COMP 8567 Advanced Systems Programming

Unix File Input/Output

Summer 2023

Outline

- Introduction
- File Descriptors
- open() system call
- read() system call
- write() system call
- lseek() system call
- **umask()** system call and **umask** command
- Summary

Introduction

Most Unix File I/O can be performed using the system calls :

open(): to open an existing file (also to create a new file)

creat(): to create a new file or rewrite an existing one The **creat()** function is redundant. Its services are also provided by the **open()** function. It has been included primarily for historical purposes

read(): to read a specified number of bytes

write(): to write a specified number of bytes

lseek(): to explicitly position at a file offset

close(): to close a file

In contrast to the std I/O functions, the Unix I/O system calls are unbuffered (All characters are read or written in **one system call** and are not read/written character by character)

File Descriptors

- File descriptors : The kernel refers to any open file by a file descriptor- **a nonnegative integer**
- In particular, the standard input, standard output and standard error have descriptors 0, 1 and 2 reserved for them respectively (The file descriptors 0,1 and 2 **cannot** be allocated to a file created by user/programs)
- The symbolic constants, defined in <unistd.h> for these values are
  - STDIN_FILENO //0
  - STDOUT_FILENO //1
  - STDERR_FILENO // 2

```
#include <stdio.h>
#include <unistd.h>
//ioconst.c
//Print some of the i/o constants defined in unistd.h

main()
{
printf("\n%d",STDIN_FILENO);
printf("\n%d",STDOUT_FILENO);
printf("\n%d",STDERR_FILENO);
}
```

open() system call

- **Synopsis :** int open(const char * *pathname*, int oag , [int mode])
  - [int mode] is used only when a file is newly created using O_CREAT
- Opens the file specified by *pathname* (can be absolute or relative)

Returns the file descriptor if OK, -1 otherwise.

The argument oag is formed by OR'ing (bitwise) together 1 or more of the following constants (in < fcntl.h >)//file control options

- O_RDONLY: Open for reading only
- O_WRONLY: Open for writing only
- O_RDWR: Open for reading and writing only
- O_APPEND: Open for writing after the end of file (For all write operations)
- O_CREAT: Create a file

Note that the third argument, **only** used when a file is created, supplies the file's **initial permission flag settings, as an octal value** (Ex: 0700)

Examples :

if ((d=open(“/home/pranga/chapter4/check.txt", O_RDONLY))==-1)
error_routine();

Both absolute and relative paths (for the filename) can be used

if ((d=open(“check.txt”, O_RDONLY))!=-1)//Continue with file operations;

d=**open(name,O_CREAT| O_RDWR, 0700)**

File Permissions:

User Group Others

RWE RWE RWE

111 000 000 (0700)

In this case, 0700 value for mode provides all rights to the owner of this fille and **no permission to group and others (Depends on umask as well)**

Other Example values for mode :

0400: Allows read by owner

0200: Allows write by owner

0100: Allows execute by owner

0040: Allows read by group

0004: Allows read by others

0777: Allows read/write/execute by all

Check umask and chmod later

```
//open.c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
//Tries to open a file and displays an error if the file cannot be opened in the specified mode
//Prints the file descriptor (a non-negative integer) if open() is successful 1

int main(void)
{
int fd1=open(“new.txt",O_CREAT|O_RDWR,0777);
if(fd1==-1)
printf("\n The operation was not successful\n");
else
printf("\n The file descriptor is %d\n",fd1);
close(fd1);
}
```

read() and write() system calls

read() synopsis:

ssize_t read(int fd, void *buf, size_t nbyte);

Reads as many bytes as it can, possibly up to nbyte, and returns the number of bytes actually read.

ssize_t and size_t are usually defined as **long integers**

Example: long int n= read(fd3,buff1,200); // fd3 is the file descriptor obtained by opening check.txt successfully

The value returned by read() can be :

- -1 : in case of an error
- smaller than nbyte : the number of bytes remaining before the end of file was less than the nbyte specified //Ex: if the no of bytes remaining before the end of the file is 150
- 0 : (if the file exists, but has no characters)

write() synopsis:

ssize_t write(int fd, const void *buf, size_t nbyte);

Example: long int n= write(fd3,buff1,200); //writes the contents of buff1 into check.txt

write returns nbyte if OK and -1 otherwise.

```
//br1.c

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main(void)
{

//Reads from check.txt into an array of characters

int fd3=open("check.txt",O_RDONLY);
char *buff1;
long int n;
n=read(fd3,buff1,30);
printf(“\the number of bytes read is %d\n”, n);

printf("%s", buff1);

close(fd3);
}
```

//bw1.c

```
//Writes an array of characters into check.txt
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main(void)
{

int fd3=open("check.txt",O_RDWR);
char *buff1="Hello";
long int n;

n=write(fd3,buff1,5);

printf("\n\nThe number of bytes written were %ld\n",n);
n=write(fd3,buff1,5);
printf("\n\nThe number of bytes written were %ld\n",n);
n=write(fd3,buff1,5);
printf("\n\nThe number of bytes written were %ld\n",n);
close(fd3);

}
```

//bow.c

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

int main(void)
{
//writes and array of characters (overwrites if already present)
int fd3=open("check.txt",O_RDWR);
char *buff1="********************";
long int n;
n=write(fd3,buff1,20);

printf("\n\nThe number of bytes written were %ld\n",n);
close(fd3);

}
```

close(fd)

Note that **int close(int fd)** frees the file descriptor fd.

close() returns 0 when OK and -1 otherwise. For example, -1 will be returned if fd was already closed.

lseek() system call

- **Synopsis : off_ t lseek(int fd, off_t offset, int whence);**

Returns the **resulting offset** if OK, -1 otherwise.

//Resulting offset is always from the beginning of the file

The return type off_ t is a long integer.

it sets the file pointer(position) associated with the open file descriptor specified by

the file descriptor fd as follows:

- If whence is SEEK SET, the pointer is set to offset bytes //From the beginning of the file
- If whence is SEEK CUR, the pointer is set to its current location plus offset.
- If whence is SEEK END, the pointer is set to the **size of the file** plus offset
  - // Includes the **Coded character set identifier** (CCSID) which is an 8-bit (1 Byte) code for UTF encoding
  - // UTF (Unicode transformation format)

These three constants are defined in < unistd.h >

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

//ls1.c with SEEK_SET

main()
{
int fd3=open("check.txt",O_RDWR);
int long n=lseek(fd3,10,SEEK_SET);
printf("\nThe resulting offset is %d\n",n);
char * buff1="COMP 8567";
n=write(fd3,buff1,9);
printf("\nThe no of bytes written from the resulting offset is
%d\n",n);
close(fd3);
}
```

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

//ls2.c with SEEK_CUR
main() {

int fd3=open("check.txt",O_RDWR);
int long n=lseek(fd3,10,SEEK_SET);
printf("\nThe resulting offset is %d\n",n);
char * buff1="COMP 8567";
n=write(fd3,buff1,9);
printf("\nThe no of bytes written from the resulting offset is
%d\n",n);

//SEEKCUR
n=lseek(fd3,0,SEEK_CUR);
printf("\nThe final offset is %d\n",n);

close(fd3);
}
```

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

//ls3.c with SEEK_END
main()
{
int fd3=open("check.txt",O_RDWR);
int long n=lseek(fd3,10,SEEK_SET);
printf("\nThe resulting offset is %d\n",n);
char * buff1="COMP 8567";

n=write(fd3,buff1,9);
printf("\nThe no of bytes written from the resulting offset is
%d\n",n);

//SEEKCUR
n=lseek(fd3,10,SEEK_CUR);
printf("\nThe resulting offset is %d\n",n);
n=write(fd3,buff1,9);
printf("\nThe no of bytes written from the resulting offset is
%d\n",n);

//SEEKEND

n=lseek(fd3,0,SEEK_END);

printf("\nThe resulting offset is %d\n",n);

n=write(fd3,buff1,9);

printf("\nThe no of bytes written from the
resulting offset is %d\n",n);


close(fd3);

} //end main
```

umask() system call and umask command

- System call umask() and command umask allow the settings of the user mask that controls newly created file permissions.
- Each mask digit is negated, then applied to the file permission/default permission using a **logical AND** operation.

