

# CS 332/532 Systems Programming

Final  
Review

Professor : Mahmut Unan – UAB CS

# Final Exam (1G)

- 12/13/2024
- Friday
- 02:00pm – 03:00pm
  - Multiple Choice
  - True/False

# Final Exam (1G)

Bring your laptop

We will be using lockdown browser

You must be present in the classroom

Bring your UAB ID (onocard)

# IDEA Survey

- Please participate
- Useful Feedback
- Comments section
  - Provide feedback for TAs

# IDEA Survey

- Please participate
- Useful Feedback
- Evaluate Dr. Unan (primary instructor)
- Comments section
  - Feedback about the TA
  - Feedback about Dr. Zhao

# Hello World !

```
1 #include <stdio.h>
2
3 ► int main() {
4     printf("Hello World!");
5     return 0;
6 }
7
```

# #include

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

- instructs the compiler to include the contents of the stdio.h file into the program before it is compiled
- if the file name is enclosed in brackets <>, the compiler searches in system dependent predefined directories, where system files reside.
- If it is enclosed in double quotes "", the compiler usually begins with the directory of the source file, then searches the predefined directories

# The main () function

```
3 > int main() {  
4     printf("Hello World!");  
5     return 0;  
6 }  
7
```

- Every C program must contain a function named main()
- main function will be automatically called when the program runs
- int indicates that the main function will return an integer
  - How about void?

# Comments

```
#include <stdio.h>
/* This program calls printf() to display a message on the screen. */
int main(void)
{
    printf("Hey Ho, Let's Go\n");
    return 0;
}
```

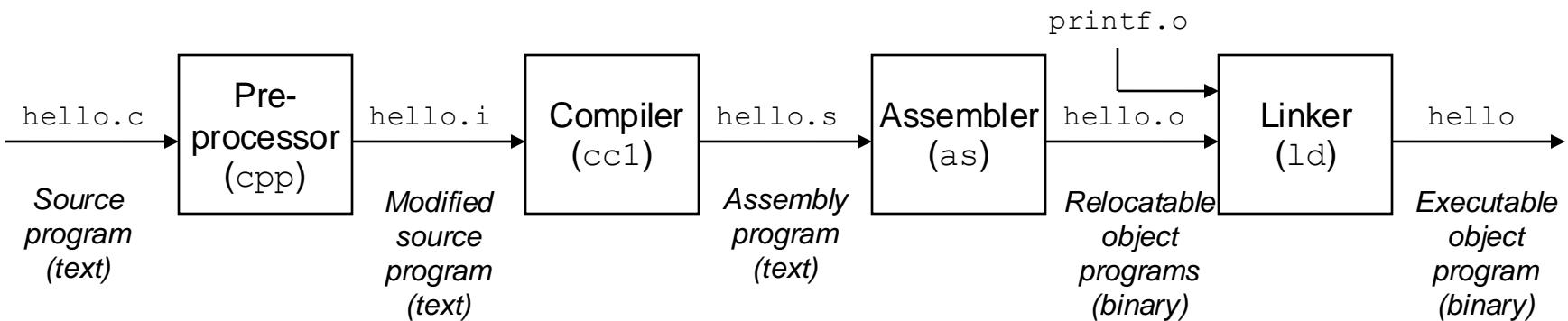
Nested comments are not allowed.

For example, the following code is illegal,  
and the compiler will raise an error message:

```
/*
/* Another comment. */
*/
```

# Compilation

- unix> gcc -o hello hello.c



## Compilation System

# Compiler Options

- You can compile your C program with various levels of optimization turned on (e.g., `-O`, `-O3`, `-Ofast`). Here are some useful/popular compiler and optimization options:
- **The most basic form:** `gcc hello.c` executes the complete compilation process and outputs an executable with name `a.out`
- **Use option `-o`:** `gcc hello.c -o hello` produces an output file with name ‘hello’.
- **Use option `-Wall`:** `gcc -Wall hello.c -o hello` enables all the warnings in GCC.
- **Use option `-E`:** `gcc -E hello.c > hello.i` produces the output of preprocessing stage
- **Use option `-S`:** `gcc -S hello.c > hello.S` produces only the assembly code
- **Use option `-C`:** `gcc -C hello.c` produces only the compiled code (without linking)
- **Use option `-O`:** `gcc -O hello.c` sets the compiler's optimization level.

# Running Hello Object File

- Running hello object file on the shell

```
unix> ./hello
```

```
hello, world
```

```
unix>
```

# Variables

- A *variable* in C is a memory location with a given name. The value of a variable is the content of its memory location. A program may use the name of a variable to access its value.
- Here are some basic rules for naming variables. These rules also apply for function names. Be sure to follow them or your code won't compile:
  - The name can contain letters, digits, and *underscore characters* `_`.
  - The name must begin with either a letter or the underscore character.
  - C is *case sensitive*, meaning that it distinguishes between uppercase and lowercase letters.

# Variables / 2

- The following keywords cannot be used as variable names because they have special significance to the C compiler.

auto	do	goto	signed	unsigned
break	double	if	sizeof	void
case	else	int	static	volatile
char	enum	long	struct	while
const	extern	register	switch	
continue	float	return	typedef	
default	for	short	union	

# Declaring Variables

- **data\_type name\_of\_variable;**

## C Data Types

Data Type	Usual Size (bytes)	Range of Values(min–max)
char	1	-128...127
short int	2	-32.768...32.767
int	4	-2.147.483.648...2.147.483.647
long int	4	-2.147.483.648...2.147.483.647
float	4	Lowest positive value: $1.17 \times 10^{-38}$ Highest positive value: $3.4 \times 10^{38}$
double	8	Lowest positive value: $2.2 \times 10^{-308}$ Highest positive value: $1.8 \times 10^{308}$
long double	8, 10, 12, 16	
unsigned char	1	0...255
unsigned short int	2	0...65535
unsigned int	4	0...4.294.967.295
unsigned long int	4	0...4.294.967.295

# What is the output ?

```
1 #include <stdio.h>
2
3 ► int main() {
4
5     float a = 3.1;
6     if (a == 3.1)
7         printf("Yes\n");
8     else
9         printf("No\n");
10    printf("%.9f\n", a - 3.1);
11    return 0;
12 }
```

# What is the output ?

```
1 #include <stdio.h>
2
3 ► int main() {
4
5     float a = 3.1;
6     if (a == 3.1)
7         printf("Yes\n");
8     else
9         printf("No\n");
10    printf("%.9f\n", a - 3.1);
11    return 0;
12 }
13
```

No  
-0.000000095

```
10 ► int main () {  
11  
12     /* variable definition: */  
13     int a, b;  
14     int c;  
15     float f;  
16     char s[15] = "sample string";  
17     double d;  
18  
19     /* actual initialization */  
20     a = 10;  
21     b = 20;  
22     d=25.0;  
23  
24     c = a + b;  
25     printf("value of c : %d \n", c);  
26  
27     f = 70.0/3.0;  
28     printf("value of f : %f \n", f);  
29  
30     d= 50.0/d;  
31     printf("value of d : %f \n", d);  
32  
33     printf("value of s : %s \n", s);  
34  
35     int functionResult = somefunction();  
36     printf("value of function call : %d\n", functionResult);  
37     return 0;  
38 }
```

```
38 ↵  int somefunction(){  
39      int a=5;  
40      int b=7;  
41      return (a+b);  
42  
43  }
```

```
value of c : 30  
value of f : 23.333334  
value of d : 2.000000  
value of s : sample string  
value of function call : 12
```

# Arithmetic Conversions

```
char c;
short s;
int i;
unsigned int u;
float f;
double d;
long double ld;
i = i+c; /* c is converted to int. */
i = i+s; /* s is converted to int. */
u = u+i; /* i is converted to unsigned int. */
f = f+i; /* i is converted to float. */
f = f+u; /* u is converted to float. */
d = d+f; /* f is converted to double. */
ld = ld+d; /* d is converted to long double. */
```

`const`

- A variable whose value cannot change during the execution of the program is called *constant*.

```
const int a = 10;
```

```
const double PI = 3.14;
```

```
int somefunction(const int *data, size_t size);
```

# printf ()

- The `printf()` function is used to display data to `stdout` (*standard output stream*).
- The `printf()` function accepts a variable list of parameters.
  - The first argument is a format string, that is, a sequence of characters enclosed in double quotes, which determines the output format.
- The format string may contain escape sequences and conversion specifications.

## Escape Sequences

- Escape sequences are used to represent nonprintable characters or characters that have a special meaning to the compiler. An escape sequence consists of a backslash (\) followed by a character.

# Escape Sequences

---

\a	Audible alert.
\b	Backspace.
\f	Form feed.
\n	New line.
\r	Carriage return.
\t	Horizontal tab.
\v	Vertical tab.
\\	Backslash.
'	Single quote.
"	Double quote.
\?	Question mark.

# Conversion Specifications

- A conversion specification begins with the % character, and it is followed by one or more characters with special significance. In its simplest form, the % is followed by one of the conversion specifiers below

Conversion Specifier	Meaning
c	Display the character that corresponds to an unsigned integer value.
d, i	Display a signed integer.
u	Display an unsigned integer.
f	Display a floating-point number. The default precision is six digits.
s	Display a sequence of characters.
e, E	Display a floating-point number in scientific notation. The exponent is preceded by the chosen specifier e or E.
g, G	%e or %E form is selected if the exponent is less than -4 or greater than or equal to the precision. Otherwise, the %f form is used.
p	Display the value of a pointer variable.
x, X	Display an unsigned integer in hex form; %x displays lowercase letters (a-f), while %X displays uppercase letters (A-F).
o	Display an unsigned integer in octal.
n	Nothing is displayed. The matching argument must be a pointer to integer; the number of characters printed so far will be stored in that integer.
%	Display the character %.

```
#include <stdio.h>
int main(void)
{
    int len;
    printf("%c\n", 'w');
    printf("%d\n", -100);
    printf("%f\n", 1.56);
    printf("%s\n", "some text");
    printf("%e\n", 100.25);
    printf("%g\n", 0.0000123);
    printf("%X\n", 15);
    printf("%o\n", 14);
    printf("test%n\n", &len);
    printf("%d%%\n", 100);
    return 0;
}
```

The program outputs:

w (the character constant must be enclosed in single quotes).

-100

1.560000

some text (the string literal must be enclosed in double quotes).

1.002500e+002 (equivalent to  $1.0025 \times 10^2 = 1.0025 \times 100 = 100.25$ ).

1.23e-005 (because the exponent is less than -4, the number is displayed in scientific form).

F (the number 15 is equivalent to F in hex).

16 (the number 14 is equivalent to 16 in octal).

test (since four characters have been printed before %n is met, the value 4 is stored into len).

100% (to display the % character, we must write it twice). 5

# Precision

```
#include <stdio.h>
int main(void)
{
    float a = 1.2365;
    printf("%f\n", a);
    printf("%.2f\n", a);
    printf("%.3f\n", a);
    printf("%.1f\n", a);
    return 0;
}
```

1.236500	
1.24	
1.237	
1	

# scanf()

- The scanf() function is used to read data from stdin (*standard input stream*) and store that data in program variables.
- The scanf() function accepts a variable list of parameters. The first is a format string similar to that of printf(), followed by the memory addresses of the variables in which the input data will be stored.
- Typically, the format string contains only conversion specifiers. The conversion characters used in scanf() are the same as those used in printf().

