study of computer operating systems and parallel processing.

Description

You will again use the IPC implementation under UNIX and the ezipc.h/ezipcc.h library to solve the problem. If ezipc is not used, points will be taken off. You may use SHOW() and SET() only for debugging purposes.

Program Guidlines

In this lab you will write a program to solve the Unisex Bathroom problem.

The Unisex Bathroom problem consists of a bathroom with n stalls and some number of males and females that share the bathroom. A male or female may be using a stall in the bathroom, waiting to use the bathroom, or doing something else. There are different ways or strategies that can be used in scheduling individuals to use the bathroom.

If someone of the opposite sex is currently using the bathroom, then an arriving individual must wait for the bathroom to empty. No additional members of the sex that are currently using the bathroom may enter the bathroom. This strategy is not 100% fair since some individuals may get to use the bathroom before others of the opposite sex that arrived earlier.

Problem Assumptions

-Males and females cannot use the bathroom at the same time.
-An individual will spend some time doing other things (other than using the bathroom).
-An individual will spend some time using the bathroom
-An individual cannot do other things while waiting to use the bathroom
-An individual does not like to wait to use the bathroom
-Each individual may spend different amounts of time in the bathroom

Description

- The number of stalls is read from the command line after proper prompt has been issued.
- To terminate the process a signal handler will intercept the ^C signal and display:
- Number of females that uses the bathroom.
- Number of males that used the bathroom.
- Print a statement saying “Program terminated”.
- Starvation must be avoided.
- Times spend in the bathroom is computed for every customer using the following formula:
    - *time = (customer_id * 3) % 4 +1*
- The program must display the following information as it executes:
- Customer arrival, identified by customer #, gender and action taken (enters bathroom or waits)
- If customer waits, display the size of the waiting line. • If customer enters or leaves the bathroom, display:
    - customer id
    - gender
    - number of customers inside the bathroom

Input

The customer gender and time of arrival must be read from a file called “unisex.txt” that has the following format: time_of_arrival gender

For example: 2F —> a female customer that arrives at time 2

So a file with the following text will have customers arrive in the order shown below:

```
5

1F

2M

3M

6F

9F
```

The previous example has the following meaning:

Female arrived at time 1
Male arrived at time 2
Male arrived at time 3
Female arrived at time 6
Female arrived at time 9

***NOTE*: Please refer to page 3 of the Lab Manual for submission requirements.

Lab 8

Processor Scheduling

Processes compete or cooperate for the use of resources in a computer system. The CPU is the most sought-after resource. The operating system assigns the CPU to processes dictated by a well defined policy.

There are many CPU scheduling policies. There is no one universal policy that will perform better in all circumstances. Not only the hardware, but the job mix is important in determining the scheduling policy used by an operating system. For example, a general purpose system will have a very different scheduling policy than an Operating System for a dedicated real time application.

Description

One of the most efficient ways to compare different algorithms is to run a simulation and compare the results. In the particular case of CPU scheduling, the best way to compare the performance of different algorithms is to simulate their behavior. This is very efficient because implementing the algorithms in a real life system is very time consuming and requires much effort.

Program Guidlines

In this lab you will write a program to simulate different CPU scheduling algorithms. The program will read a series of parameters which will describe the scheduling policy, and the characteristics of a set of processes. With these parameters, the program will simulate the execution of the processes and display some statistics once the simulation terminates. The program will simulate the execution of a set of processes using one of the following scheduling policies:

First Come First Serve
Priority
Round Robin
Shortest Job Next
Shortest Remaining Time Next

Input:

```
Number-of-Concurrent-Processes Overhead-time
Algorithm-name <time slice for Round Robin>
Process-name Priority arrival-time CPU I/O CPU I/O CPU I/O CPU I/O... CPU 0
Process-name Priority arrival-time CPU I/O CPU I/O CPU I/O CPU I/O... CPU 0
…
Process-name Priority arrival-time CPU I/O CPU I/O CPU I/O CPU I/O... CPU 0
```

The Number-of-Concurrent-Processes will determine the number of processes that the system can multiprogram. The Algorithm-name can be one of the following Priority, SJN (Shortest Job Next), SRTN (Shortest Remaining Time Next), FCFS (First Come First Serve) or RR (Round Robin)

Except for the case of SJN and SRTN where the first two letters must be used, only the first letter of the algorithm name is significant. If the algorithm is Round Robin then it must be followed by the time slice. It must be noted that the priority algorithm refers to internal and not external priorities.