  Ex: If the user has requested the file permission 0666 (110 110 110) in open()

  and If umask is 0022 (000 010 010), permission of the newly created file would be (110 100 100)

  i e //Negation of mask (111 101 101) **AND** (110 110 110) = 110 100 100

- **umask() system call Synopsis:**
- mode t umask(mode t mask);
  - umask() sets the calling process's fille mode creation to (!mask & mode)
    - Ex: if mask is 0055 (000 101 101) and the mode is 0777 (111 111 111) , the new file would be created with permission 0755
      (111 101 101) 111 010 010 111 010 010 0722
- Need header files:
  - <sys/types.h> and <sys/stat.h>

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

//umaskex.c
main()
{


int fd1=open("check24.txt",O_CREAT|O_RDWR,0777);


}
```

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

//umaskex1.c
main()
{

umask(0000); //system call within a program, overrides the previously set mask (in the command line)
int fd1=open("check24.txt",O_CREAT|O_RDWR,0777);



}
```

umask –Linux command

- Command umask does a simlar job as the system call umask()
- Synopsis: umask [-S] [mask]
- When option -S is present, accept symbolic representation of mask // **$umask -S**
- When no mask is provided, umask returns the current user mask.
- Examples: umask -S g+w: allows write permission for my group, if requested.
- umask 0000: makes your mask neutral
- umask 0077: no permission for your group and others.
- umask acts as a safety measure that disables some permissions when files are created (**however, they can be changed later using chmod**)

Sample chmod and umask commands

- $ chmod g+w check24.txt
- $ chmod g-w check24.txt
- $ umask -S u-w
- $ umask -S u+w
- $ umask -S g+w

Summary

- Introduction
- File Descriptors
- open() system call
- read() system call
- write() system call
- lseek() system call
- **umask()** system call and **umask** command

COMP 8567 Advanced Systems Programming

Advanced C Programming Techniques
(Pointers and Functions)

Summer 2023

Outline

- Introduction
- Pointers
- Manipulation of pointers
- The special pointer NULL
- Pointer arithmetic + and –
- Pointers and arrays
- The pointer-to void type
- Dynamic Memory Allocation
- Passing Parameters by value and reference
- Passing Functions as Parameters
- Summary and Conclusion

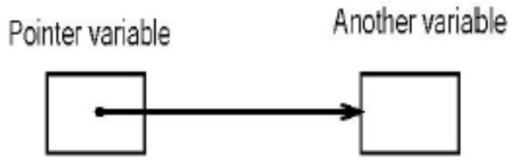
Introduction

Pointers are crucial to C, and they are mainly used to:

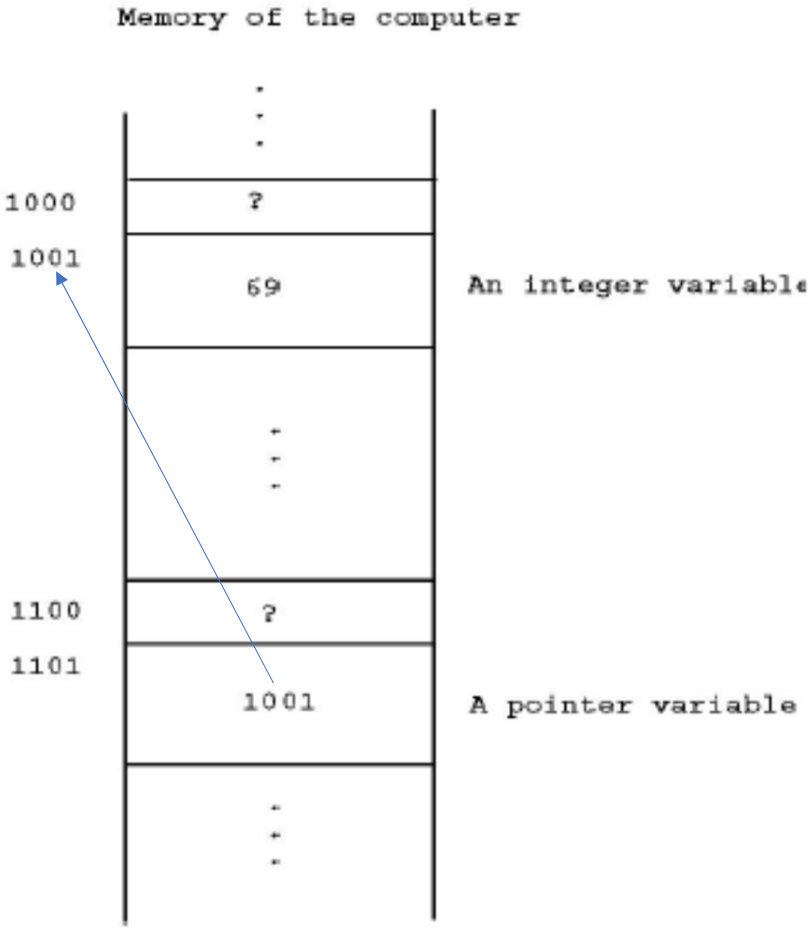
- Provide the means by which functions can modify their **calling arguments.**
- Support the **dynamic allocation** of memory.
- Refer to **a large data structure** in a simple manner.
- Support data structures such as **linked lists.**

Pointers are Addresses

- A pointer is a **variable** whose value is a memory addresses.
- A pointer variable is similar to other variables
  - It requires memory and stores a value.
- However, the purpose of a pointer is special.
- A data variable stores data (e.g. age).
- A pointer variable stores memory addresses (of variables)



A pointer contains/stores the **memory address** of another variable (hence it **points** to another variable)



- Syntax: ***base type *pointer name*;**
- where base type defines the type of variable the pointer can point to.
- Example : **int *ptr; float * p1; int *p2;**
  - ptr is a pointer to an integer variable.
  - p1 is a pointer to a float variable

Pointers- Simple Example //prelim.c

```
# include <stdio.h>

int main(void)
{

int *a; //a is a pointer variable to an integer
int b=100;
a=&b; // pointer variable a is assigned the address of b
*a=200; //dereferencing

printf("\nThe address of b is %d", &b);
printf("\nThe value of a is %d", a);

printf("\nThe value of b is %d", *a);

}
```

Manipulation of Pointers

Operators:

& : a unary operator that returns the address of its operand.

int a=10;

int *ptr; //ptr is a pointer to an integer

ptr=&a; //ptr now contains the address of the integer variable a

* : A unary operator used to **dereference** its operand (a pointer)

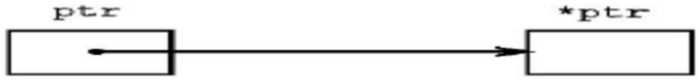
int a=10;

int *ptr;

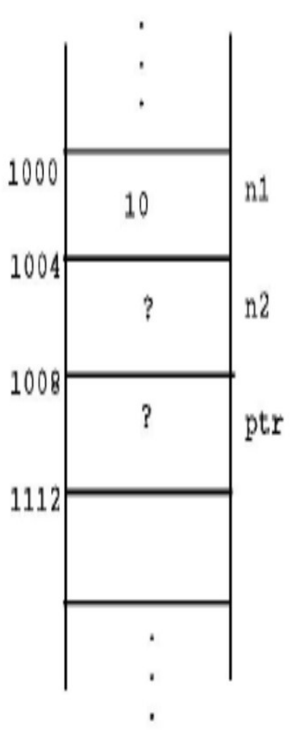
ptr=&a;

***ptr** refers to the **value of the variable** referenced by ptr (i.e variable **a**)

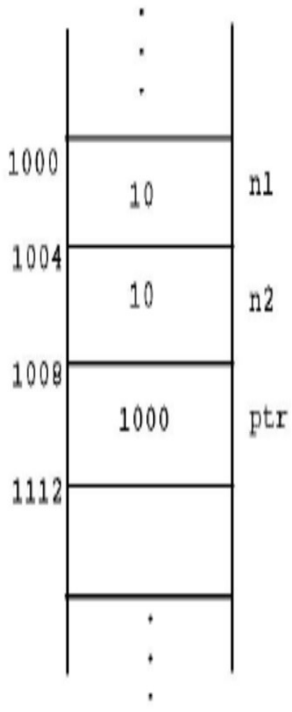
printf("%d",*ptr); //output =10



```
#include <stdio.h>
//ex1.c
int main(void) {

int n1=10, n2;   // 2 variables of type integer

int *ptr;       //a pointer to integer

ptr = &n1;      // ptr = address of num1

n2 = *ptr;    //n2 = value stored at the address pointed by ptr, i.e value of n1

printf("\n n1:%d  n2:%d  *ptr:%d \n\n", n1, n2, *ptr);

return(0);
}
```



Decalaration of n1, n2 and ptr



The contents of n1, n2, and ptr before the end

Pointer Declaration and Assignments: Examples:

```
float z;
int x, y;
int *ptr1 = &x  // Like any variable, pointers can be initialized at declaration time with the address of a compatible variable, but not with a constant
int *ptr2 = &y;
```

The syntax of the following assignments is correct:

```
ptr2 = ptr1;
        /* ptr2 receives the contents of ptr1 which is the address of x in that case */
*ptr1 = *ptr2;  // this is equivalent to x=y
*ptr2 = 4;      // this is equivalent to y=4
x = *ptr2;      // this is equivalent to x=y
```

However, the syntax of the following assignments is not correct:

```
ptr1 = x;       //  type mismatch: assigning a data value (integer) to a pointer
y = ptr1;       // type mismatch: assigning an address to an integer variable
ptr1=&z;        //  this is wrong because ptr1 is supposed to point to an integer variable and z is declared as float.
```

Special Pointer NULL //nullp.c

We sometimes need to indicate that a pointer is not pointing to any valid data. For that purpose, the **constant NULL** is defined in **stdlib.h** and can be used in a C program.

A common use of NULL: When a function returning a pointer wants to indicate a failure operation, the function can return NULL to specifically indicate a failure

Example :

```
int *afunction();      /* a function that returns a pointer to an integer */
int *ptr;                    /* a pointer to integer */
ptr = afunction();
if (ptr == NULL)           /* afunction() was unsuccessful */
{take-appropriate-action}
```

Pointer Arithmetic

When a pointer is incremented (or decremented), it will point to the memory location of the next (or previous) element of its base type.