# scanf ( )

```
int i;  
float j;  
scanf ("%d%f", &i, &j);
```

```
char str[100];  
scanf ("%s", str);
```

```
1 #include <stdio.h>  
2 int main(void)  
3 {  
4     char ch;  
5     int i;  
6     float f;  
7     printf("Enter character, int and float: ");  
8     scanf("%c%d%f", &ch, &i, &f);  
9     printf("\nC:%c\tI:%d\tF:%f\n", ch, i, f);  
10    return 0;  
11 }
```

```
Enter character, int and float: s 17 22.6
```

C:s I:17 F:22.600000

# The assignment operator $=$

```
int a,b,c,d,e;
```

```
a=b=c=d=e=25;
```

or even the following is legal

```
a=25;
```

```
d=a + ( b = ( e = a+10 ) + 40 ) ;
```

# Arithmetic Operators

- + - / \* %
- int/int = cuts off the decimal part

int a=7;

int b=5;

a/b will be equal to 1

also, be careful with the % operator

if ((n%2)==1) is dangerous\*\*

if ((n%2)!=0) is safe

\*\* if n is odd and negative

# **++ and --**

- Similar to Java

```
int a=25;
```

```
a++; /* is equal to a = a+1; */
```

```
a--; /* is equal to a = a-1; */
```

```
int c=5, d;
```

```
d=c++;
```

or

```
d=++c;
```

# Compound Assignment Operators

```
#include <stdio.h>
int main(void)
{
    int a = 4, b = 2;
    a += 6;
    a *= b+3;
    a -= b+8;
    a /= b;
    a %= b+1;
    printf("Num = %d\n", a);
    return 0;
}
```

# Comparisons

- >   >=   <   <=   !=   ==
- if (a == 10)

# Logical Operators

- ! not operator, && operator, || operator

```
#include <stdio.h>
int main(void)
{
    int a = 4;
    printf("%d\n", !a);
    return 0;
}
```

# The Comma Operator

- The comma (,) operator can be used to merge several expressions to form a single expression

```
#include <stdio.h>
int main(void)
{
    int b;
    b = 20, b = b+30, printf("%d\n", b);
    return 0;
}
```

# Operator Precedence

Category	Operator	Associativity
Postfix	( ) [] -> . ++ --	Left to right
Unary	+ - ! ~ ++ -- (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^=  =	Right to left
Comma	,	Left to right

```
1 #include <stdio.h>
2
3 ► int main() {
4
5     int a=10,b=20,c=30,d=40,e;
6
7     e = (a + b) * c / d;          // (10+20)* 30 / 40
8     printf("Value of (a + b) * c / d is : %d\n", e );
9
10    e = ((a + b) * c) / d;       // ((10+20)* 30 ) / 40
11    printf("Value of ((a + b) * c) / d is : %d\n" , e );
12
13    e = (a + b) * (c / d);      // (10+20) * (30/40)
14    printf("Value of (a + b) * (c / d) is : %d\n", e );
15
16    e = a + (b * c) / d;        // 10 + (20*30)/40
17    printf("Value of a + (b * c) / d is : %d\n" , e );
18
19    return 0;
20 }
```

```

1 #include <stdio.h>
2
3 ► int main() {
4
5     int a=10,b=20,c=30,d=40,e;
6
7     e = (a + b) * c / d;          // (10+20)* 30 / 40      22
8     printf("Value of (a + b) * c / d is : %d\n", e );
9
10    e = ((a + b) * c) / d;       // ((10+20)* 30 ) / 40    22
11    printf("Value of ((a + b) * c) / d is : %d\n" , e );
12
13    e = (a + b) * (c / d);      // (10+20) * (30/40)      0
14    printf("Value of (a + b) * (c / d) is : %d\n" , e );
15
16    e = a + (b * c) / d;        // 10 + (20*30)/40      25
17    printf("Value of a + (b * c) / d is : %d\n" , e );
18
19    return 0;
20 }
```

# if else else if

```
#include <stdio.h>
int main(void)
{
    int a = 10, b = 20, c = 30;
    if(a > 5)
    {
        if(b == 20)
            printf("1 ");
        if(c == 40)
            printf("2 ");
        else
            printf("3 ");
    }
    else
        printf("4\n");
    return 0;
}
```

# switch statement

```
#include <stdio.h>
int main(void)
{
    int a;
    printf("Enter number: ");
    scanf("%d", &a);
    switch(a)
    {
        case 1:
            printf("One\n");
            break;
        case 2:
            printf("Two\n");
            break;
        default:
            printf("Other\n");
            break;
    }
    printf("End\n");
    return 0;
}
```

# for loop

```
1      #include <stdio.h>
2 ►  - int main(void)
3   {
4       int a;
5       for(a = 0; a < 5; a++)
6       {
7           printf("%d ", a);
8       }
9       return 0;
10      }
```

# The break Statement

```
#include <stdio.h>
int main(void)
{
    int i;
    for(i = 1; i < 10; i++)
    {
        if(i == 5)
            break;
        printf("%d ", i);
    }
    printf("End = %d\n", i);
    return 0;
}
```

# The continue Statement

```
#include <stdio.h>
int main(void)
{
    int i;
    for(i = 1; i < 10; i++)
    {
        if(i < 5)
            continue;
        printf("%d ", i);
    }
    return 0;
}
```

# while loop

```
#include <stdio.h>
int main(void)
{
    int i = 10;
    while(i != 0)
    {
        printf("%d\n", i);
        i--;
    }
    return 0;
}
```

# do-while loop

```
#include <stdio.h>
int main(void)
{
    int i = 1;
    do
    {
        printf("%d\n", i);
        i++;
    } while(i <= 10);
    return 0;
}
```

# References

- [https://www.tutorialspoint.com/cprogrammin  
g/c constants.htm](https://www.tutorialspoint.com/cprogramming/c_constants.htm)
- C From Theory to Practice - 2nd edition,  
Nikolaos D. Tselikas and George S. Tselikis

# Data Types in C

## Integer Types

The following table provides the details of standard integer types with their storage sizes and value ranges –

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

# Floating-Point Types

The following table provide the details of standard floating-point types with storage sizes and value ranges and their precision –

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

The header file float.h defines macros that allow you to use these values and other details about the binary representation of real numbers in your programs. The following example prints the storage space taken by a float type and its range values –

# for loop

```
1 #include <stdio.h>
2 int main() {
3     for (;;) {
4         printf("This is a strange infinite loop");
5     }
6 }
```

# while loop

```
#include <stdio.h>
int main(void)
{
    int i = 10;
    while(i != 0)
    {
        printf("%d\n", i);
        i--;
    }
    return 0;
}
```

# do-while loop

```
#include <stdio.h>
int main(void)
{
    int i = 1;
    do
    {
        printf("%d\n", i);
        i++;
    } while(i <= 10);
    return 0;
}
```

# Arrays

- **One-Dimensional Arrays**
  - An array is a data structure that contains a number of values, or else elements, of the same type.
  - Each element can be accessed by its position within the array
  - Always declare the array before you try to use them

```
data_type array_name[number_of_elements];  
int sampleArray[100];  
float anotherArray[250];
```

# predefined size

```
/* use macros */  
#define ARRAY_SIZE 250  
float sampleArray[ARRAY_SIZE];
```



```
/* never use const */  
const int array_size = 250;  
float sampleArray(array_size)  
/* this is not legal */
```



# sizeof()

```
1 #include <stdio.h>
2 ► int main() {
3
4     printf("%lu\n", sizeof(char));
5     printf("%lu\n", sizeof(int));
6     printf("%lu\n", sizeof(float));
7     printf("%lu\n", sizeof(double));
8
9     int a = 25;
10    double d= 40.55;
11    printf("%lu\n", sizeof(a+d));
12
13    int arr[10] = {5,8,9,12};
14    printf("\n Size of the array :%lu", sizeof(arr));
15    printf("\n Capacity the array :%lu", sizeof(arr)/sizeof(arr[0]));
16
17    int arr2[] = {5,8,9,12};
18    printf("\n Size of the array2 :%lu", sizeof(arr2));
19    printf("\n Capacity the array2 :%lu", sizeof(arr2)/sizeof(arr2[0]));
20    return 0;
21 }
```

# sizeof()

```
1 #include <stdio.h>
2 ► int main() {
3
4     printf("%lu\n", sizeof(char));    1
5     printf("%lu\n", sizeof(int));    4
6     printf("%lu\n", sizeof(float));   4
7     printf("%lu\n", sizeof(double));  8
8
9     int a = 25;
10    double d= 40.55;
11    printf("%lu\n", sizeof(a+d));    8
12
13    int arr[10] = {5,8,9,12};           40
14    printf("\n Size of the array :%lu", sizeof(arr));
15    printf("\n Capacity the array :%lu", sizeof(arr)/sizeof(arr[0])); 10
16
17    int arr2[] = {5,8,9,12};           16
18    printf("\n Size of the array2 :%lu", sizeof(arr2));
19    printf("\n Capacity the array2 :%lu", sizeof(arr2)/sizeof(arr2[0])); 4
20    return 0;
21 }
```

# Initialize the Array

```
int arr[3]={10,20,30};
```

```
int arr2[10]={10,20};
```

```
int arr3[]={10,20,30,40};
```

```
/* be careful with the  
following*/
```

```
const int arr4[] = {10,20,30,40}
```

# Assign - Access elements

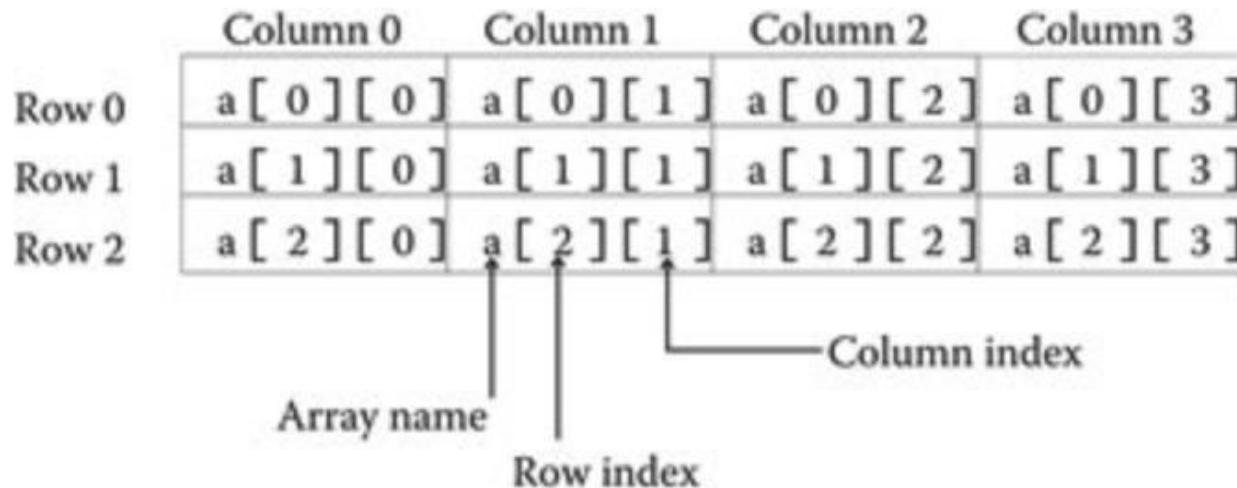
```
1 #include <stdio.h>
2 ► int main() {
3     int i, j=10, arr[10];
4     arr[0]=10;
5     arr[1]=arr[0]*2;
6     for (i=2;i<10;i++){
7         arr[i]=j*(i+1);
8     }
9     for (i=0;i<10;i++)
10        printf("\n arr[%d] :%d",i,arr[i]);
11
12    return 0;
13 }
```

arr[0]	:	10
arr[1]	:	20
arr[2]	:	30
arr[3]	:	40
arr[4]	:	50
arr[5]	:	60
arr[6]	:	70
arr[7]	:	80
arr[8]	:	90
arr[9]	:	100

# 2D Arrays

- **data\_type array\_name[number\_of\_rows][number\_of\_columns]**

```
int a[3][4];
```