This first two data lines determine the runtime parameters.

The runtime parameters are followed by the job mix load.

Each line will characterize a process as follows:

The *process name* is the name of the process for identification purposes even though the Operating System will know the process by its process id assigned as they are loaded to the input queue.

The process name is followed by the *process priority* which will be used in priority assignment. It must be present, but will be ignored if another processor assignment policy is selected. The priority parameter is followed by the *arrival time*. This is the time at which the process requests execution. At this time it will be *eligible* for loading into the ready queue. Of course, loading depends on the Number-of-Concurrent-Processes parameter. The arrival time is followed by a *sequence of CPU bursts and I/O bursts*. A CPU burst indicates the amount of time required by the process before it requests a resource which will force it to relinquish the CPU and move to the wait queue. The wait time is determined by the I/O bursts which follows the CPU burst. Pairs of CPU I/O Burst will characterize the execution pattern of the process which must end with a CPU burst and an I/O burst of 0. The CPUI/O bursts may be enclosed in parenthesis to indicate repetition. For example a pair of the form 3(2 1) is equivalent to 2 1 2 1 2 1.

Sample Input File:

```
10    0
RR    50
Proc1 0     0   20     5    30    15    15     2    10     0
Proc2 0     0   12     3     5     5    13     8     3     0
Proc3 0     0   12     8    19    17    18    15     2     0
Proc4 0     0   20     5    30    15    15     2    10     0
Proc5 0     0   12     3     5     5    13     8     3     0
Proc6 0     0   12     8    19    17    18    15     2     0
```

Sample Output:
The previous input will produce the following output:

```
Wall clock:       318
Algorithm:  Round Robin

 process  process  arrival  start  total   total    total  end
    id      name     time    time   CPU   I/O time  ready  time
    1      proc1      0       0     75      22       204   301
    2      proc2      0       0     33      16       242   291
    3      proc3      0       0     51      40       212   303
    4      proc4      0       0     75      22       219   316
    5      proc5      0       0     33      16       257   306
    6      proc6      0       0     51      40       227   318
```

Things to consider.
Multiple interrupts can occur at the same time. This may affect the results of the simulation. For example, if an I/O termination, and a new job processes enters the system, the order in which this interrupts are processed is important, because it will determine the order in which the processes are placed in the ready queue. The same may happen with multiple I/O interruptions occurring at the same time.

To solve the problem, interrupts are assigned priorities. The simulator will use the following priorities, from highest to lowest:

CPU - time quantum, process termination, preemption.
I/O - In case of multiple I/O occurring, at the same time they will be processed in the order in which they were originated, that is the process that has been the longest in the wait queue has preference
Process creation

***NOTE*: Please refer to page 3 of the Lab Manual for submission requirements.

```
for (i=0; i < sim_steps; i++) {
   do {
          get size n of next request
          mm_request(n);
       }
   while (request successful )

   record memory utilization
   select block p to be released
   release (p)
}
```

Lab 9

Main Memory Management
In this project, we will examine main memory management in systems without virtual memory. The memory manager of such systems must maintain a list of free spaces, called holes. Processes can request and release blocks of variable sizes by invoking the appropriate functions of the memory manager.

Your tasks are the following:

--Design a memory manager that supports the following allocation strategies, s

first-fit
next-fit
best-fit
worst-fit

-Evaluate the different strategies by comparing how well they maintain the available memory

Description

The memory manager
The memory manager is a set of functions that manage a hypothetical physical memory.

Main Memory
The memory manager is intended to manage a linear physical memory. While developing the manager, we need no access to the actual physical memory in the system. Instead, we can emulate physical memory by allocating a sequence of consecutive bytes, and develop all the necessary functions using this emulated main memory. The functions use pointers instead of physical addresses.
A main memory of size *mem_size* may be created in one of two ways:

1. By using a dynamic memory management function of the OS, such as mm = `malloc`(mem_size) in UNIX.
2. By declaring a character array, `mm[mem_size]`, that represents the physical memory.