Example :

Assume that ptr, a pointer to int, contains the address 200 and n is an integer.

Also assume that the size of an integer is 4 bytes.

ptr+1 will contain the address 200 + 4=204.

ptr- 1 will contain the address 200 - 4=196.

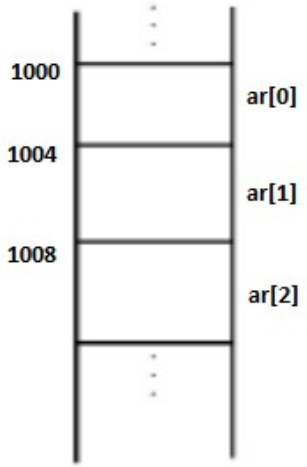
ptr = ptr + n ptr will contain the address 200 + nx4. //ptr=ptr+2 = ptr+2x4= 208

ptr = ptr - n ptr will contain the addres 200 - nx4. //ptr=ptr-2=ptr-2x4=192

```
//ex3.c
#include <stdio.h>
int main(void)
{
        int a=100;
        int *p=&a;
        printf("\nThe address of p1 %d\n",p);
        p=p+1;
        printf("\nThe address of  p1+1  is %d\n",p);
        p=p-1;
        printf("\nThe address of  p1-1   is %d\n",p);
        p=p+2;
        printf("\nThe address of  p1+2  is %d\n",p);
        p=p-2;
        printf("\nThe address of  p1-2  is %d\n",p);
}
```

Pointers and Arrays

- Pointers and arrays are closely related.
- In particular, the name of an array is the address of the array's first element.
- For example, the declaration **int ar[3]** reserves 3 consecutive units of memory. Each unit has the same size, size of an integer (e.g., 4 bytes).



&ar[0] = 1000, &ar[1] = 1004, &ar[2] = 1008.
In general, &ar[i] = ar + i 4(size of integer).
ar corresponds to the address 1000.
ar is
a pointer except that it is a constant.

//ex9.c

Pointers and arrays are interchangeable!

Consider the example :

```
main(){
int ar[4] = {5, 10, 15, 20};
int *ptr;
int i;
for (i=0; i<4; i++)
printf(``%d ``, ar[i]);

ptr=ar;   /* Equivalent to ptr=&ar[0] */
for (i=0; i<4; i++)
printf(``%d ``, ptr[i]);
}
```

Important : Because the name of an array is a constant pointer, it is illegal to make the following assignment ar =ptr (ptr=ar is fine!)

```
#include <stdio.h> //ex10.c
int main(void){
int ar[4] = {5, 10, 15, 20};
int *ptr; int i;

for (i=0; i<4; i++) {
printf("Using dereferencing %d\n", *(ar+i)); //Print the value of each array element using deferefencing
printf("Using array index %d\n", ar[i]);
}

//Print the values of the array by assigning ptr=ar using dereferencing
ptr=ar; /* Equivalent to ptr=&ar[0] */
for (i=0; i<4; i++){
printf("Using dereferencing %d\n", *(ptr+i));
printf("Using array index %d\n", ptr[i]);
}}
```

Limitations of Pointer Arithmetic

- * , /, and % cannot be used with pointers.
- + and - are restricted : only **integer offsets** can be added to or subtracted from a pointer.
  - ptr1=ptr1+4;
  - ptr2=ptr2-3
- Two pointers **cannot** be added together
  - ptr3=ptr2+ptr2
- A pointer p1 can be subtracted from another pointer p2. The result is an integer, the number of elements(not bytes or addresses) between p1 and p2 (**offset**)
  - p1=p2-p3

  Valid:
  p1=p1-4
  p2=p2-3
  p3=p1-p2

```
//arrdiff.c
#include <stdio.h>
int main(void){
int ar[100]; int *p1, *p2;
p1=ar; p2=&ar[60];
printf("p2-p1(offset): %d\n", p2 - p1);
p1=&ar[30]; p2=&ar[80];
printf("p2-p1(offset): %d\n", p2 - p1);
}
```

Note: If p1 and p2 are pointers to integers containing addresses 300 and 400 respectively, p2 - p1 will be equal to (400 - 300)/4 = 25 assuming 4 bytes for the size of an integer.

Recap

- Introduction ✓
- Pointers ✓
- Manipulation of Pointers ✓
- The Special pointer NULL ✓
- Pointer arithmetic + and – ✓
- Pointers and Arrays ✓
- The pointer-to void type
- Dynamic Memory Allocation
- Passing Parameters by value and Reference
- Passing Functions as Parameters

The Pointer to Void Type

- A pointer variable ptr defined as **void * ptr** is a generic pointer variable (it can point to any type)
- Advantage : a pointer to void may be converted, without a cast operation, to a pointer to another data type.
- void *Malloc() //Pointer to Void

```
void *myfunction(); /* a fn that returns a generic ptr
int n1;
int *ptr1;
float n2;
float *ptr2;
ptr2 = &n1; /* Error */
ptr1 = &n1; /* OK */
ptr2 = &n2; /* OK */
ptr1 = myfunction(); // OK 'void *' is casted to 'int *' */
ptr2 = myfunction(); // OK, 'void *' is casted to 'float *’
```

Important : Pointers to void **cannot be dereferenced**

Example :

```
int n;
void *ptr=&n;
*ptr = 25;   /* Error */

void *ptr;
int a=200;
ptr=&a;
int *b;
b=ptr;
```

```
# include <stdio.h>
//void.c
main()
{
void * ptr1;
int a=100;
ptr1=&a;
printf("\n The current address of the generic pointer ptr1 is %d\n", ptr1);

float b=20.32;

ptr1= &b;
printf("\n The current address of the generic pointer ptr1 is %d\n", ptr1);

//printf("%f",*ptr1); //generic pointers cannot be dereferenced

/*
float *temp;
temp =ptr1;
printf("%f",*temp);
*/
}
```

Dynamic Memory Allocation

There are 3 types of memory allocation in C:

Static allocation : a variable's memory is allocated and persists throughout the entire life of the program. This is the case of **global variables**.

Automatic allocation : When local variables are **declared inside a function**, the space for these variables is allocated when the function is called (starts) and is freed when the function terminates. This is also the case of parametric variables.

Dynamic allocation :

Allows a program at the execution time to allocate memory **when needed** and to **free it** when it is no longer needed. Advantage : it is often impossible to know, prior to the execution time, the size of memory needed in many cases. For example, the size of an array based on any input size n

- Through its standard library (include stdlib.h), C provides functions for allocating new memory from the heap (available unused storage).
- The most commonly used functions for managing dynamic memory are :
  - **void * malloc(int size)** : to allocate a block (number of bytes) of memory of a given **size** and returns a pointer to this newly allocated block.
  - **void free(void *ptr)** : to free a previously allocated block of memory.
- Note: **sizeof** is an operator often used with malloc. It returns the size in bytes of its operand(a data type name or a variable name). For instance, sizeof(char)=1 //so.c

Example : ex8.c

```
main(){
int *ptr1;
float *ptr2;

ptr1 = malloc (sizeof(int)); /* allocate space for an integer */

ptr2 = malloc (sizeof(float)); /* allocate space for a float */

*ptr1 = 20;
*ptr2 =13.5;
free(ptr1); /* free previously allocated space */
free(ptr2);
}
```

Dynamic Arrays

When the size of an array is not known before the execution time, allocating arrays dynamically is a good solution.

The steps for creating a dynamic array are :

1. Declare a pointer with an appropriate base type.

2. Call malloc to allocate memory for the elements of the array. The argument of malloc is equal to the **desired size of the array multiplied by the size in bytes of each element of the array**.

3. Assign the result of malloc to the pointer variable.

```
#include <stdio.h>
#include <stdlib.h> //dynarr.c
int main(void) {
int *ar,n;
printf("\nEnter the number of elements in the array\n");
scanf("%d",&n);
ar=malloc(n*sizeof(int));

                for(int i=0;i<n;i++)
                {
                printf("Enter element %d\n", i);
                scanf("%d",&ar[i]);
                }
                printf("\nThe elements of the array are\n");
                for(int i=0;i<n;i++)
                {
                printf("\n%d",ar[i]);
                }
free(ar);
//End dynarr.c
}
```

Declared Array vs. Dynamic Array

Declared arrays vs. dynamic arrays

- The memory associated with a declared array is allocated automatically when the function containing the declaration is called whereas, the memory associated with a dynamic array is not allocated until malloc is called.
- The size of a declared array is a constant and must be known prior to the execution time whereas, the size of a dynamic array can be of any size and need not to be known in advance.

IMPORTANT: If the heap runs out of space, malloc will return a NULL instead of a pointer to void

```
ar = malloc(n * sizeof(float));
if (ar == NULL ) // malloc has failed  take-appropriate-action and suitable message has to be displayed
```

Passing Parameters by Address (Reference)

- One of the most common application of pointers in C is their use as function parameters, making it possible for a function to get the **address of actual parameters**.
- When an argument (parameter) is passed by address to a function, the latter receives the **address** of the actual argument which is passed as a **pointer from the calling function.**
- Passing by values creates **copies of the variables** and results in wastage of space and processing power
- **Pointers can be directly dereferenced and manipulated as required**
- Passing by reference is more efficient in terms of memory and speed.