# initialize 2D array

```
int arr[3][3] = {{10, 20, 30},  
{40, 50, 60}, {70, 80, 90}};
```

```
int arr[3][4] = {10, 20, 30, 40,  
50, 60, 70, 80, 90};
```

10 20 30 40

50 60 70 80

90 0 0 0

# Pointers

```
int a = 5;  
float arr[25];
```

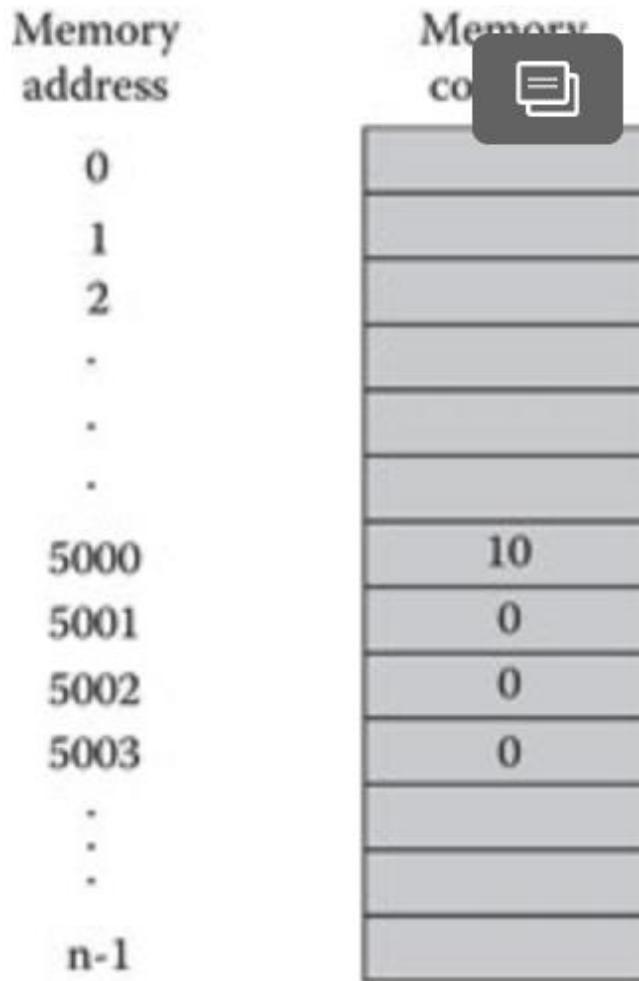
- /\* To reach the memory location variables, arrays...etc use ampersand (&) operator \*/

```
printf("\n %x", &a);  
printf("\n %x", &arr);
```

e7ad78b8

e7ad78c0

# Memory Address



# Pointers / 2

- How to store this address?
  - we use the pointers
  - Pointers is a variable whose value is the address of another variable
- Declare the pointer before you use it

```
data_type *pointer_name;
```

```
int *ptr, a, b, c;
```

```
int * ptr, a, b, c;
```

```
int* ptr, a, b, c;
```

## !!! Caution !!!

All three statements are correct and the result of each statement the “ptr” will be declared as the pointer but a,b, and c will be declared as int.

However; it is always better to use the first syntax

```
data_type *pointer_name;  
int *ptr, a, b, c;  
int * ptr, a, b, c;  
int* ptr, a, b, c;
```

# size of a pointer ???

```
1  #include <stdio.h>
2 ►  int main() {
3      char c;
4      char *ptrC = &c;
5      int a=5;
6      int *ptrI = &a;
7      float f =20.66;
8      float *ptrF = &f;
9      double d = 44.445;
10     double *ptrD = &d;
11
12     printf("size of c: %u\n", sizeof(c));
13     printf("size of ptrC: %u\n", sizeof(ptrC));
14     printf("size of a: %u\n", sizeof(a));
15     printf("size of ptrI: %u\n", sizeof(ptrI));
16     printf("size of f: %u\n", sizeof(f));
17     printf("size of ptrF: %u\n", sizeof(ptrF));
18     printf("size of d: %u\n", sizeof(d));
19     printf("size of ptrD: %u\n", sizeof(ptrD));
20
21 }
```

```
size of c: 1
size of ptrC: 8
size of a: 4
size of ptrI: 8
size of f: 4
size of ptrF: 8
size of d: 8
size of ptrD: 8
```

```
1 #include <stdio.h>
2 ▶ int main(void)
3 {
4     int *ptr, a;
5     a = 10;
6     ptr = &a;
7     printf("Val = %d\n", *ptr);
8     return 0;
9 }
```

10

Always initialize the pointer before using it, otherwise you will get segmentation fault error

# Example - page 1/2

```
1 #include <stdio.h>
2 ► int main()
3 {
4     int *ptr, a;
5     a = 25;
6     /* without using a pointer */
7     printf("Address of a: %p\n", &a);
8     printf("Value of a: %d\n", a);
9
10    /*let's use a pointer */
11    ptr = &a;
12    printf("Address of the pointer : %p\n", ptr);
13    printf("Value of the pointer : %d\n", *ptr);
14
15    /* how about if we change the value of int */
16    a = 125;
17    printf("Address of the pointer : %p\n", ptr);
18    printf("Value of the pointer : %d\n", *ptr);
19
```

# Example - page 2/2

```
20     /* let's change the value using pointer*/
21     *ptr = 250;
22     printf("Address of a: %p\n", &a);
23     printf("Value of a: %d\n", a);
24
25     /* we can reuse the pointer */
26     int b = 50;
27     ptr = &b;
28     printf("Address of the pointer : %p\n", ptr);
29     printf("Value of the pointer : %d\n", *ptr);
30
31 }
```

# Example - output

```
Address of a: 0x7ffeee56291c
Value of a: 25
Address of the pointer : 0x7ffeee56291c
Value of the pointer : 25
Address of the pointer : 0x7ffeee56291c
Value of the pointer : 125
Address of a: 0x7ffeee56291c
Value of a: 250
Address of the pointer : 0x7ffeee562918
Value of the pointer : 50
```

# Example - page 1/2

```
1 #include <stdio.h>
2 ► int main()
3 {
4     int *ptr, a;
5     a = 25;
6     /* without using a pointer */
7     printf("Address of a: %p\n", &a);
8     printf("Value of a: %d\n", a);
9
10    /*let's use a pointer */
11    ptr = &a;
12    printf("Address of the pointer : %p\n", ptr);
13    printf("Value of the pointer : %d\n", *ptr);
14
15    /* how about if we change the value of int */
16    a = 125;
17    printf("Address of the pointer : %p\n", ptr);
18    printf("Value of the pointer : %d\n", *ptr);
19
```

# Example - page 2/2

```
20     /* let's change the value using pointer*/
21     *ptr = 250;
22     printf("Address of a: %p\n", &a);
23     printf("Value of a: %d\n", a);
24
25     /* we can reuse the pointer */
26     int b = 50;
27     ptr = &b;
28     printf("Address of the pointer : %p\n", ptr);
29     printf("Value of the pointer : %d\n", *ptr);
30
31 }
```

# Example - output

```
Address of a: 0x7ffeee56291c
Value of a: 25
Address of the pointer : 0x7ffeee56291c
Value of the pointer : 25
Address of the pointer : 0x7ffeee56291c
Value of the pointer : 125
Address of a: 0x7ffeee56291c
Value of a: 250
Address of the pointer : 0x7ffeee562918
Value of the pointer : 50
```

# The \* and & cancel each other when used together

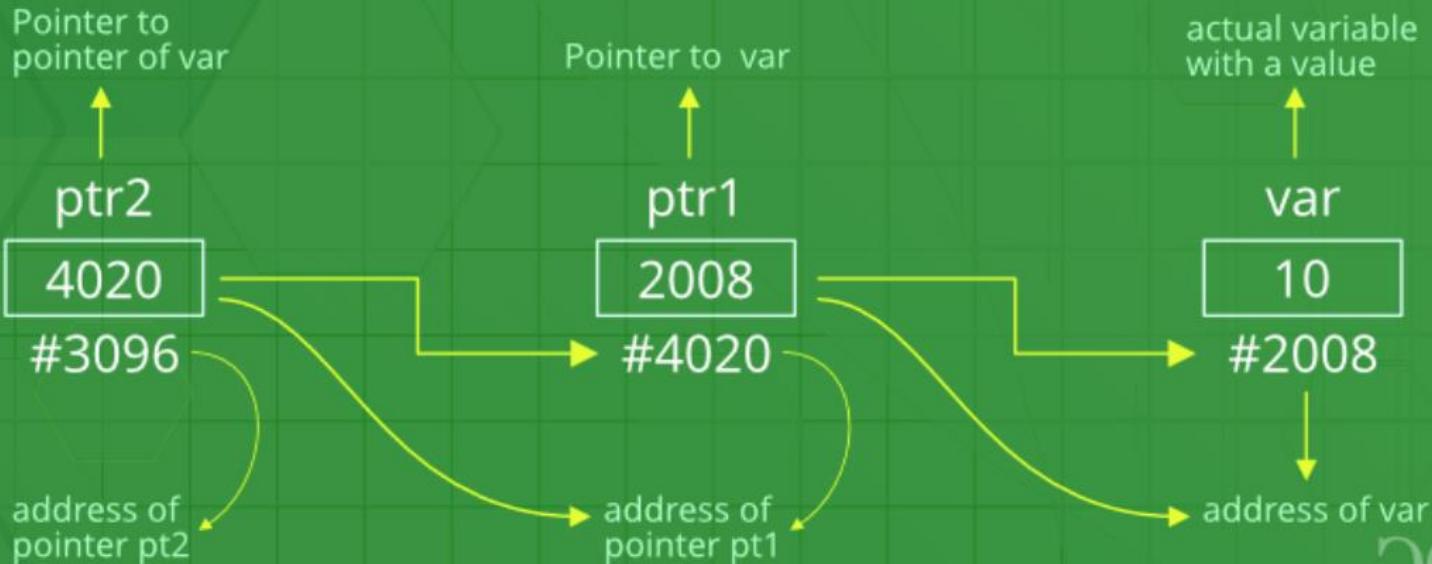
```
#include <stdio.h>

int main() {
    int *ptr, i=5;
    ptr = &i;
    printf("%p %p %p %d %p\n", &i, *ptr, &ptr, *ptr, &ptr);
    return 0;
}
```

```
0x7ffee636291c 0x7ffee636291c 0x7ffee636291c 5 0x7ffee6362920
```

# double pointer? even triple...

## Double Pointer



```
#include <stdio.h>
int main() {
    int a = 25;
    int *ptr = &a;//the first pointer

    printf("Address : %p\n", ptr);
    printf("Value : %i\n", *ptr);

    *ptr = 45;// change the value
    printf("Address : %p\n", ptr);
    printf("Value : %i\n", *ptr);

    int **ptr2 = &ptr;// second pointer
    printf("Address - First pointer: %p\n", ptr);
    printf("Value -First Pointer: %i\n", *ptr);
    printf("Address - First pointer: %p\n", ptr2);
    printf("Value -First Pointer: %i\n", **ptr2);
    return 0;
}
```

```
#include <stdio.h>
int main() {
    int a = 25;
    int *ptr = &a;//the first pointer
    printf("Address : %p\n", ptr);
    printf("Value : %i\n", *ptr);

    *ptr = 45;// change the value
    printf("Address : %p\n", ptr);
    printf("Value : %i\n", *ptr);

    int **ptr2 = &ptr;// second pointer
    printf("Address - First pointer: %p\n", ptr);
    printf("Value -First Pointer: %i\n", *ptr);
    printf("Address - First pointer: %p\n", ptr2);
    printf("Value -First Pointer: %i\n", **ptr2);
    return 0;
}
```

```
Address : 0x7ffee76aa928
Value : 25
Address : 0x7ffee76aa928
Value : 45
Address - First pointer: 0x7ffee76aa928
Value -First Pointer: 45
Address - First pointer: 0x7ffee76aa920
Value -First Pointer: 45
```