In both cases, we obtain a sequence of `mem_size` consecutive bytes - the emulated main memory. The pointer `mm` corresponds to physical address 0 of the actual physical memory. All holes and allocated blocks are maintained by the memory manager within this main memory array. (Note that type-casting must be used whenever the tag, size and pointer fields, which are of different types, are written into or read from this memory.)

The user interface
The memory manager must provide the following functions that can be invoked by other processes:

`1. void *mm_request(int n)`
This function requests a block of n consecutive bytes of memory. If the request can be satisfied, it returns a pointer to the first usable byte of the allocated block. If the request cannot be specified - because there is no hole large enough or the parameter n is illegal (e.g. a negative number) - it returns NULL. (Note that this function is analogous to the malloc function of UNIX.)

`2. void *mm_release(void  *p)`
This function releases a previously allocated block of memory, pointed to by the pointer p. If the released block is adjacent to another hole, the function must coalesce the hole to prevent the fragmentation of memory into increasingly smaller holes. If the released block is surrounded by

allocated blocks, it is simply appended to the list of holes. (Note that this function is analogous to the free function of UNIX.)

`3. void *mm_init(int mem_size)`
This function initializes the memory to become a single hole by creating the appropriate tag, size, and pointer fields. It returns the pointer mm introduced earlier.

The simulation experiment
Implement the four versions of the memory manager by varying the allocation strategy. Then design and carry out an experiment that will compare the performance of the allocation strategies in terms of:

1. average memory utilization
2. average number of steps needed to find an appropriate hole

To set up the simulation experiment, let us make the following assumptions. Request and release calls arrive at the manager in two separate queues. Releases are always processed first. Since any release is immediately satisfiable, the queue of releases is always empty. A request may or may not be satisfiable, depending on wether there is a hole large enough to accommodate the request. If that is not the case, the request remains on the queue until adequate space has been freed by subsequent release operations. The request queue is processed in FIFO order; otherwise starvation could occur.

Under these assumptions, we can model the behavior of the memory manager as follows. We start in a state where memory already consists of occupied blocks and holes; there is no release request, and the request at the head of the queue cannot be satisfied. At this point, the manager cannot do anything but wait for the next release. During this time, the memory allocation does not change. When a release is processed, the manager attempts to satisfy the request at the head of the queue. If this fails, the manager is again waiting for the next release. If the request succeeds, the manager immediately attempts to satisfy the next request in the queue, and so on. Thus, following every release, zero or more pending requests are satisfied, depending on the size of the released block and the sizes of the pending requests.

To evaluate the effectiveness of different allocation strategies, we must assume that memory is always filled maximum capacity. That means, we assume that the stream of requests in unbounded, and the queue is never empty.

The following is a skeleton of the simulation experiment based on the above assumptions. It gathers the performance statistics for one chosen allocation strategy.

The program consists of a main loop (line 1), which is repeated for a number os simulation steps (sim_steps). For each iteration of the outer loop, the program performs the following tasks. The inner loop (lines 2-5) attempt to satisfy as many of the pending requests as possible. Each request is generated on the fly by getting a random number, n, to represent the request size (line 3).

Once a request fails, the memory manager must wait until a block is released. Until then, the current memory allocation does not change, and we can record the current memory use. (Note that we are making the implicit assumption that the time during which a block resides in memory is independent of the block size. Thus, each iteration of the outer loop represents a constant interval of time during which the memory does not change.)

Next, one of the allocated blocks, p, is selected and released (lines 7-8). This frees some space and allows the simulation to proceed with the next iteration.

Generating request size
Let us assume that requests sizes can be approximated using a Gaussian distribution. Thus we can generate each new requests size, n, using the function;

```
        n = gauss(a,d)
```

where *a* is the average request size and d is the standard deviation. (Most math libraries will provide a set of such distribution functions.) Choosing a close to zero means that the average request will ask for just a few bytes of memory; a large *a* implies that processes typically request large blocks of memory. The standard deviation determines the shape of the Gaussian curve. A small *d* will result in a steep curve, implying that the majority of requests are clustered close to the mean; a large *d* implies that the distribution of sizes is more uniform.