Example: Passing by Reference

```
#include <stdio.h>
//pbyref.c

void swapnum(int *i, int *j) {
  int temp = *i;
  *i = *j;
  *j = temp;
}

int main(void) {
  int a = 10;
  int b = 20;
  swapnum(&a, &b);
  printf("\nPassing by reference\n");
  printf("After swapping, a is %d and b is %d\n", a, b);
  return 0;
}
```

Passing Functions as Parameters

- C allows functions to be passed as arguments.
- How to declare a pointer to functions?

Some Examples:

- Pointer to a function with no parameters and not returning anything.
  - void (*ptrFunc)();
- Pointer to a function with one integer parameter and returning a value **double**
  - double (*ptrFunc)(int)
  - ptrFunc=abc;
- Pointer to a function with two parameters (int, float) and returning a value **int**
  - int (*ptrFunc)(int,float);

```
#include <stdio.h> //ptof.c

int add(int a,int b)
{
return a+b;
}

int subtract(int a,int b)
{
return a-b;
}

int main(void)
{
int (*f1)(int, int);

f1=add;  //the address of the add function is assigned to pointer f1
int retvalue=f1(10,20);

printf("\nThe return value is %d\n",retvalue);

f1=subtract; //the address of the add function is assigned to pointer f1

retvalue=f1(10,20);
printf("\nThe return value is %d\n",retvalue);
}
```

```
#include <stdio.h>
int print1()
{
   printf("\nHello World from print1!\n");
   return 0;
}
int print2()
{
   printf("\nHello World from print2!\n");
   return 0;
}
void helloworld(int (*fn)(),int (*fn1)())
{
fn();
 fn1();
}
```

```
//fap.c
int main(void)
{
   helloworld(print1,print2);
   return (0);
}
```

- // fap9.c

Recap and Summary

- Introduction ✓
- Pointers ✓
- Manipulation of Pointers ✓
- The Special pointer NULL ✓
- Pointer arithmetic + and – ✓
- Pointers and Arrays ✓
- The pointer-to void type ✓
- Dynamic Memory Allocation ✓
- Passing Parameters by value and Reference ✓
- Passing Functions are Parameters ✓
- Conclusion

THANK YOU

COMP 8567

Advanced Systems Programming

Introduction

Summer 2023

Dr. Prashanth Ranga

Where is Linux used?

- **Webservers**

  Linux is used to power **96.3%** of the world's top 1 million web servers. Windows (1.9%), and FreeBSD 1.8%) are the other players: https://www.enterpriseappstoday.com/stats/linux-statistics.html
- Single Board Computers (**Raspberry Pie**)
- **Android** uses a modified version of the Linux Kernel
- **macOS** is based on a BSD Unix kernel known as Darwin which is open-source.
- **iOS** is a variant of Darwin, derived from BSD, a UNIX-like kernel
- **Supercomputers**

  https://en.wikipedia.org/wiki/TOP500

  List of the fastest supercomputers in the world.

COMP 8567 – List of Topics

1. Introduction to UNIX
2. Advanced C programming techniques
3. Unix Input/Output
4. Process Management and Control
5. Signals
6. Inter Process Communication (Pipes)
7. Unix Shells
8. Multithreading
9. Client-Server applications

Introduction to Unix –Outline

- Connecting to the Remote Linux Desktop
- The Role of Operating Systems
- Architecture of the Unix Operating System
- GNU
- History of Unix
- Other implementations of UNIX
- Shells
- **File System**
- **Input Output**
- **Process and Process ID**
- **Signals**
- **Pipes**
- **System Calls and Library Functions**
- Conclusion and Summary

Remote Desktop

Remote Desktop via a Web Browser

http://nx.cs.uwindsor.ca

Since the School of Computer Science has a NoMachine Enterprise license, the **cs.uwindsor.ca** remote desktop can be accessed directly from the **browser** through the link provided!

This method of remote access requires neither the NoMachine client nor the VPN client.

Download the App (Recommended)

https://www.nomachine.com/download/download&id=8

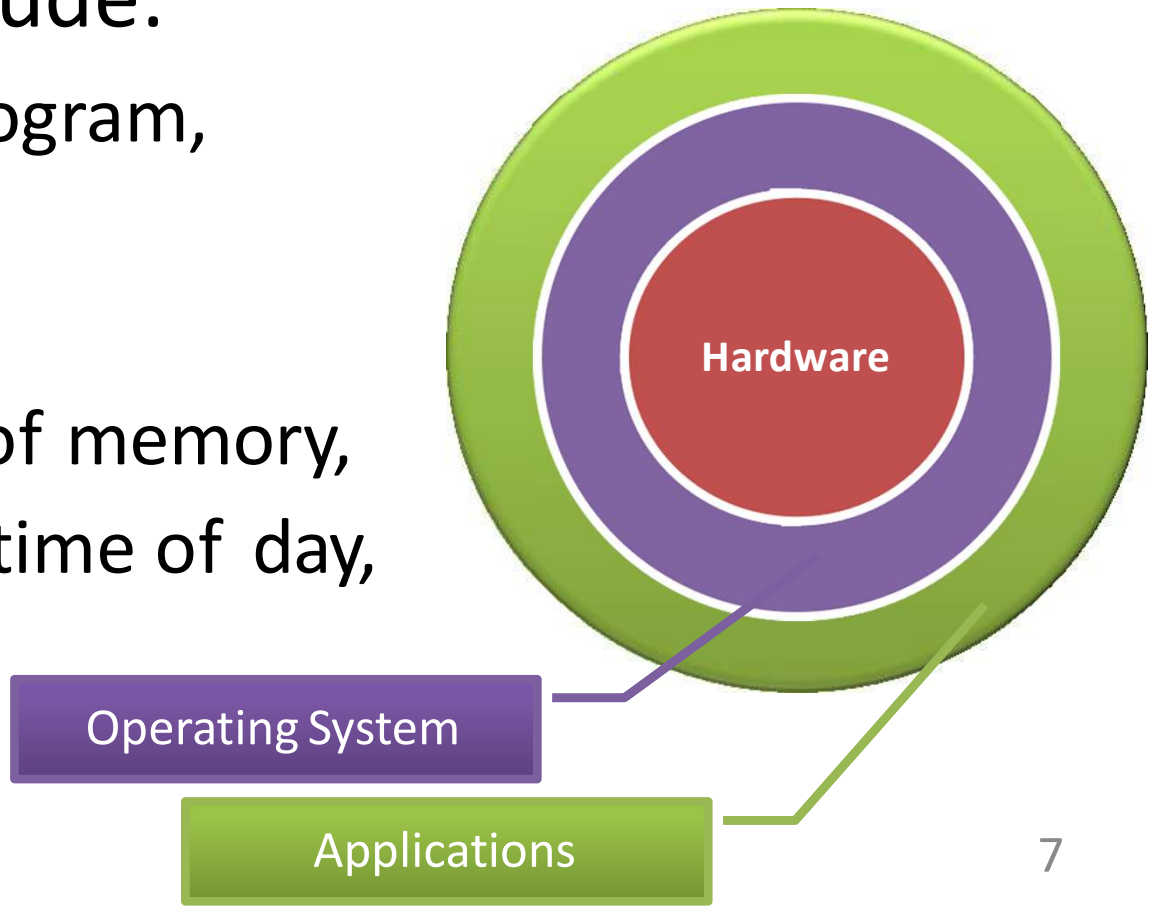
Sample C Program (welcome.c)

```
# include <stdio.h>  //welcome.c

main()
{

        printf("\nWelcome to COMP 8567\n");

}
```

```
$ gcc -o welcome welcome.c

$ ./welcome
```

Role of Operating Systems

- All operating systems provide services for programs they run.
- Typical services include:
  - executing a new program,
  - opening a file,
  - reading a file,
  - allocating a region of memory,
  - getting the current time of day,
  - and so on.