# Arrays & Pointers

```
1 #include <stdio.h>
2 ► └ int main() {
3     int i, arr[5];
4     double arr2[5];
5
6     for(i = 0; i < 5; i++)
7         printf("address of arr[%d] = %p\n", i, &arr[i]);
8
9     for(i = 0; i < 5; i++)
10        printf("address of arr2[%d] = %p\n", i, &arr2[i]);
11
12 }
```

# Arrays & Pointers

```
1      address of arr[0] = 0x7ffeeece0910
2      address of arr[1] = 0x7ffeeece0914
3      address of arr[2] = 0x7ffeeece0918
4      address of arr[3] = 0x7ffeeece091c
5      address of arr[4] = 0x7ffeeece0920
6
7      address of arr2[0] = 0x7ffeeece08e0
8      address of arr2[1] = 0x7ffeeece08e8
9      address of arr2[2] = 0x7ffeeece08f0
10     address of arr2[3] = 0x7ffeeece08f8
11     address of arr2[4] = 0x7ffeeece0900
12 
```

# Array Manipulation

```
1 #include <stdio.h>
2 ► int main() {
3     int arr[5] = {10, 20, 30, 40, 50};
4     int *ptr;
5
6     ptr = &arr[3]; // address of the fourth element
7
8     printf("\n Pointer value : %d", *ptr);
9     printf("\n Next Value : %d", *(ptr+1));
10    printf("\n Previous Value : %d", *(ptr-1));
11
12    printf("\n Address of the Pointer : %p", &(*(ptr)));
13    printf("\n Address of the Next Value : %p", &(*(ptr+1)));
14    printf("\n Address of the Previous Value : %p", &(*(ptr-1)));
15    return 0;
16 }
```

# Array Manipulation

```
1 Pointer value : 40
2 Next Value : 50
3 Previous Value : 30
4 Address of the Pointer : 0x7ffeeb08e91c
5 Address of the Next Value : 0x7ffeeb08e920
6 Address of the Previous Value : 0x7ffeeb08e918
7
```

```
8     printf("\n Pointer value : %d", *ptr);
9     printf("\n Next Value : %d", *(ptr+1));
10    printf("\n Previous Value : %d", *(ptr-1));
11
12    printf("\n Address of the Pointer : %p", &(*(ptr)));
13    printf("\n Address of the Next Value : %p", &(*(ptr+1)));
14    printf("\n Address of the Previous Value : %p", &(*(ptr-1)));
15    return 0;
16 }
```

# What is the Result ?

```
1      #include <stdio.h>
2  ►  int main(void)
3  {
4      int *ptr, totalSum, arr[5] = {10, 20, 30, 40, 50};
5      totalSum = 0;
6      for(ptr = arr; ptr < arr+5; ptr++)
7      {
8          --*ptr;
9          totalSum += *ptr;
10     }
11     printf("Sum = %d\n", totalSum);
12     return 0;
13 }
```

# What is the Result ?

```
1 #include <stdio.h>
2 ► int main(void)
3 {
4     int *ptr, totalSum, arr[5] = {10, 20, 30, 40, 50};
5     totalSum = 0;
6     for(ptr = arr; ptr < arr+5; ptr++)
7     {
8         --*ptr;
9         totalSum += *ptr;
10    }
11    printf("Sum = %d\n", totalSum);
12    return 0;
13 }
```

145

# Passing Pointers to functions

```
1 #include <stdio.h>
2 ← void test(int a);
3 ► ⌂ int main(void)
4 {
5     void (*ptr)(int a);
6     ptr = test;
7     (*ptr)(10);
8     return 0;
9 }
10 ← ⌂ void test(int a)
11 {
12     printf("%d\n", 2*a);
13 }
```

20

```
1 #include <stdio.h>
2 ↵ int addTwoNumbers(int a, int b);
3 ↵ int subtractTwoNumbers(int a, int b);
4
5 ► ↴ int main(void)
6 {
7     int (*ptr[2])(int a, int b);
8     int i, j, result;
9     ptr[0] = addTwoNumbers;
10    ptr[1] = subtractTwoNumbers;
11
12    printf("Enter two integer numbers: ");
13    scanf("%d %d", &i, &j);
14    if(i > 0 && i < 25)
15        result = ptr[0](i, j);
16    else
17        result = ptr[1](i, j);
18    printf("Result : %d\n", result);
19    return 0;
20 }
21 ↵ int addTwoNumbers(int a, int b)
22 { return a+b; }
23 ↵ int subtractTwoNumbers(int a, int b)
24 { return a-b; }
```

```
1 #include <stdio.h>
2 ↵ int addTwoNumbers(int a, int b);
3 ↵ int subtractTwoNumbers(int a, int b);
4
5 ► ↵ int main(void)
6 {
7     int (*ptr[2])(int a, int b);
8     int i, j, result;
9     ptr[0] = addTwoNumbers;
10    ptr[1] = subtractTwoNumbers;
11
12    printf("Enter two integer numbers: ");
13    scanf("%d %d", &i, &j);
14    if(i > 0 && i < 25)
15        result = ptr[0](i, j);
16    else
17        result = ptr[1](i, j);
18    printf("Result : %d", result);
19    return 0;
20 }
21 ↵ int addTwoNumbers(int a, int b)
22 { return a+b; }
23 ↵ int subtractTwoNumbers(int a, int b)
24 { return a-b; }
```

Enter two integer numbers: 20 30

Result : 50

Enter two integer numbers: 45 20

Result : 25

# The char Type

- Since a character in the ASCII set is represented by an integer between 0 and 255, we can use the **char** type to store its value.
- Once a character is stored into a variable, it is the character's ASCII value that is actually stored.

```
char ch;
```

```
ch = 'c';
```

- the value of `ch` becomes equal to the ASCII value of the character 'c'.
- Therefore,
  - the statements `ch = 'c';` and `ch = 99;` are equivalent.
  - Of course, 'c' is preferable than 99; not only it is easier to read, but also your program won't depend on the character set as well.

```
c 1 #include <stdio.h>
2 ► int main(void)
3 {
4     char ch;
5     ch = 'a';
6     printf("Char = %c and its ASCII code is %d\n", ch, ch);
7     return 0;
8 }
9
```

Char = a and its ASCII code is 97

- Since C treats the characters as integers, we can use them in numerical expressions. For example:

```
char ch = 'c';  
  
int i; ch++; /* ch becomes 'd'. */  
ch = 68; /* ch becomes 'D'. */  
i = ch-3; /* i becomes 'A', that is 65 */
```

# getchar () and putchar ()

- The getchar () function is used to read a character from stdin.
- The putchar () function writes a character in stdout, for example, putchar ('a')

# Strings

- A string literal is a sequence of characters enclosed in double quotes.
- C treats it as a nameless character array.
- To store a string in a variable, we use an array of characters.
- Because of the C convention that a string ends with the null character, to store a string of **N** characters, the size of the array should be **N+1** at least.

```
char str[8];
```

An array can be initialized with a string, when it is declared. For example, with the declaration:

```
char str[8] = "message";
```

the compiler copies the characters of the "message" into the str array and adds the null character. In particular, str[0] becomes 'm', str[1] becomes 'e', and the value of the last element str[7] becomes '\0'. In fact, this declaration is equivalent to:

```
char str[8] = { 'm', 'e', 's',
's', 'a', 'g', 'e', '\0' };
```

# puts()

```
1 #include <stdio.h>
2 ► int main(void)
3 {
4     char str[] = "UAB CS 330 Course";
5     puts(str);
6     puts(str);
7     str[4] = '\0';
8     printf("%s\n", str);
9     return 0;
10 }
```

```
UAB CS 330 Course
UAB CS 330 Course
UAB
```

# Passing Pointers to functions

```
1 #include <stdio.h>
2 ← void test(int a);
3 ► ⌂ int main(void)
4 {
5     void (*ptr)(int a);
6     ptr = test;
7     (*ptr)(10);
8     return 0;
9 }
10 ← ⌂ void test(int a)
11 {
12     printf("%d\n", 2*a);
13 }
```

20

```
1 #include <stdio.h>
2 ↵ int addTwoNumbers(int a, int b);
3 ↵ int subtractTwoNumbers(int a, int b);
4
5 ► ↴ int main(void)
6 {
7     int (*ptr[2])(int a, int b);
8     int i, j, result;
9     ptr[0] = addTwoNumbers;
10    ptr[1] = subtractTwoNumbers;
11
12    printf("Enter two integer numbers: ");
13    scanf("%d %d", &i, &j);
14    if(i > 0 && i < 25)
15        result = ptr[0](i, j);
16    else
17        result = ptr[1](i, j);
18    printf("Result : %d\n", result);
19    return 0;
20 }
21 ↵ int addTwoNumbers(int a, int b)
22 { return a+b; }
23 ↵ int subtractTwoNumbers(int a, int b)
24 { return a-b; }
```

```
1 #include <stdio.h>
2 ↵ int addTwoNumbers(int a, int b);
3 ↵ int subtractTwoNumbers(int a, int b);
4
5 ► ↵ int main(void)
6 {
7     int (*ptr[2])(int a, int b);
8     int i, j, result;
9     ptr[0] = addTwoNumbers;
10    ptr[1] = subtractTwoNumbers;
11
12    printf("Enter two integer numbers: ");
13    scanf("%d %d", &i, &j);
14    if(i > 0 && i < 25)
15        result = ptr[0](i, j);
16    else
17        result = ptr[1](i, j);
18    printf("Result : %d", result);
19    return 0;
20 }
21 ↵ int addTwoNumbers(int a, int b)
22 { return a+b; }
23 ↵ int subtractTwoNumbers(int a, int b)
24 { return a-b; }
```

Enter two integer numbers: 20 30

Result : 50

Enter two integer numbers: 45 20

Result : 25

# The char Type

- Since a character in the ASCII set is represented by an integer between 0 and 255, we can use the **char** type to store its value.
- Once a character is stored into a variable, it is the character's ASCII value that is actually stored.

```
char ch;
```

```
ch = 'c';
```

- the value of `ch` becomes equal to the ASCII value of the character 'c'.
- Therefore,
  - the statements `ch = 'c';` and `ch = 99;` are equivalent.
  - Of course, 'c' is preferable than 99; not only it is easier to read, but also your program won't depend on the character set as well.

```
c 1 #include <stdio.h>
2 ► int main(void)
3 {
4     char ch;
5     ch = 'a';
6     printf("Char = %c and its ASCII code is %d\n", ch, ch);
7     return 0;
8 }
9
```

Char = a and its ASCII code is 97

- Since C treats the characters as integers, we can use them in numerical expressions. For example:

```
char ch = 'c';  
  
int i; ch++; /* ch becomes 'd'. */  
ch = 68; /* ch becomes 'D'. */  
i = ch-3; /* i becomes 'A', that is 65 */
```

# getchar () and putchar ()

- The getchar () function is used to read a character from stdin.
- The putchar () function writes a character in stdout, for example, putchar ('a')

# Strings

- A string literal is a sequence of characters enclosed in double quotes.
- C treats it as a nameless character array.
- To store a string in a variable, we use an array of characters.
- Because of the C convention that a string ends with the null character, to store a string of **N** characters, the size of the array should be **N+1** at least.

```
char str[8];
```

An array can be initialized with a string, when it is declared. For example, with the declaration:

```
char str[8] = "message";
```

the compiler copies the characters of the "message" into the str array and adds the null character. In particular, str[0] becomes 'm', str[1] becomes 'e', and the value of the last element str[7] becomes '\0'. In fact, this declaration is equivalent to:

```
char str[8] = { 'm', 'e', 's',  
's', 'a', 'g', 'e', '\0' };
```

# puts()

```
1 #include <stdio.h>
2 ► int main(void)
3 {
4     char str[] = "UAB CS 330 Course";
5     puts(str);
6     puts(str);
7     str[4] = '\0';
8     printf("%s\n", str);
9     return 0;
10 }
```

```
UAB CS 330 Course
UAB CS 330 Course
UAB
```

# scanf()

- `scanf()` takes as an argument a pointer to the array that will hold the input string.
- Since we're using the name of the array as a pointer, we don't add the address operator `&` before its name.
- Because `scanf()` stops reading once it encounters the space character, only the word this is stored into `str`. Therefore, the program outputs this.
- To force `scanf()` to read multiple words, we can use a more complex form such as `scanf ("%[^\\n]", str);`

gets ()                    fgets ()

char \*gets (char \*str) ;

gets ()    **is not safe, don't use it**

```
1 #include <stdio.h>
2 ► int main(void)
3 {
4     char str[100];
5     int num;
6     printf("Enter number: ");
7     scanf("%d", &num);
8     printf("Enter text: ");
9     fgets(str, sizeof(str), stdin);
10    printf("%d %s\n", num, str);
11    return 0;
12 }
```

# The `strlen()` Function

```
size_t strlen(const char *str);
```

The `size_t` type is defined in the C library as an unsigned integer type (usually as `unsigned int` ).