Regardless of the values of *a* and *d*, the number *n* ranges from minus to plus *infinity*. Thus, we must truncate the range to reflect the physical size of memory. That means, whenever *n* exceeds mem_size, it is discarded because a request that exceeds the total memory size could never be satisfied. We must also discard all negative values of *n* (or use their absolute values) since all memory blocks are always greater than zero. (A block of size zero is, in principle, possible, but useless.)

Gathering performance data
There are two types of performance data we must gather. *Memory utilization* is expresses as the ratio of the allocated space to the total memory size. One such value is obtained during each iteration of the simulation loop. These values may be accumulated in a file and averaged at the end. Alternatively, a running average may be computed during each simulation run.

The second performance value is the average *search time*, i.e., number of steps needed to satisfy a request. This may be gathered by implementing the mm_request function for each memory allocation strategy. It must keep track of the number of holes that must be examined for each request and compute the average over the duration os the simulation run.

Choosing block to release
If we assume that the time a given block stays in memory is independent of its size, we can select the block to be released during each iteration at random. The simulation program must keep track of all the allocated blocks; a linked list is the most appropriate data structure for that purpose. If there are k elements on the list of allocated blocks, we choose a random number, p, between one and k, and release the block at position p on the list.

Summary of specific tasks

1. Design and implement the functions of the memory manager. Implement the different allocation strategies.
2. Design and implement a driver program that allows you to test and demonstrate the functionality of your memory manager.
3. Evaluate the performance of each allocation strategy using the simulation experiment. Generate families of curves that show how the average memory utilization and average search time (number of steps to satisfy a request) vary with the average request size. Repeat the experiments for different choices of the parameters a and d of the *gauss* function.

Review of allocations strategies

Best Fit
The allocator places a process in the smallest block of unallocated memory in which it will fit.
Problems:

1. It requires an expensive search of the entire free list to find the best hole.
2. More importantly, it leads to the creation of lots of little holes that are not big enough to satisfy any requests. This situation is called fragmentation, and is a problem for all memory-management strategies, although it is particularly bad for best-fit.

Solution:
One way to avoid making little holes is to give the client a bigger block than it asked for. For example, we might round all requests up to the next larger multiple of 64 bytes. That doesn't make the fragmentation go away, it just hides it.

Worst Fit
The memory manager places process in the largest block of unallocated memory available. The idea is that this placement will create the largest hole after the allocations, thus increasing the possibility that, compared to best fit, another process can use the hole created as a result of external fragmentation.

First Fit
Another strategy is first fit, which simply scans the free list until a large enough hole is found. Despite the name, first-fit is generally better than best-fit because it leads to less fragmentation.

Problems:
Small holes tend to accumulate near the beginning of the free list, making the memory allocator search farther and farther each time.

Solution:
Next Fit

Next Fit
The first fit approach tends to fragment the blocks near the beginning of the list without considering blocks further down the list. Next fit is a variant of the first-fit strategy. The problem of small holes accumulating is solved with next fit algorithm, which starts each search where the last one left off, wrapping around to the beginning when the end of the list is reached (a form of one-way elevator)

Appendix

Inter Process Communication

Interprocess communication is a fundamental concept in Operating Systems. Processes communicate with other processes to carry out a specific task. At the same time process share resources which must be protected while in use. For example variables which are updated by concurrent processes must be considered a critical resource and thus must be protected when used concurrently.

UNIX provides a library, the IPC system, which supports primitives for processes to interact with each other. The IPC provides processes with semaphores, shared memory and message queues. The three IPC resources are maintained by the kernel and can be shared among processes. Each resource provides a very complete set of primitives. The discussion of such structure is beyond the scope of this lab. A complete description of the implementation of the IPC mechanism in UNIX can be found in [1] and [10] describes the Linux implementation.