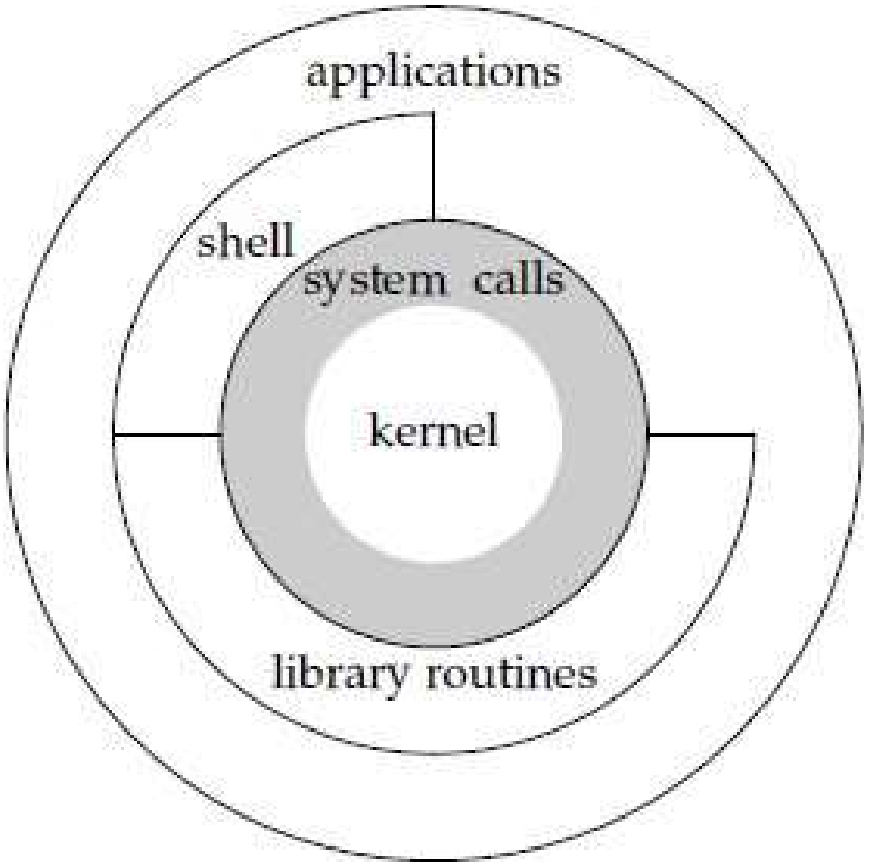


Major Software Components of OS

- Process Management
- Memory Management (Primary Memory)
- File Management (Secondary Memory)
- I/O Management
- Network Management
- Security Management

Architecture of the UNIX operating system

- The core of the UNIX OS is called the ***kernel.***
- The ***kernel*** directly interacts with the hardware and provides services to the applications
- The interface to the kernel is a layer of software called the ***system calls*** .
  - A system call is the programmatic way in which a computer program requests a service from the kernel of the operating system on which it is executed[1]
- **Libraries** of common functions are built on top of the system call interface, but applications are free to use both.
- The **shell** is a special application that provides an interface for running other applications.



For example, **Linux** is the kernel used by the GNU operating system.

History of Unix

- 1969-71 AT&T Bell Labs
- MULTICS: Multiplexed Information and Computing Service
  - Mainframe-TimeSharing- MULTICS was used to mange Time-sharing
- AT&T Stopped the project
- Side Project was developed by the employees who were interested
- MULTICS became a single user system UNICS (Uniplexed information and Computing Service)
  - UNICS was modified to support multiple users and eventually became Unix (x for multiple users)
- 1972: Unix was rewritten in C
  - Unix became portable to other hardware

History..

- 1975 –US govt bans AT&T from making any software
- AT& T gave away free software (Unix) to almost anyone who requested for it -Universities, Govt agencies and Corporations
- A number of mainframes were now running on Unix

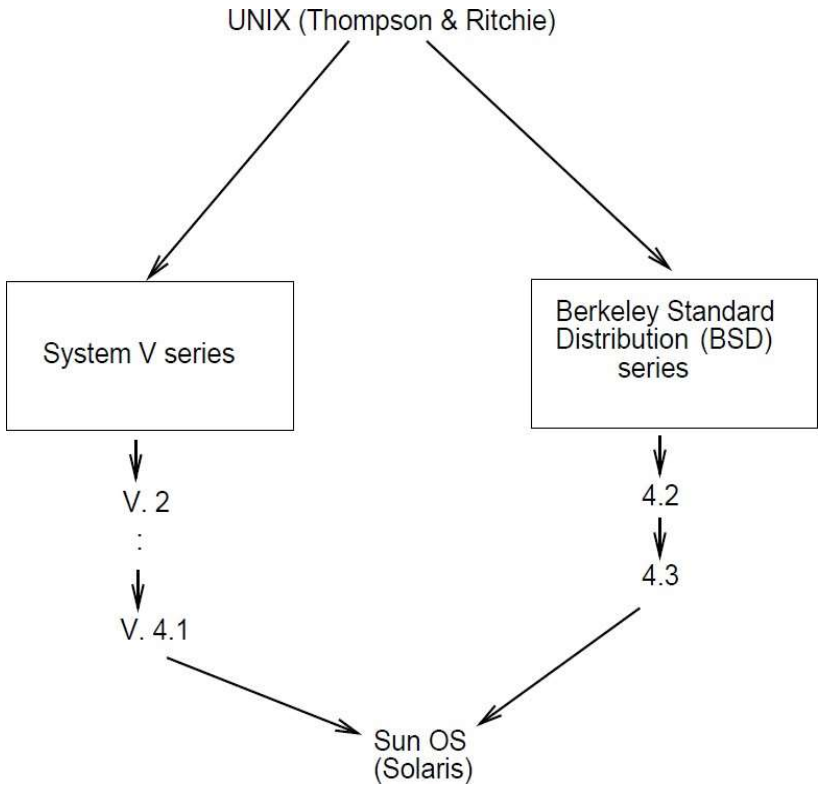
- First public version in 1975 is referred to as system 5 Open Source BSD, Linux

- Closed Source (Solaris, AIX etc) Proprietary HPUX

- Mixed Source (Mac OS / Darwin)
  IOS, ANDROID

Classic versions

- Two (classic) popular versions of Unix :
  - BSD (Berkeley Standard Distribution) Unix: introduction of Socket (Academics-Oriented)
  - System V: from Bell Laboratories (Industry-Oriented)



Other implementations of Unix

- **FreeBSD:** post BSD line after UCB decided to end work on BSD versions. Both binary and sources of the FreeBSD are freely available
- **Linux:** Freely available under GNU public license. Created in 1991, by **Linus Torvalds** to replace Minix
  - Became extremely popular(E.g., Debian, Ubuntu), in particular
    - **it is widely used on servers and mainframe computers**
    - **Android is built on top of the Linux kernel**

Where is Linux used?

- **Webservers**

  Linux is used to power **96.3%** of the world's top 1 million web servers. Windows (1.9%), and FreeBSD 1.8%) are the other players: https://www.enterpriseappstoday.com/stats/linux-statistics.html
- Single Board Computers (**Raspberry Pie**)
- **Android** uses a modified version of the Linux Kernel
- **macOS** is based on a BSD Unix kernel known as Darwin which is open-source.
- **iOS** is a variant of Darwin, derived from BSD, a UNIX-like kernel
- **Supercomputers**

  https://en.wikipedia.org/wiki/TOP500

  List of the fastest supercomputers in the world.

Other implementations of Unix..

- **Mac OS X:** a set of Unix-based operating systems with a graphical interface.
  - The core operating system is called **Darwin**.
  - Based on the FreeBSD OS.
- **IOS,** the mobile OS for iPhone/iPod/iPad and Apple TV, is based on Darwin with many similarities to Mac OS X.
- **Yosemite** is the later version of Mac OS X (10.10), released in Oct. 2014.
- **Android :** Uses Linux Kernel

Other implementations of Unix..

- **AIX**: (Advanced Interactive eXecutive), IBM's own version of Unix
- **HP-UX**: Hewlett-Packard Unix.

Shells

- A ***shell*** is a command-line interpreter that reads user input and executes commands.
- The user input to a shell is normally from the **terminal** (an interactive shell) or sometimes from a file (called a ***shell script***).

Name	Path	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
Bourne shell	`/bin/sh`	•	•	copy of `bash`	•
Bourne-again shell	`/bin/bash`	optional	•	•	•
C shell	`/bin/csh`	link to `tcsh`	optional	link to `tcsh`	•
Korn shell	`/bin/ksh`	optional	optional	•	•
TENEX C shell	`/bin/tcsh`	•	optional	•	•

Figure 1.2 Common shells used on UNIX systems

//Current Shell

$ echo $SHELL

//Various shells supported by the system

$ cd /

$ cd bin

$ ls *sh -1

//Get the Version of Linux

$ uname –a

FILE SYSTEM

File System

- The UNIX file system is a hierarchical arrangement of **directories and files.**
- Everything starts in the directory called ***root***, whose name is the single character /
- A ***directory*** is a file that contains directory entries. Logically, we can think of each directory entry as containing a filename along with a structure of information describing the attributes of files and other directories.
- The attributes of a file are such things as the type of file (regular file, directory), the size of the file, the owner of the file, permissions for the file (whether other users may access this file), and when the file was last modified.
- The `stat` function returns information containing all the attributes of a file.
- $ cd / $ cd home $ cd pranga
- Home directory is denoted by the ~ symbol
- **$ stat sample.c**

Filename

- The only two characters that **cannot** appear in a filename are the slash character (/) and the null character.
- Characters are **CASE-SENSITIVE**
- The slash separates the filenames that form a pathname and the null character terminates a pathname.
- For portability, POSIX.1**(Portable Operating System Interface-IEEE standard)** recommends restricting filenames to consist of the following characters: letters (a-z, A-Z), numbers (0-9), period (.), dash (-), and underscore (_).
- Two filenames are automatically created whenever a new directory is created: .(called *dot*) and .. (called *dot-dot*).
- **$ cd ..**
- **$ cd .**
- Dot refers to the current directory, and dot-dot refers to the parent directory. In the root directory, dot-dot is the same as dot.