`strlen()` returns the number of characters in the string pointed to by `str`, not counting the null character.

```
1 #include <stdio.h>
2 #include <string.h>
3 ► int main(void)
4 {
5     char str1[100], str2[100];
6     printf("Enter text: ");
7     fgets(str2, sizeof(str2), stdin);
8     strcpy(str1, str2);
9     printf("Copied text: %s\n", str1);
10    return 0;
11 }
```

Enter text: Hello CS330

Copied text: Hello CS330

# Search the following functions

strcat()

strcmp()

# Functions → Array as Arguments

- When a parameter of a function is a one-dimensional array, we write the name of the array followed by a pair of brackets.
- The length of the array can be omitted; in fact, this is the common practice.
- For example:

```
void test(int arr[]);
```

- When passing an array to a function, we write only its name, without brackets. For example:

```
test(arr);
```

# Memory Blocks

- Code
- Data
- Stack
- Heap

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 ↵ void test(void);
4     int global;
5 ► int main(void)
6 {
7     int *ptr;
8     int i;
9     static int st;
10
11    /* Allocate memory from the heap. */
12    ptr = (int*) malloc(sizeof(int));
13
14    if(ptr != NULL)
15    {
16        printf("Code seg: %p\n", test);
17        printf("Data seg: %p %p\n", &global, &st);
18        printf("Stack seg: %p\n", &i);
19        printf("Heap: %p\n", ptr);
20        free(ptr);
21    }
22    return 0;
23 }
24 ↵ void test(void)
25 { /* Do something. */
26 }
```

```
Code seg: 0x106e1ff30
Data seg: 0x106e21024 0x106e21020
Stack seg: 0x7ffee8de091c
Heap: 0x7f8a55405840
```

# Static Memory Allocation

- In static allocation, the memory is allocated from the stack.
- The size of the allocated memory is fixed; we must specify its size when writing the program and it cannot change during program execution.
- For example, with the statement:

```
float grades [1000] ;
```

# Dynamic Memory Allocation

- In dynamic allocation, the memory is allocated from the heap during program execution. Unlike static allocation, its size can be dynamically specified.
- Furthermore, this size may dynamically shrink or grow according to the program's needs.
- Typically, the default stack size is not very large, the size of the heap is usually much larger than the stack size.

# malloc()

```
void *malloc(size_t size);
```

The `size_t` type is usually a synonym of the **unsigned int** type.

The `size` parameter declares the number of bytes to be allocated.

If the memory is allocated successfully;

`malloc()` returns a pointer to that memory,  
NULL otherwise.

# Check the following functions

realloc()

calloc()

free()

memcpy()

memmove()

memcmp()

# Static Memory Allocation

- In static allocation, the memory is allocated from the stack.
- The size of the allocated memory is fixed; we must specify its size when writing the program and it cannot change during program execution.
- For example, with the statement:

```
float grades [1000] ;
```

# Dynamic Memory Allocation

- In dynamic allocation, the memory is allocated from the heap during program execution. Unlike static allocation, its size can be dynamically specified.
- Furthermore, this size may dynamically shrink or grow according to the program's needs.
- Typically, the default stack size is not very large, the size of the heap is usually much larger than the stack size.

# malloc()

```
void *malloc(size_t size);
```

The `size_t` type is usually a synonym of the **unsigned int** type.

The `size` parameter declares the number of bytes to be allocated.

If the memory is allocated successfully;

`malloc()` returns a pointer to that memory,  
NULL otherwise.

# Check the following functions

realloc()

calloc()

free()

memcpy()

memmove()

memcmp()

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 ► int main()
5 {    int *ptr,n,i;
6     /* the number of array elements */
7     printf("How many elements?:\n");
8     scanf("%d",&n);
9
10    ptr = (int*)malloc(n * sizeof(int));
11
12    if (ptr == NULL) {
13        printf("Memory allocation was NOT successful.\n");
14        exit(0);
15    }
16    else {
17        printf("Memory allocation was successful.\n");
18        for (i = 0; i < n; i++)
19            ptr[i] = (i+1) * 10;
20
21        for (i = 0; i < n; ++i)
22            printf("%d, ", ptr[i]);
23    }
24    free(ptr);
25    printf("\nMemory deallocation was successful.\n");
26    return 0;
27 }
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 ► int main()
5 {    int *ptr,n,i;
6     /* the number of array elements */
7     printf("How many elements?:\n");
8     scanf("%d",&n);
9
10    ptr = (int*)malloc(n * sizeof(int));
11
12    if (ptr == NULL) {
13        printf("Memory allocation was NOT successful.\n");
14
15    }
16
17    }
18
19
20
21
22
23
24
25
26
27 }
```

How many elements?:

8

Memory allocation was successful.

10, 20, 30, 40, 50, 60, 70, 80,

Memory deallocation was successful.

```
return 0;
```

# Structures & Unions

```
struct structure_tag {  
    member_list;  
} structure_variable_list;
```

A **struct** declaration defines a type. Although the structure\_tag is optional, we prefer to name the structures we declare and use that name later to declare variables.

```
struct company
{
    char name[50];
    int start_year;
    int field;
    int tax_num;

    int num_empl;
    char addr[50];
    float balance;
};
```

# sizeof()

```
#include <stdio.h>
struct date
{
    int day;
    int month;
    int year;
};
int main(void)
{
    struct date d;
    printf("%u\n", sizeof(d));
    return 0;
}
```

# sizeof()

```
struct test1
{
    char c;
    double d;
    short s;
};

struct test2
{
    double d;
    short s;
    char c;
};
```

```
1 #include <stdio.h>
2 struct student
3 {
4     int code;
5     float grd;
6 };
7 ► int main(void)
8 {
9     struct student s1, s2;
10    s1.code = 1234;
11    s1.grd = 6.7;
12    s2 = s1; /* Copy structure. */
13    printf("C:%d G:%.2f\n", s2.code, s2.grd);
14    return 0;
15 }
```

# Unions

- Like a structure, a union contains one or more members, which may be of different types. The properties of unions are almost identical to the properties of structures; the same operations are allowed as on structures.
- Their difference is that the members of a structure are stored at *different* addresses, while the members of a union are stored at the *same* address.

```
#include <stdio.h>
union sample
{
    char ch;
    int i;
    double d;
};
int main(void)
{
    union sample s;
    printf("Size: %u\n", sizeof(s));
    return 0;
}
```

# Operating Systems

- What is an operating system?
  - What stands between the user and the bare machine
  - The most basic and the important software to operate the computer
  - Similar role to that conductor of an orchestra
- It manages the computer's memory and processes, as well as all of its software and hardware.
- It also allows you to communicate with the computer without knowing how to speak the computer's language (hide the complexity from user)
- Without an operating system, a computer is useless.

# The Role of OS

- OS exploits the hardware resources of one or more processors to provide a set of services to system users
- OS manages secondary memory and I/O devices on behalf of its users
- In short,
  - OS manages the computer's resources, such as the central processing unit, memory, disk drives, and printers
  - establishes a user interface
  - executes and provides services for applications software.

# OS

- A general –purpose, modern OS can exceed 50 million lines of code
- New OS are being written all the time
  - E-book reader
  - Tablet
  - Smartphone
  - Mainframe
  - Server
  - PC
  - .....

# Why to learn OS?

- To be able to write concurrent code
- Resource management
- Analyze the performance
- To fully understand how your code works
- ....
- In short,
  - this class isn't to teach you how to CREATE an OS from scratch, but to teach you how an OS works

# *Unsolved problem*

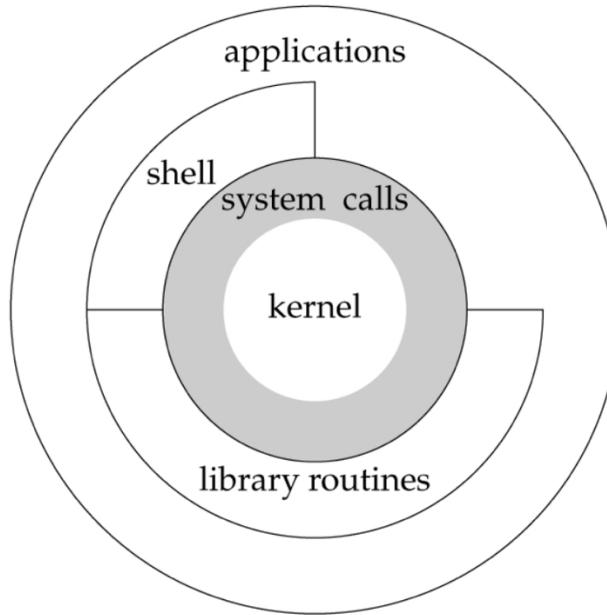
*Operating systems are an unsolved problem in computer science. Because;*

- *Most of them do not work well.*
  - Crashes, not fast enough, not easy to use, etc.
- *Usually they do not do everything they were designed to do.*
  - Needs are increasing every day
- *They do not adapt to changes so easily.*
  - New devices, processors, applications.
- .....