Description

The IPC implementation under UNIX is quite complex. For the exercise you will use a library which simplifies the use of semaphores and shared memory (`ezipc.h` (C version) or `ezipcc.h` (C++ version)). The library is described in [3] and is available for download in the class website. The use of the library is very simple. After the library is made available to your program via the include command in C, the semaphore structures must be initialized with the following statement:

```
SETUP();
```

command. This will create all necessary kernel data structures to use semaphores and shared memory within your program. Next, you must declare and initialize each semaphore you want to use with the following function call:

```
room = SEMAPHORE(SEM_CNT, 4);

bin_sem = SEMAPHORE(SEM_BIN, 1);
```

The first call declares `room` to be a counting semaphore initialized to 4, and the second call declares `bin_sem` to be a binary semaphore initialized to 1. Both `room` and `sem_bin` must be declared as integers and not be manipulated by the program except when making a semaphore call.

To use the semaphores you call either `P()` or `V()` with the proper semaphore. For example:

```
P(room);
```

This will execute the `P()` operation upon semaphore `room`. And

```
V(bin_sem);
```

will execute the `V()` operation upon semaphore `sem_bin`.

The following example solves the dinning philosophers problem using the ezipc.h library.

Example : Dinning Philosophers

```
#include <stdio.h>
#include "ezipc.h"
#define numforks 5      /* number of forks at table */
int eats;               /* number of times each philosopher can eat */
int pid;
int room; /* counting sem that allows only four philosophers in room at one time */
int frk[numforks]; /* array of binary semaphores that represent forks at table */
void philosopher(int i)
{
int x;
for(x=1;x<=eats;x++)
{
/* THINK */
  printf("Philosopher %d thinks\n",i);
  P(room);                                /* try to enter dining hall */
  P(frk[(i+1) % numforks]);               /* wait for right fork */
  P(frk[i]);                              /* wait for left fork */
  /* EAT */
  printf("Philosopher %d eats\n",i);
  V(frk[i]);                              /* release left fork */
  V(frk[(i+1) % numforks]);               /* release right fork */
  V(room);                                /* exit room */
}
}
main() {
int  x, status, signal;
SETUP();
printf(" number of times philosophers can eat?");
scanf("%d",&eats);

/* initialize counting semaphore to control number of philosophers in dining hall */
room = SEMAPHORE(SEM_CNT, 4);

/* initialize binary semaphores that correspond to numforks forks on table */
for (x=0;x<=4;x++)
  frk[x] = SEMAPHORE(SEM_BIN, 1);

/* spawn philosopher processes */
 for (x=0;x<=4;x++){
  pid = fork();
  if (pid == 0) {
    philosopher(x);
    exit(0);
  }
 }
while(wait(&signal) != -1 );        /* wait for child processes */
}
```

Bibliography

1. Bach, M., "The Design of the UNIX Operating System", Prentice Hall.

2. Bic, L., Shaw, A., "OPERATING SYSTEMS PRINCIPLES", Prentice Hall, 2003.

3. Cañas, D., Argenta, C., “Using semaphores and Shared Memory in a Unix System V Teaching Environment”.

4. Gray, John. S., “Interprocess communications in LINUX”, Prentice Hall, 2003.

5. Gray, John. S., “Interprocess communications in UNIX”, Prentice Hall, Second Edition, 1998.

6. Kernighan B., Ritchie D., “The C Programming Language”, Prentice Hall, 1978.

7. Raymond, E., “The Art of UNIX Programming”, Addison Wesley, 2004.

8. Robbins K., Robbins S., UNIX Systems Programming”, Prentice Hall,2003.

9. Rochkind M., Advanced UNIX Programming”, Addison Wesley, Second Edition, 2004.

10. Shapley

11. Stallings W., “Operating Systems: Internals and Design Principles, Prentice Hall, 2009.

12. Stevens.W.R, Rago S., “Advanced Programming in the UNIX Environment”, Addison Wesley, Second Edition, 2005.

13. J.A. Trono, "A new exercise in concurrency," SIGCSE Bulletin, Vol. 26, pp. 8-10, 1994.

14. Andrews, G., “Foundations of Multithreaded, Parallel, and Distributed Processing”, Addison-Wesley, 2000.

15. Powers, D., “Dining Philosophers: Strategies for Success”, International Conference on Parallel and Distributed Processing (PDP05), 2005.