Pathname

- A sequence of one or more filenames, separated by slashes and optionally starting with a slash, forms a *pathname*.
- A pathname that begins with a slash is called an ***absolute pathname***; otherwise, it's called a ***relative pathname***. Relative pathnames refer to files relative to the current directory.
- The name for the root of the file system (/) is a special-case absolute pathname that has no filename component.

Absolute and Relative Pathnames and Other file operations

$ cd Music

$ cd /home/pranga/Music

//Moving files from one folder to another
// cp, mv and rm

$ cp hello.c Music // Keeps the original file after copying
$ mv hello.c Music //Removes/deletes the original file after moving
$ cd Music
$ rm hello.c //Permanently deletes the original file
//Rename hello.c to helloworld.c
$ mv hello.c helloworld.c

Working Directory

- Every process has a *working directory*, sometimes called the *current working directory*.

  `pwd` //**`Print Working Directory`**

- This is the directory from which all relative pathnames are interpreted.

Home Directory

- When we log in, the working directory is set to our *home directory*.
- Our home directory is obtained from our entry in the password file

```
cd
cd ~
```

INPUT AND OUTPUT

Input and Output

- **Standard Input, Standard Output, and Standard Error**
- By convention, all shells open three descriptors whenever a new program is run: standard input, standard output, and standard error.
- If nothing special is done, as in the simple command `ls` then all three are connected to the terminal.
- Most shells provide a way to redirect any or allof these three descriptors to any file.
- For example, `ls > listing.txt` executes the ls command with its standard output redirected to the file named listing.txt
- (use < for input redirection, >> for concatenation of output)

PIPES

Pipes

- **Pipes** : a mechanism that allows the user to specify that the output of one program is to be used as the input of another program.
  - several programs can be connected in this fashion to make a pipeline of data owing from the first process through the last one.

Example :

```
Ls -1 | grep .c | wc -w
```

PROCESSES AND PROCESS ID

Processes and Process ID

- An executing instance of a program is called a ***process***. (Some operating systems use the term *task* to refer to a program that is being executed).
- The UNIX System guarantees that every process has a unique numeric identifier called the ***process ID***. The process ID is always a non-negative integer.
- Try ps(Process Status)
- Every process has a **process id**, **parent process** id and **group id**.

Process ID, Parent ID, Group ID

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


// processids.c
//process id, parent id, process group id
        main(void)
        {

                printf("\n The current process id is %d \n",getpid());

                printf("\n The parent id of the current process is %d \n",getppid());

                printf("\n The group id of the current process is %d \n",getgid());

        }
```

Three Important Process- Related Operations

- Fork() //Used to create a new child process
- Exec() //Used to replace a child process with a new process
- Waitpid() // Used to wait for a process to complete execution

User ID

- The *user ID* from our entry in the password file is a numeric value that identifies us to the system.
- This user ID is assigned by the **system administrator** when our login name is assigned, and we cannot change it.
- The user ID is unique for every user.
- The kernel uses the user ID to check whether we have the appropriate permissions to perform certain operations.
- We call the user whose user ID is 0 as the ***superuser*.** The entry in the password file normally has a login name of root, and we refer to the special privileges of this user as **superuser privileges**.

A simple C program that prints the userid using the getuid() system call

```
// userid.c
//uses getuid() system call

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

        main(void)
        {
                printf("\nThe current userid is %d\n",getuid());
        }
```

SIGNALS

Signals

- Signals (Software Signals) are used to notify a process of the occurrence of some condition.
- For example, the following generate signals :
- A division by zero : the signal **SIGFPE** is sent to the responsible process that has three choices. Ignore the signal, terminate the process or, call a function to handle the situation.
- The Control-C key : when pressed, it generates a signal that causes the process receiving it to interrupt.
- Calling the function *kill* : a process can send a signal to another process causing its death. This is an example where Unix checks our permissions before allowing the signal to be sent.
  - A number of signals can be used along with the ***kill*** command: $ kill –l

Kill Processes – Example

Run Multiple Processes and Kill Processes

$ ps –u
$ kill SIGKILL *Processid*

$Kill –l *//List of all signals*

SYSTEM CALLS AND LIBRARY FUNCTIONS

System calls and library functions

- “In computing, a system call is the programmatic way in which a computer program **requests a service from the kernel** of the operating system on which it is executed” [1]
- Each system call in Unix has an **interface function**, in the C standard library, with the same name that the user process invokes.
- The interface function then invokes the appropriate kernel service, using whatever technique is required on the system.
- An interface function for a system call cannot be re-written/overridden by the user
- For our purpose, a system call will be viewed as a **regular C function.**

System Calls- Common Examples

Open()
Read()
Write()
Close()
Fork()
Exec()
Exit()
Getpid()
Getgid()
Getuid()
--
--
--

A simple C program that prints the current process id using the getpid() system call

```
// pid.c
//uses getpid() system call

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

        main(void)
        {
                printf("\nThe current process id is%d\n",getpid());
        }
```

System calls and library functions

- The functions which are a part of standard C library are known as Library functions: strcmp(), strlen() etc are library functions.
- Note that a user process can invoke either a system call or a library function.
- A library function might invoke a system call.

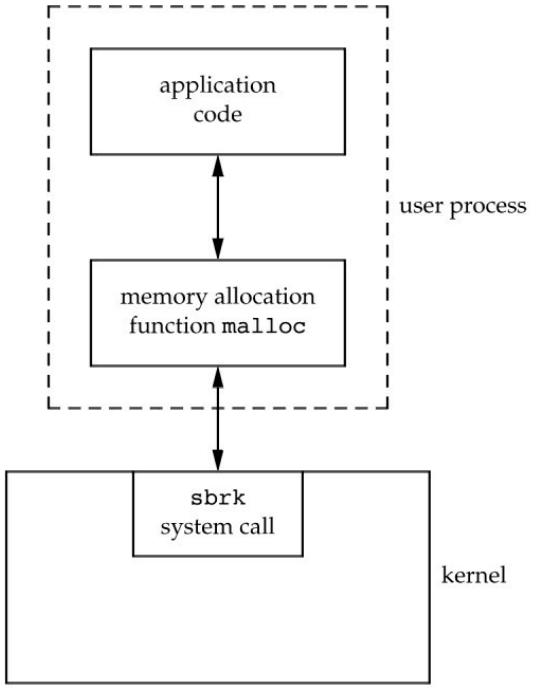


Figure 1.11 Separation of `malloc` function and `sbrk` system call

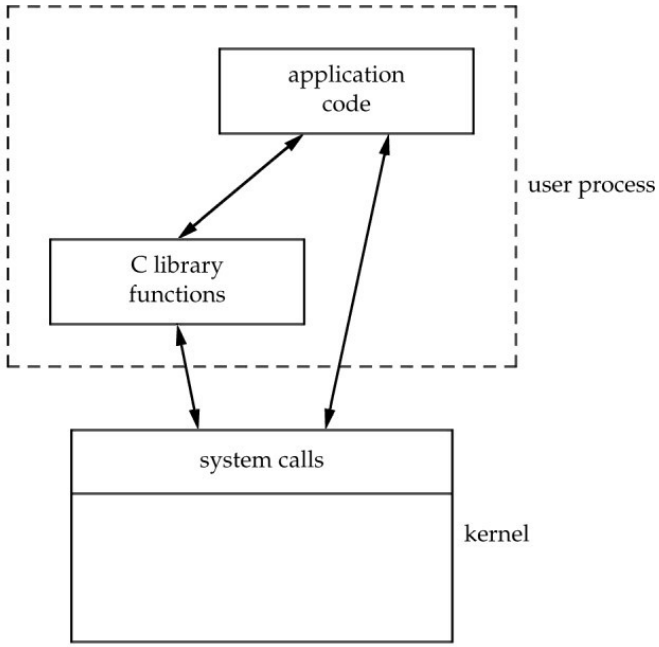


Figure 1.12 Difference between C library functions and system calls

Summary

- Operating system: Software to manage computer resources, in particular,
  - it runs a program for a user
  - it allows communication with devices and processes
- A program is a le containing instructions
- A process is a program being executed
- Unix is a multiuser operating system
- Most of Unix is written in the C language
- Unix has a simple philosophy: a program should do one thing and do it well.
- Entry points in Unix are called system calls. They allow the user to get services from the kernel.

LinkedIn Learning (Lab 1)
Due: May/16/2023

Unix Essential Training:

https://www.linkedin.com/learning-login/share?account=2212217&forceAccount=false&redirect=https%3A%2F%2Fwww.linkedin.com%2Flearning%2Funix-essential-training%3Ftrk%3Dshare_ent_url%26shareId%3DnHIsKhovToqnR9nuQigrsQ%253D%253D

APPENDIX

Linux

6.3.1 (Apr/30/2023 –SR

6.4-rc1 (May/07/2023 –PR)

version 11 (Debian)
The current stable distribution of Debian is version 11, codenamed bullseye. It was initially released as version 11.0 on August 14th, 2021 and its latest update, version 11.7, was released on April 29th, 2023. {debian.org)

COMP 8567

Advanced Systems Programming

Process Control

Outline

- Unix Processes
- Creating a new process: **fork()**
- Terminating a process: **exit()**
- Waiting for a process: **wait()**
- Waiting for a specific process: **waitpid()**
- Orphan and Zombie Processes
- **exec()**
- Changing Directory : **chdir()**
- Summary

Unix Processes

- Unix is a **multiuser** and **multitasking** operating system
  - **Multiple users** can run their programs concurrently and share hardware resources
  - An **user can run multiple programs** and tasks concurrently
- It appears that the execution is done in parallel, however in reality the OS switches between multiple users and processes rapidly (back and forth) in an **interleaved** manner
- **Unix Processes:**
  - An executing program is a process.
  - ex1.c (program)-> ex1 (executable)->./ex1 (process)
- **Every process in Unix has the following :**
  - A unique process ID (PID)
  - Some code : instructions that are being executed
  - Some data : variables (typically)
  - A stack : a form of memory where it is possible to push and pop variables/data
  - An environment : CPU registers' contents, tables of open files

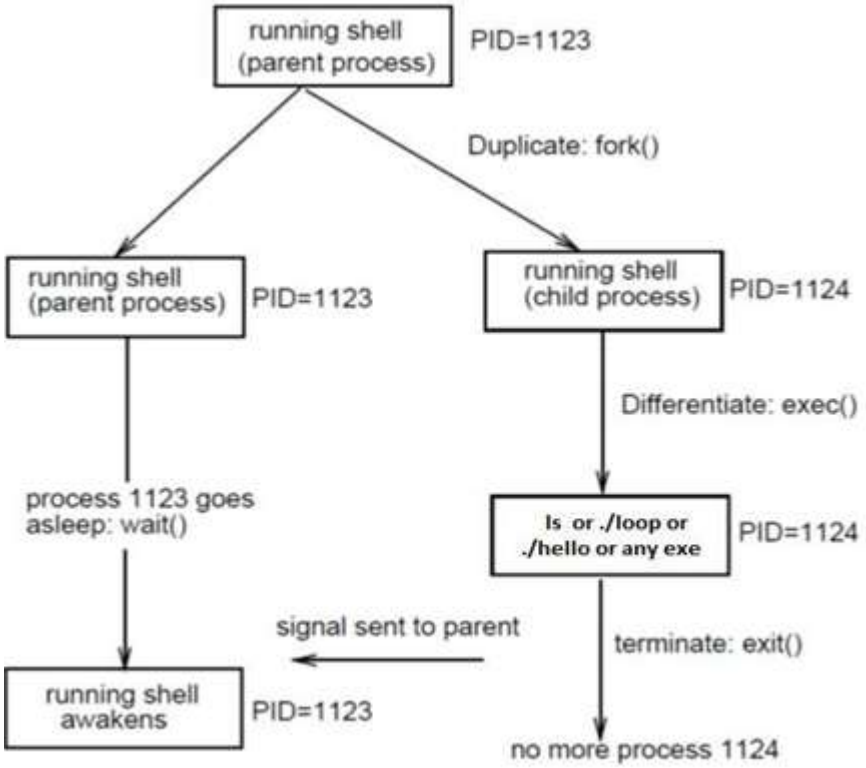
Unix Processes..

- Unix starts as a single process, called **init.** The PID of init is 1.
- The only way to create a new process in Unix, is to **duplicate** an existing one.
- the process **init** is the **ancestor** of all subsequent processes.
- Process **init** never dies

The **creation or spawning** of new processes is done with two **system calls** :

- **fork() :** duplicates the caller process
- **exec() :** replaces the caller process by a new one.

//loop.c
Ex: Running <u>any</u> executable from the shell (./loop, ls etc)



Creating a New Process: fork()

- Synopsis: **pid_t fork(void)**
- when successful, the fork() system call :
  - creates a copy of the caller (parent) process
  - returns the **PID** of the newly created child process to the parent
- returns 0 to the new process (the child).
- If not successful, fork() returns -1.
- fork() is a strange system call : called by a single process but **returns twice**, to two different processes (parent and child)
- **Very important:** After the fork() system call is invoked in the parent and the child process is created, both the parent and the child process run **concurrently** in the system.
- Execution resumes **at the line immediately after the fork() statement** in both the parent and the child process.

Parent Process