# Operating System Services

- execute a new program
- open a file
- read a file
- allocate a region of memory
- get the current time of day
- so on

# UNIX Architecture



**Figure 1.1** Architecture of the UNIX operating system

# Linux vs UNIX

- Linux refers to the kernel of the GNU/Linux operating system. More generally, it refers to the family of derived distributions.
- Unix refers to the original operating system developed by AT&T. More generally, it refers to family of derived operating systems.
- GNU/Linux and derivates like Debian and Fedora. System-V Unix and derivatives like IBM-AIX and HP-UX; Berkeley Unix and derivatives like FreeBSD and macOS
- Linux is broadly available as configurable software download and installer. UNIX is typically shipped along with hardware e.g. MacBook

# Working in the UNIX Environment

- UNIX like OS
  - Solaris
  - FreeBSD
  - macOS
  - NetBSD
  - ....
- Logging In
  - login name - password
  - password file
    - /etc/passwd

# Shells

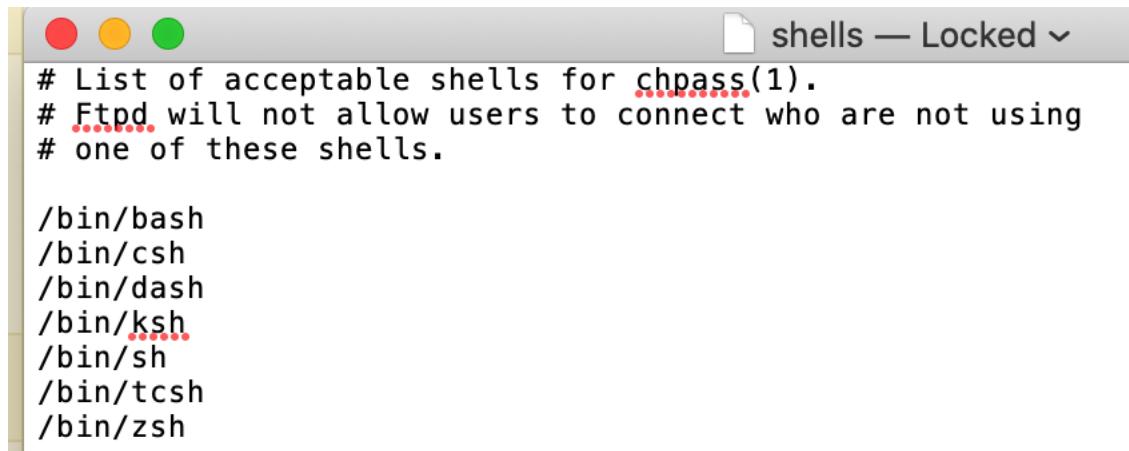
- A shell is the interface between the user and the kernel.
- Users can interact with the shell using shell commands in terminal or from a file (shell script).
- The common shells are;

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

# MacOS users

- Start the Terminal app on your Mac
- Terminal > Preferences, then click General.
- Under “Shells open with,” select “Command (complete path),” then enter the path to the shell you want to use.
- If you want to check the available shells in your mac;
  - go to /etc folder and check the shells file



The screenshot shows a terminal window with a title bar labeled "shells — Locked". The window contains the following text:

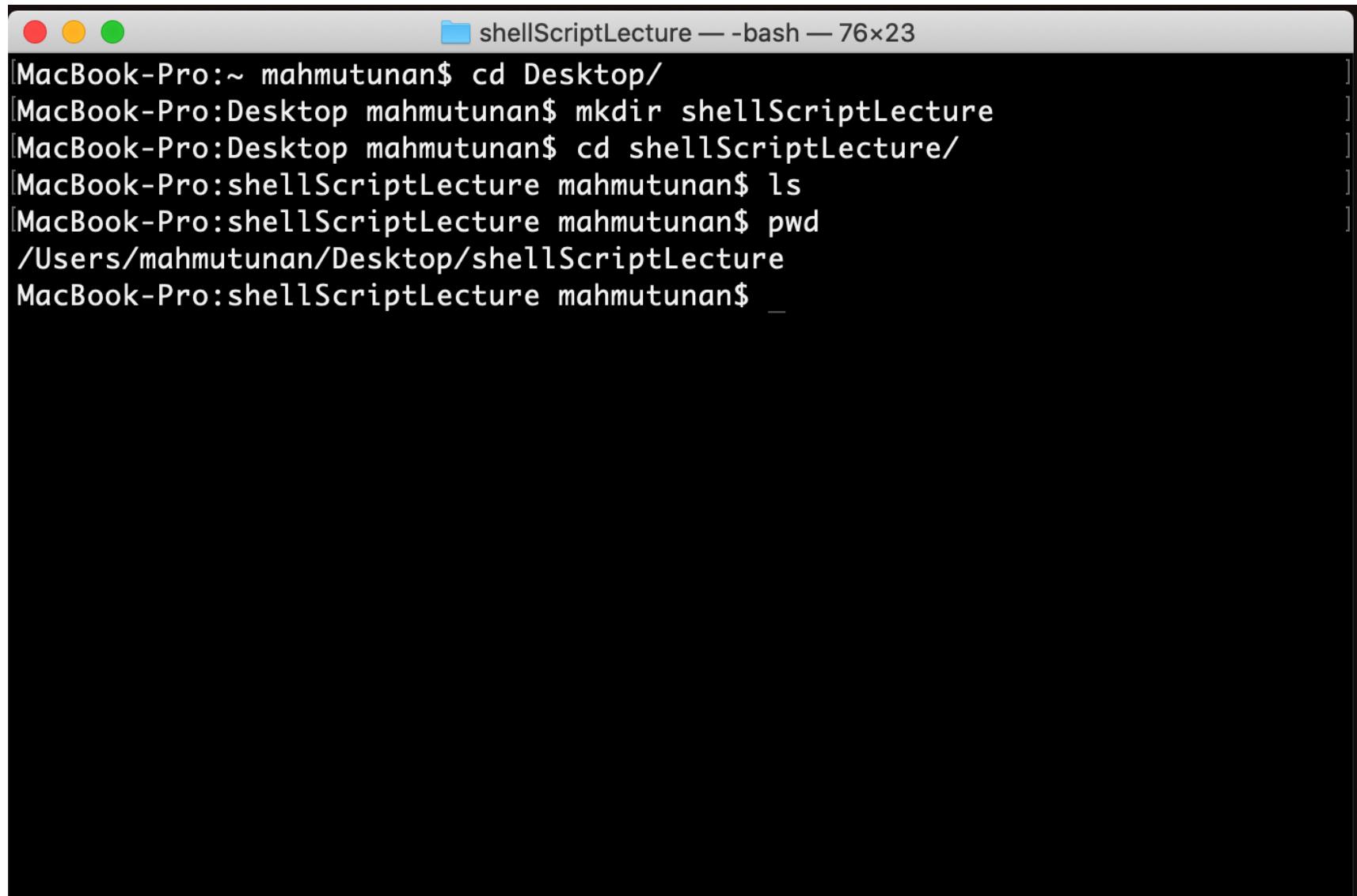
```
# List of acceptable shells for chpass(1).
# Ftpd will not allow users to connect who are not using
# one of these shells.

/bin/bash
/bin/csh
/bin/dash
/bin/ksh
/bin/sh
/bin/tcsh
/bin/zsh
```

# Windows Users

- Windows Subsystem for Linux
  - Bash Shell
- Git Bash
- <https://www.geeksforgeeks.org/use-bash-shell-natively-windows-10/>
- <https://www.howtogeek.com/249966/how-to-install-and-use-the-linux-bash-shell-on-windows-10/>

# Exercise 1 - first script file

A screenshot of a macOS terminal window titled "shellScriptLecture — -bash — 76x23". The window shows a series of terminal commands being run:

```
[MacBook-Pro:~ mahmutunan$ cd Desktop/  
[MacBook-Pro:Desktop mahmutunan$ mkdir shellScriptLecture  
[MacBook-Pro:Desktop mahmutunan$ cd shellScriptLecture/  
[MacBook-Pro:shellScriptLecture mahmutunan$ ls  
[MacBook-Pro:shellScriptLecture mahmutunan$ pwd  
/Users/mahmutunan/Desktop/shellScriptLecture  
MacBook-Pro:shellScriptLecture mahmutunan$ _
```

The terminal has its standard red, yellow, and green close buttons at the top left. The title bar is light gray with the window title and dimensions.

# .sh file

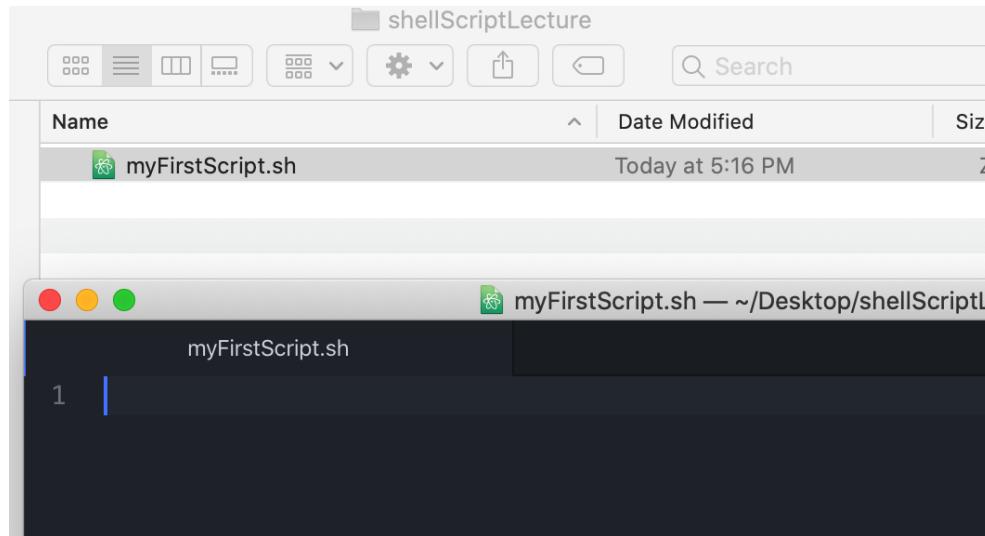
- It is a script programmed for bash
  - It contains instructions written in the Bash language
  - It can be executed by typing text commands within the shell's command-line interface.
- How to run the .sh file?
  - First, give the execute permission
    - chmod 755 somefilename.sh
  - Next, run your script file
    - sh somefilename.sh
    - bash somefilename.sh
    - ./somefilename.sh
    - if you want to run it as a root user
      - sudo bash somefilename.sh

# myFirstScript.sh

- You can use your terminal to create the file and use nano to edit the file

```
[MacBook-Pro:shellScriptLecture mahmutunan$ touch myFirstScript.sh  
[MacBook-Pro:shellScriptLecture mahmutunan$ nano myFirstScript.sh
```

- OR, you can use any editor to create and edit the file



## myFirstScript.sh

```
1 #!/bin/bash
2
3 # some comment
4 echo Hello CS332!!!
5
6 LECTURE="Lecture 8"
7 echo "This is $LECTURE"
8
9 echo -n "How old are you: "
10 read AGE
11 if [[ $AGE -ge 18 ]]
12 then
13     echo "You can vote"
14 else
15     echo "You are not eligible to vote"
16 fi
17
```

## myFirstScript.sh

```
1 #!/bin/bash
2
3 # some comment
4 echo Hello CS332!!!
5
6 LECTURE="Lecture 8"
7 echo "This is $LECTURE"
8
```

```
[MacBook-Pro:shellScriptLecture mahmutunan$ bash myFirstScript.sh
Hello CS332!!!
This is Lecture 8
How old are you: 21
You can vote
[MacBook-Pro:shellScriptLecture mahmutunan$ bash myFirstScript.sh
Hello CS332!!!
This is Lecture 8
How old are you: 11
You are not eligible to vote
MacBook-Pro:shellScriptLecture mahmutunan$ ]
```

# FILE Conditions

## File operators

Operator	Note
<code>-e</code>	To check if the file exists.
<code>-r</code>	To check if the file is readable.
<code>-w</code>	To check if the file is writable.
<code>-x</code>	To check if the file is executable.
<code>-s</code>	To check if the file size is greater than 0.
<code>-d</code>	To check if the file is a directory.

```
1 #!/bin/sh
2
3 FILE_NAME="someFileThatDoesntExist.txt"
4
5 # check
6 if [ -e $FILE_NAME ]
7 then
8     echo "Heyyooo, the file exists!"
9 else
10    echo "OOPSSS, the file does not exists!"
11 fi
```

```
[MacBook-Pro:shellScriptLecture mahmutunan$ bash fileOperations.sh
OOPSSS, the file does not exists!
```

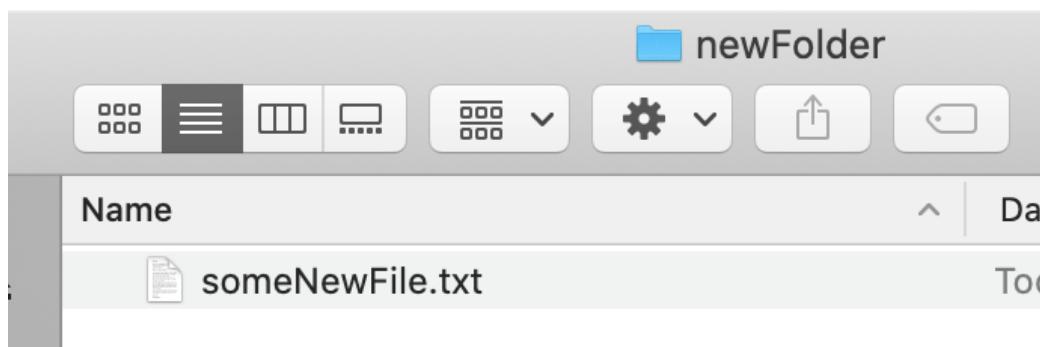
# Loops & Arrays

```
MY_COURSES="CS203 CS330 CS332"  
for COURSE in $MY_COURSES  
do  
    echo $COURSE  
done
```

CS203  
CS330  
CS332

```
mkdir newFolder  
touch "newFolder/someNewFile.txt"  
echo "This message goes to the file" >> "newFolder/someNewFile.txt"  
echo "This message appears on the terminal"
```

This message appears on the terminal  
MacBook-Pro:shellScriptLecture mahmutunan\$



```
1 #!/bin/sh
2 clear
3 echo "Current Directory :"
4 pwd
5 echo "What is in this directory? :"
6 ls
7
8 head "myFirstScript.sh"
9
10 echo "Disk Usage :"
11 df -h
12
13 exit
14
```

```

Current Directory :
/Users/mahmutunan/Desktop/shellScriptLecture
What is in this directory? :
exercise3.sh           fileOperations.sh      myFirstScript.sh      newFolder
#!/bin/bash

# some comment
echo Hello CS332!!!

LECTURE="Lecture 8"
echo "This is $LECTURE"

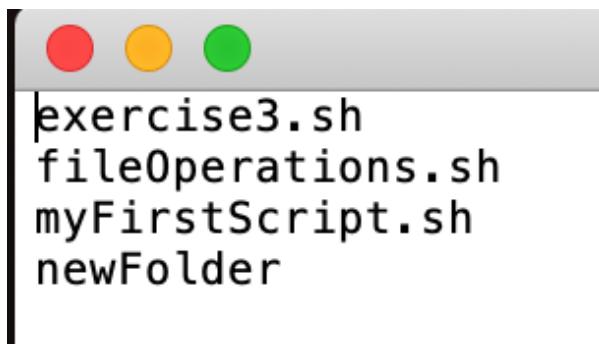
echo -n "How old are you: "
read AGE

Disk Usage :
Filesystem              Size   Used  Avail Capacity iused ifree %iused
/dev/disk1s6             466Gi  10Gi  221Gi    5%  488316 4881964564  0%
devfs                   231Ki  231Ki  0Bi     100%  800      0  100%
/dev/disk1s1             466Gi  208Gi  221Gi   49% 1063805 4881389075  0%
/dev/disk1s4             466Gi  15Gi   221Gi    7%   15 4882452865  0%
/dev/disk1s5             466Gi  10Gi   221Gi    5%  487048 4881965832  0%
map auto_home            0Bi    0Bi   0Bi     100%   0      0  100%
ome
Box                      466Gi  208Gi  221Gi   49% 1063805 586421779  0%
/Users/mahmutunan/Documents/Atom.app 466Gi  199Gi  243Gi   45% 971867 4881481013  0%
k/l02njqgs56v17md44tfknd7c0000gn/T/AppTranslocation/71F900AC-7A89-4E16-BC6F-66BBF16E283E
/dev/disk1s3             466Gi  1.0Gi  221Gi    1%   94 4882452786  0%
/dev/disk3s1             309Mi  229Mi  80Mi    75% 1433 4294965846  0%
MacBook-Pro:shellScriptLecture mahmutunan$ 

```

# I/O Redirection

- Regular UNIX system commands;
  - take input from terminal (stdin)
  - writes output to terminal (stdout)
- Output redirection
  - Output to a file
  - > filename notation will be used
  - ls >> "newFolder/someNewFile.txt"



- Input Redirection
  - < filename

```
Mail -s "Subject" to-address < Filename
```

Attachment File →  
guru99@virtualBox:~\$ mail -s "News Today" abc@ymail.com < NewsFlash  
E-mail Subject ↑      E-mail Address ↑

# Man Page

\$man command

\$man cat

CAT(1)	BSD General Commands Manual	CAT(1)
<b>NAME</b>		
<b>cat</b> -- concatenate and print files		
<b>SYNOPSIS</b>		
<b>cat</b> [ <b>-benstuv</b> ] [ <u>file</u> ...]		
<b>DESCRIPTION</b>		
The <b>cat</b> utility reads files sequentially, writing them to the standard output. The <u>file</u> operands are processed in command-line order. If <u>file</u> is a single dash (`-') or absent, <b>cat</b> reads from the standard input. If <u>file</u> is a UNIX domain socket, <b>cat</b> connects to it and then reads it until EOF. This complements the UNIX domain binding capability available in <i>inetd(8)</i> .		
The options are as follows:		
<b>-b</b>	Number the non-blank output lines, starting at 1.	
<b>-e</b>	Display non-printing characters (see the <b>-v</b> option), and display a dollar sign (`\$') at the end of each line.	
<b>-n</b>	Number the output lines, starting at 1.	
<b>-s</b>	Squeeze multiple adjacent empty lines, causing the output to be single spaced.	
<b>-t</b>	Display non-printing characters (see the <b>-v</b> option), and display tab characters as `^I'.	
<b>-u</b>	Disable output buffering.	
<b>-v</b>	Display non-printing characters so they are visible. Control characters print as `^X' for control-X; the delete character (octal 0177) prints as `^?'. Non-ASCII characters	

# UNIX Files System

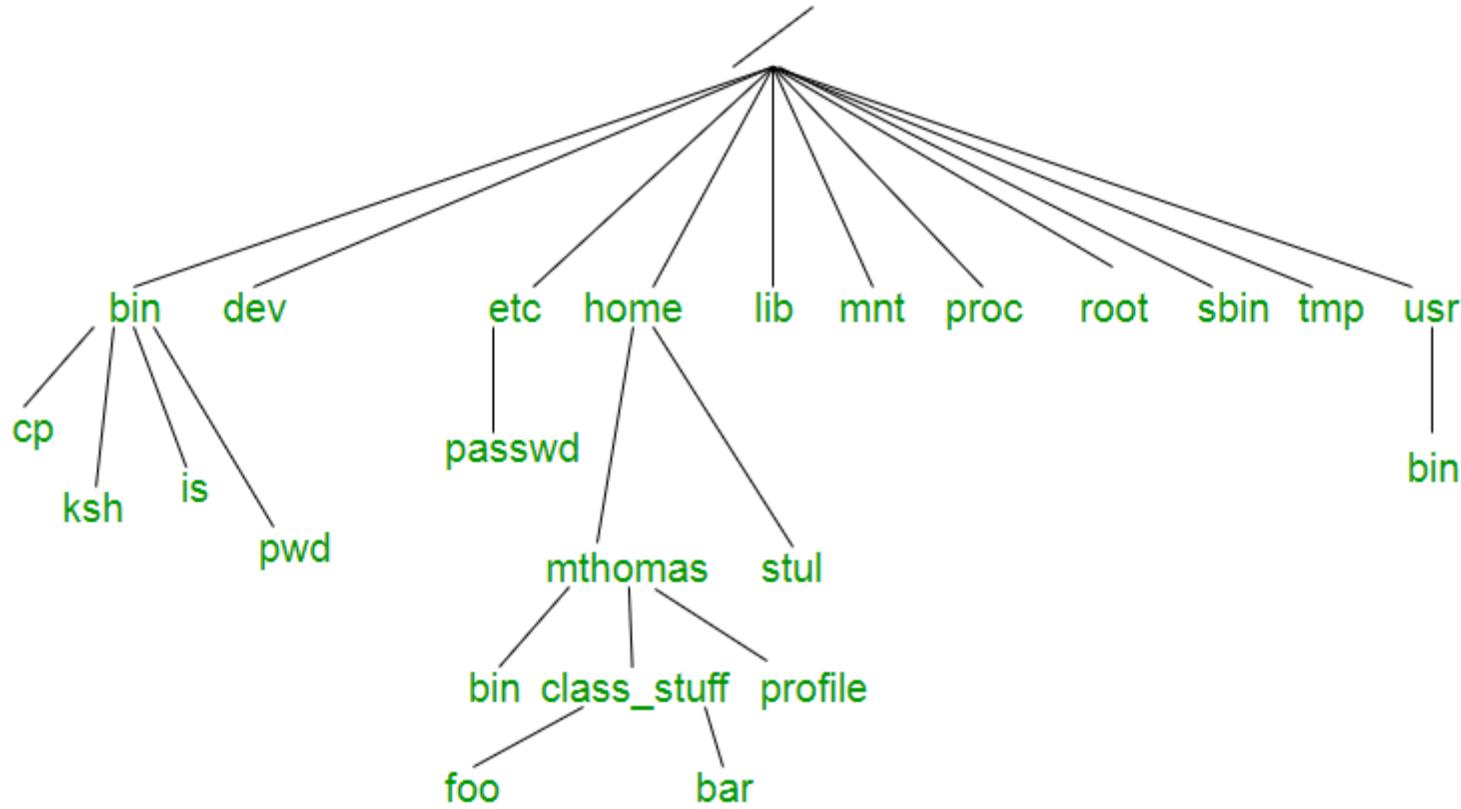
- All data organized into files
- All files organized into directories
- Directories
  - Tree-like structures
  - Multi-level hierarchy (directory tree)
  - Top level - root
    - /
    - All other directories are the children of the root
- File = sequence of bytes

# Windows vs Unix

- Windows files are stored in folders on different data drives
  - C: D: E: ....
- Unix files are ordered in a tree structure
  - root and the children
- Peripherals such as hard drives, CD-Rom, printers, scanners
  - Windows consider them as devices
  - Unix consider them as files
- Naming convention
  - Windows → UAB and uab are the same, can't be under the same folder
  - Unix → they are different; UAB, uab, Uab, uAb...

# Unix Directory Structure

- A file system consists of files, relationships to other files, as well as the attributes of each file
- root contains other files and directories
- each file or directory
  - uniquely identified by;
    - name
    - the directory that contains the file/directory
    - unique identifier (inode)
  - includes;
    - file type, size, owner, protection/privacy. time stamp
- each file is self contained



# Listing the names of all files

---

```
#include "apue.h"
#include <dirent.h>

int
main(int argc, char *argv[ ])
{
    DIR            *dp;
    struct dirent  *dirp;

    if (argc != 2)
        err_quit("usage: ls directory_name");

    if ((dp = opendir(argv[1])) == NULL)
        err_sys("can't open %s", argv[1]);
    while ((dirp = readdir(dp)) != NULL)
        printf("%s\n", dirp->d_name);

    closedir(dp);
    exit(0);
}
```

---

Figure 1.3 List all the files in a directory

# File Types

- Directory Files
- Ordinary Files
  - text
  - data
  - program instruction....
  - ...
- Special Files
  - devices
  - shortcuts
  - ...