```
int main(void)
{
int i=fork(); // returns pid of child (>0) on
success
if(i==0)
{
printf("\n\nCHILD PROCESS\n");
else if (i<0)
{
printf("\n\nERROR\n");
}
else
{
printf("\n\nPARENT PROCESS\n");
}
}
```

Duplicate child process is created. Both parent and child execute **concurrently**

Child Process

```
int main(void)
{
int i=fork(); //returns 0 on success
if(i==0)
{
printf("\n\nCHILD PROCESS\n");
else if (i<0)
{
printf("\n\nERROR\n");
}
else
{
printf("\n\nPARENT PROCESS\n");
}
}
```

f1.c
fork9loop.c

- After forking, the child process will have :
  - its own unique PID
  - a different PPID (PID of its parent process)
  - its own **copy** of the parent's data segment and **file descriptors**

fork() is primarily used in two situations :

- A process wants to execute another program (Ex: Bash is a process that wants to run ./welcome)
- A process has a main task and when necessary, creates a child to handle a subtask (servers/sockets)

```
main() {

int a;
a=fork();
a=fork();
printf("\nProcess id = %d, Parent id= %d", getpid(), getppid());

}
```

```
main() {

int a;
a=fork();
a=fork();
printf("\nProcess id = %d, Parent id= %d", getpid(), getppid());

}
```

```
main() {

int a;
a=fork();
a=fork();
printf("\nProcess id = %d, Parent id= %d", getpid(), getppid());

}
```

Note: ***n*** consecutive fork() calls in a process creates ***(2 ^{n} – 1)*** ***child/descdendent*** processes

After the fork() system call creates a child process, the execution **resumes from the line immediately after fork()** in both the parent and the child process

```
main() {

int a;
a=fork();
a=fork();
printf("\nProcess id = %d, Parent id= %d", getpid(), getppid());

}
```

```
#include <stdio.h>
#include <stdlib.h>
//f44.c

main()
{

int a;
a=fork();
a=fork();
printf("\nProcess id = %d, Parent id= %d", getpid(), getppid());

}
```

```
// fork33.c

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

int main(int argc, char *argv[]){
int pid;
pid = fork();
pid = fork();
pid = fork();

//seven new processes are created, in addition to the parent process.

if(pid==0)
{
for(;;);
}
else if(pid<0)
{
printf("Error Forking");
}
else
{
for(;;);
}

}
```

Terminating a Process: exit()

Synopsis: **void exit(int status);**

This call terminates a process and does not return a value

The **status** value is available to the parent process through the wait() system call.

When invoked by a process, the exit() system call :

- closes all the process's file descriptors
- frees the memory used by its code, data and stack
- sends a **SIGCHLD** signal to its parent (remember, every process has a parent(barring init) and waits for the parent to accept its return code.
  - **SIGCHLD:** A signal that indicates a process started by the current process has terminated
  - Note: SIGCHILD is a signal and **not a return value**

Waiting for a process: wait() // f2.c

- Synopsis: **pid_t wait(int *status);**
- This call allows a parent process to wait for one of its children to terminate and to accept its child's termination code.
- When called, wait() can:
  - Block (suspend) the caller process, until one of its children terminates
  - or return the PID of the child process, If a child has terminated and is waiting for its termination to be accepted, or
  - return immediately with an error(-1) if it does not have any child process.
- When successful, **wait()** returns the PID of the terminating child process.

//forkexit1.c

- Some bit-manipulation macros have been defined to deal with the value in the variable **status**(you need to include < sys/wait.h >)
- **WIFEXITED(status)** : returns true for normal child termination.
- **WEXITSTATUS(status)** : used only when WIFEXITED(status) is true, it returns the exit status as an integer(0-255).
- **WIFSIGNALED(status)** : true for abnormal child termination
- **WTERMSIG(status)** : used only when WIFSIGNALED(status) is true, it returns the signal number that caused the abnormal death of the child process.

Waiting for a specific process: waitpid()

Synopsis:

Pid_t waitpid(pid_t pid, int *status, int options);

This call allows a parent process to wait for a specific child to terminate and to accept its child's termination code.

Note: wait(&status) is equivalent to waitpid(-1, &status, 0)

//pid is always a positive integer

Orphan and Zombie Processes //op.c

- A process that terminates does not actually leave the system before the parent process accepts its return.
- There are 2 interesting situations :
  - Parent exits(**for example, the parent has been killed prematurely**) while its children are still alive- **The children become orphans.**
  - Parent is not in a position to accept the exit and the termination code of the child process (parent is in an infinite loop)- **The children become zombies**
- Because some process must accept their return codes, the kernel simply changes their **PPID to 1** (init process) in the absence of a parent process
- Orphan processes are systematically adopted by the process init (PID of init is 1) and init accepts all its children returns.

Zombie Processes //zomex.c

When parent processes is not able to accept the termination code of their child processes, the children become zombies and remain in the system's process table waiting for the acceptance of their return. However, they loose their resources (data, code, stack...).

Because the system's process table has a fixed-size, too many zombie processes can require the intervention of the system administrator.

```
//zomex.c
int main(int argc, char *argv[]){
int pid;
pid = fork();
if(pid==0){
//Child
printf("child process, pid=%d\n", getpid());
exit(0);
}
else if(pid<0){
printf("Error Forking");
}
else{
//Parent
while(1)
sleep(5);
}}
```

Use the following command/s to periodically kill all the **forked processes** by the user in your system. Otherwise, the system performance will be negatively affected.

$ ps –u //obtain the list of user processes

$kill -9 pid //to kill a specific process

$ killall –u uername //to kill all the processes spawned by the user

PATH VARIABLE

$ echo $PATH

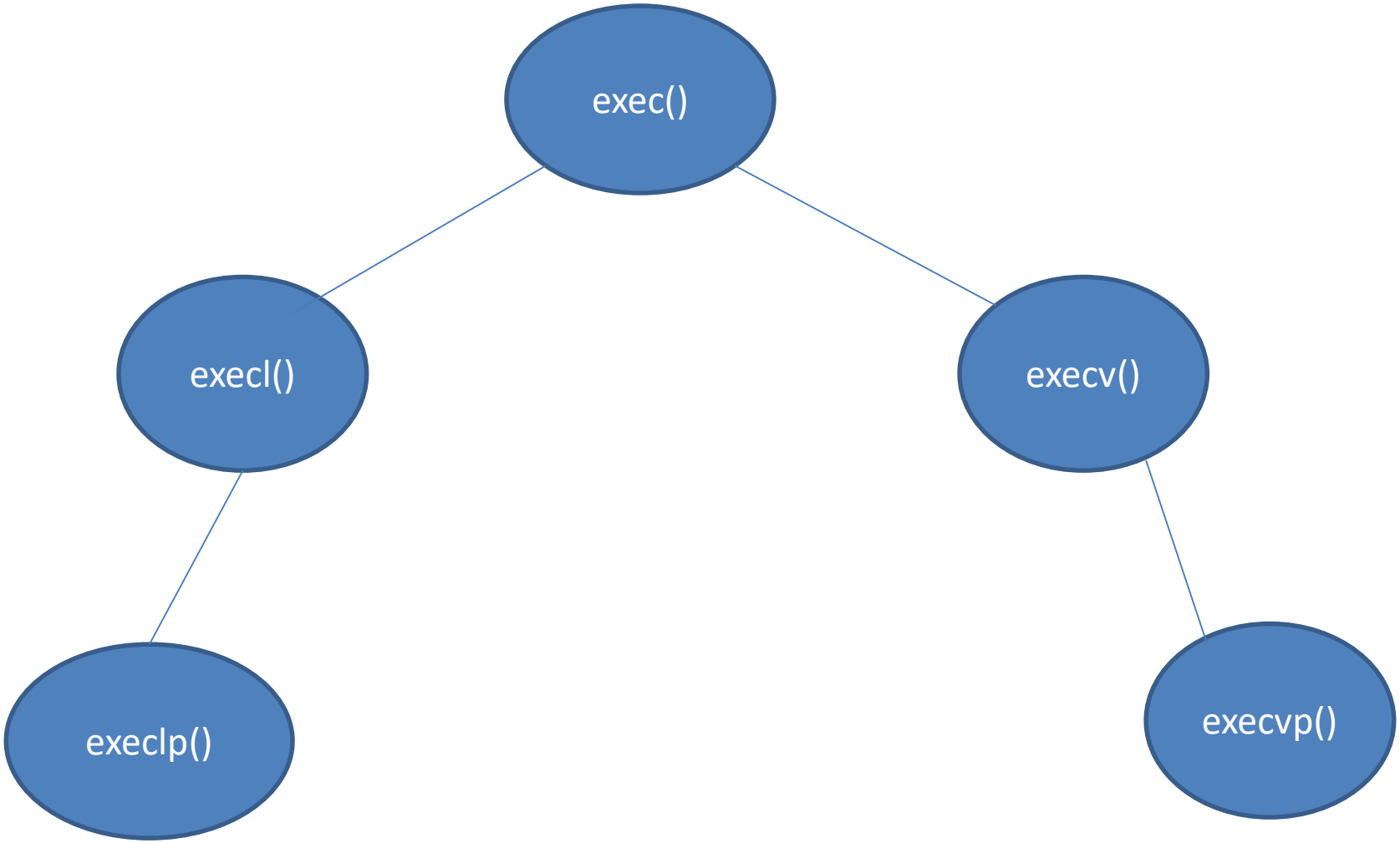
$ export PATH= $PATH:/home/pranga

$ export PATH=$PATH:~ //both are the same

$ export PATH=$PATH:~/chapter5

$ echo $PATH

The exec() family of system calls



exec()

The exec() family of system calls allows a process to **replace its current code, data and stack with those of another program (PID remains the same)**

- **int execl(const char *path, [const char *argi,]+ NULL)** // whole pathname of the executable is required
- Same as: int execl(const char *path, arg(0),arg(1),arg(2),……arg(n),NULL);
  - Example: execl(“/bin/ls”, “/bin/ls”, “-1”,NULL); //executes ls -1
  - arg(0) is once again the whole path of the executable
  - arg(1), arg(2) …..arg(n) are the argument list of the program (if any)

- **int execlp(const char *path, [const char *argi,]+ NULL)** //whole pathname of the executable not required if it is added to the PATH variable
- Same as: int execlp(const char *path, arg(0),arg(1),arg(2),……arg(n),NULL);
  - Example: execlp(“ls”, ”ls”, “-1”, NULL); //whole pathname of the executable is not required

exec() …continued

- **int execv(const char *path, const char *argv[])** // whole pathname of the executable required
- Has only two arguments
- The name of the program, argument list of the program and the terminal NULL character are represented by an array of charcters

- **int execvp(const char *path, const char *argv[])** // whole pathname of the executable not required if it is added to the PATH variable
  - All the arguments along with the NULL terminator are first stored in argv[] , the argument vector

Cont…

- where i = 0; : : : ; n and + means one or more times.
- The difference between these 4 system calls has to do with syntax.
- execl() and execv() require the **whole pathname of the executable** program to be supplied.
- execlp() and execvp() use the variable **$PATH** to find the program.
- exec*() never returns when it is successful. It returns -1 if it is not successful

```
int main(){    //forkexec.c
   int p;
   p = fork();
   if(p==-1)   {
      printf("There is an error while calling fork()");
   }
   if(p==0) {
      printf("\nWe are in the child process\n");
      printf("\nThe child process is now being replaced by the executable ls -1 \n");
      int k=execlp(“ls”, “ls”, “-1”, NULL);
      exit(0); }
   else
   {
     int k=wait();
      printf("\nWe are in the parent process\n");
      exit(0);   }
   return 0;
}
```

execl() //execl.c

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

int main(void) {
  char *programName = "/bin/ls";
  char *arg1 = "-1";
 int k=execl(programName, programName, arg1, NULL);
 //int execl(const char *path, [const char *argi,]+ NULL)
// arg0 must be the name of the program
  return 0;
}
```

execlp() //execlp.c

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

int main(void) {
  char *programName = “ls";
  char *arg1 = "-1";
 int k=execlp(programName, programName, arg1, NULL);
 //int execlp(const char *path, [const char *argi,]+ NULL)
// arg0 must be the name of the program
  return 0;
}
```

execv() //execv.c

```
#include <unistd.h>

int main(void) {
  char *programName = "/bin/ls";
  char *args[] = {“/bin/ls”, "-1", "/home/pranga/chapter3", NULL}; //ls has two
  arguments in this example
  int k=execv(programName, args);
 //int execv(const char *path, const char *argv[])
 //argv[0] must be the name of the program
  return 0;
}
```

execvp() //execvp.c

```
#include <unistd.h>

int main(void) {
  char *programName = "ls";
  char *args[] = {“ls”, "-1", "/home/pranga/chapter5", NULL};
  //Display the contents of  "/home/pranga/chapter5“
  int k=execvp(programName, args);
 //int execvp(const char *path, const char *argv[])
 //argv[0] must be the name of the program
  return 0;
}
```

For both execl() and execlp() arg0 must be the name of the program.

For both execv() and execvp() arg[0] must be the name of the program.

chdir() //chdirfork.c

A child process inherits its current working directory from its parent.

Each process can change its working directory using chdir().

Synopsis int chdir(const char * pathName);

returns 0 if successful -1 otherwise.

It fails if the specified path name does not exist or if the process does not have execute permission from the directory.

Summary

- Unix Processes
- Creating a new process: fork()
- Terminating a process: exit()
- Waiting for a process: wait()
- Waiting for a specific process: waitpid()
- Orphan and Zombie Processes
- exec()
- Changing Directory : chdir()
- Summary

Three State Changes from the normal (execution state)

Terminated
- Normally
- Signalled

Paused
Resumed

-1 meaning wait for any child process.
0 meaning wait for any child process whose process group ID is equal to that of the calling process at the time of the call to **waitpid**(). > 0 meaning wait for the child whose process ID is equal to the value of *pid*.

APPENDIX

REMOVING A PATH FROM THE PATH VARIABLE (ONE OF THE WAYS)

$export PATH=${PATH%:/home/pranga/chapter5} //Removes /home/pranga/chapter5 from the path variable

THANK YOU