# Unix File Descriptors

- It is a non-negative integer number that uniquely identifies an open file in Unix.
- It describes a data resource, and how that resource may be accessed.
- The first three file descriptors;
  - STDIN (standard input). 0
  - STDOUT (standard output). 1
  - STDERR (standard error). 2

# File Permissions

- use `ls -l` command to display the permissions
  - r w x - → read write execute no permission
- Owners permission -> First three characters
- Group permission -> next three characters
- World (other) Permission -> last three characters
- For example;

**-rwxrw-r--**

**drwxr-xr--**

# Change Permission

- Use the chmod command to set permissions (read, write, execute) on a file/directory for the owner, group and the world

Number	Permission Type	Symbol
0	No Permission	---
1	Execute	--x
2	Write	-w-
3	Execute + Write	-wx
4	Read	r--
5	Read + Execute	r-x
6	Read +Write	rw-
7	Read + Write +Execute	rwx

# UNIX file I/O Functionalities

- Processes needs system calls to handle file operation
  - Opening a file
    - open or openat functions
- fd = open(path, flag, mode)

[https://man7.org/linux/man-  
pages/man2/open.2.html](https://man7.org/linux/man-pages/man2/open.2.html)

# flag

O\_RDONLY

O\_WRONLY

O\_RDWR

O\_EXEC

O\_APPEND

O\_CREAT

.....

.....

# mode

- The third argument → *mode* specifies the permissions to use in case a new file is created.

The following symbolic constants are provided for *mode*:

```
S_IRWXU 00700 user (file owner) has read, write, and execute  
        permission  
  
S_IRUSR 00400 user has read permission  
  
S_IWUSR 00200 user has write permission  
  
S_IXUSR 00100 user has execute permission  
  
S_IRWXG 00070 group has read, write, and execute permission  
  
S_IRGRP 00040 group has read permission  
  
S_IWGRP 00020 group has write permission  
  
S_IXGRP 00010 group has execute permission  
  
S_IRWXO 00007 others have read, write, and execute permission  
  
S_IROTH 00004 others have read permission  
  
S_IWOTH 00002 others have write permission  
  
S_IXOTH 00001 others have execute permission
```

According to POSIX, the effect when other bits are set in *mode* is unspecified. On Linux, the following bits are also honored in *mode*:

```
S_ISUID 0004000 set-user-ID bit  
  
S_ISGID 0002000 set-group-ID bit (see inode\(7\)).  
  
S_ISVTX 0001000 sticky bit (see inode\(7\)).
```

# close

- system call
- the file descriptor is returned to the pool of available descriptors

```
int close(int fd);
```

- `close()` returns zero on success. On error, -1 is returned, and `errno` is set appropriately.

# read()

- **ssize\_t read(int *fd*, void \**buf*, size\_t *count*);**
- **read()** attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.
- On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number.

# write()

- **ssize\_t write(int *fd*, const void \**buf*, size\_t *count*);**
- **write()** writes up to *count* bytes from the buffer starting at *buf* to the file referred to by the file descriptor *fd*.
- On success, the number of bytes written is returned. On error, -1 is returned, and ***errno*** is set to indicate the cause of the error.

# Exercise

- C code to copy one file and copy the contents of that file to a new file

```
1  #include <stdio.h>
2
3  #include <stdlib.h>
4
5  #include <unistd.h>
6
7  #include <fcntl.h>
8
9
10 #define BUFFSIZE 4096
11
12
13 int main(int argc, char *argv[]) {
14     int readFileDescriptor, writeFileDescriptor;
15     long int n;
16     char buf[BUFFSIZE];
```

- Check if the correct numbers of the argument are given. There should be three arguments: name of the program, input file name, and output file name. If the number of arguments is not three then the program should print an error message and terminate. Also, input and output file names should not be the same.

```
12
13     if (argc != 3){
14         printf("Usage: %s <source_filename> <destination_filename>\n", argv[0]);
15         exit (-1);
16     }
17 }
```

- Use the *open* function in *read only mode* to read the input file.
- The open function takes the name of the file as the first argument and the open flag as the second argument.
- The open flag specifies if the file should be opened in read only mode (O\_RDONLY), write only mode (O\_WRONLY), or read-write mode (O\_RDWR).
- There is an optional third argument that specifies the file permissions called *mode*, we will not use the optional third argument here. The third argument specifies the file permissions of the new file created for writing. Note that the UNIX file uses {*read, write, execute*} (rwx) permissions for the user, group, and everyone

```
readFileDescriptor = open(argv[1], O_RDONLY);
```

- The function returns a file descriptor which is typically a non-negative integer.
- If there is an error opening the file, the function returns -1.
- Note that most programs have access to three standard file descriptors: standard input (stdin) - 0, standard output (stdout) - 1, and standard error (stderr) - 2.
- Instead of using the values 0, 1, and 2 we can also use the POSIX name STDIN\_FILENO, STDOUT\_FILENO, and STDERR\_FILENO, respectively.

- Use the open function to create a new write for writing the output such that if the file does not exist it will create a new file and if a file with the given name exists it will overwrite the existing file.
- This is accomplished by ORing the different open flags: O\_CREAT, O\_WRONLY, and O\_TRUNC. O\_CREAT specifies to open a new empty file if the file does not exist and requires the third file permission argument to be provided.
- O\_WRONLY specifies that the file should be open for writing only and the O\_TRUNC flag specifies that if the file already exists, then it must be truncated to zero length (i.e., destroy any previous data).

```
19 |     writeFileDescriptor = open(argv[2], O_CREAT|O_WRONLY|O_TRUNC, 0700);
```

- Check the file descriptor to see if there is a problem or not

```
18     readFileDescriptor = open(argv[1], O_RDONLY);
19     writeFileDescriptor = open(argv[2], O_CREAT|O_WRONLY|O_TRUNC, 0700);
20
21     if (readFileDescriptor == -1 || writeFileDescriptor == -1){
22         printf("Error with file open\n");
23         exit (-1);
24 }
```

- Now read the file by reading fixed chunks of data using the *read* function by providing the file descriptor, input buffer address, and the maximum size of the buffer provided .
- A successful read returns the number of bytes read, 0 if the end-of-file is reached, or -1 if there is an error.
- After you read the data in to the buffer write the buffer to the new file using the *write* function.
- The *write* function takes the file descriptor, buffer to write, and the number of bytes to write. If the write is successful, it will return the number of bytes actually written.

```
26     while ((n = read(readFileDescriptor, buf, BUFFSIZE)) > 0){  
27         if (write(writeFileDescriptor, buf, n) != n){  
28             printf("Error writing to output file\n");  
29             exit (-1);  
30     }  
31 }
```

- check the error condition

```
32     if (n < 0){  
33         printf("Error reading input file\n");  
34         exit (-1);  
35     }  
36 }
```

- After completing the copy process use the *close* function to close both file descriptors. The *close* function takes the file descriptor as the argument and returns 0 on success and -1 in case of an error.

```
38     close(readFileDescriptor);  
39     close(writeFileDescriptor);  
40     return 0;  
41 }
```

# Exercise

- C code to copy one file and copy the contents of that file to a new file

```
1  #include <stdio.h>
2
3  #include <stdlib.h>
4
5  #include <unistd.h>
6
7  #include <fcntl.h>
8
9
10 #define BUFFSIZE 4096
11
12
13 int main(int argc, char *argv[]) {
14     int readFileDescriptor, writeFileDescriptor;
15     long int n;
16     char buf[BUFFSIZE];
```

- Check if the correct numbers of the argument are given. There should be three arguments: name of the program, input file name, and output file name. If the number of arguments is not three then the program should print an error message and terminate. Also, input and output file names should not be the same.

```
12
13     if (argc != 3){
14         printf("Usage: %s <source_filename> <destination_filename>\n", argv[0]);
15         exit (-1);
16     }
17 }
```

- Use the *open* function in *read only mode* to read the input file.
- The open function takes the name of the file as the first argument and the open flag as the second argument.
- The open flag specifies if the file should be opened in read only mode (O\_RDONLY), write only mode (O\_WRONLY), or read-write mode (O\_RDWR).
- There is an optional third argument that specifies the file permissions called *mode*, we will not use the optional third argument here. The third argument specifies the file permissions of the new file created for writing. Note that the UNIX file uses {*read, write, execute*} (rwx) permissions for the user, group, and everyone

```
readFileDescriptor = open(argv[1], O_RDONLY);
```

- The function returns a file descriptor which is typically a non-negative integer.
- If there is an error opening the file, the function returns -1.
- Note that most programs have access to three standard file descriptors: standard input (stdin) - 0, standard output (stdout) - 1, and standard error (stderr) - 2.
- Instead of using the values 0, 1, and 2 we can also use the POSIX name STDIN\_FILENO, STDOUT\_FILENO, and STDERR\_FILENO, respectively.

- Use the open function to create a new write for writing the output such that if the file does not exist it will create a new file and if a file with the given name exists it will overwrite the existing file.
- This is accomplished by ORing the different open flags: O\_CREAT, O\_WRONLY, and O\_TRUNC. O\_CREAT specifies to open a new empty file if the file does not exist and requires the third file permission argument to be provided.
- O\_WRONLY specifies that the file should be open for writing only and the O\_TRUNC flag specifies that if the file already exists, then it must be truncated to zero length (i.e., destroy any previous data).

```
19 |     writeFileDescriptor = open(argv[2], O_CREAT|O_WRONLY|O_TRUNC, 0700);
```

- Check the file descriptor to see if there is a problem or not

```
18     readFileDescriptor = open(argv[1], O_RDONLY);
19     writeFileDescriptor = open(argv[2], O_CREAT|O_WRONLY|O_TRUNC, 0700);
20
21     if (readFileDescriptor == -1 || writeFileDescriptor == -1){
22         printf("Error with file open\n");
23         exit (-1);
24 }
```

- Now read the file by reading fixed chunks of data using the *read* function by providing the file descriptor, input buffer address, and the maximum size of the buffer provided .
- A successful read returns the number of bytes read, 0 if the end-of-file is reached, or -1 if there is an error.
- After you read the data in to the buffer write the buffer to the new file using the *write* function.
- The *write* function takes the file descriptor, buffer to write, and the number of bytes to write. If the write is successful, it will return the number of bytes actually written.

```
26     while ((n = read(readFileDescriptor, buf, BUFFSIZE)) > 0){  
27         if (write(writeFileDescriptor, buf, n) != n){  
28             printf("Error writing to output file\n");  
29             exit (-1);  
30     }  
31 }
```

- check the error condition

```
32     if (n < 0){  
33         printf("Error reading input file\n");  
34         exit (-1);  
35     }  
36 }
```

- After completing the copy process use the *close* function to close both file descriptors. The *close* function takes the file descriptor as the argument and returns 0 on success and -1 in case of an error.

```
38     close(readFileDescriptor);  
39     close(writeFileDescriptor);  
40     return 0;  
41 }
```

# Run the example

```
gcc filecopy.c -o exercise1
```

```
./exercise1 smallTale.txt outputFile.txt
```

```
[MacBook-Pro:Desktop mahmutunan$ gcc filecopy.c -o exercise1
[MacBook-Pro:Desktop mahmutunan$ ./exercise1 smallTale.txt outputFile.txt
MacBook-Pro:Desktop mahmutunan$ ]
```



# `lseek()`

```
off_t lseek(int fd, off_t offset, int whence);
```

- repositions the file offset of the open file description associated with the file descriptor *fd* to the argument ***offset*** according to the directive ***whence*** as follows:

<https://man7.org/linux/man-pages/man2/lseek.2.html>

- **SEEK\_SET** The file offset is set to *offset* bytes.
- **SEEK\_CUR** The file offset is set to its current location plus *offset* bytes.
- **SEEK\_END** The file offset is set to the size of the file plus *offset* bytes.

# Example 2

- Now, we will use `fseek` function to move to particular location in the file and modify it by performing following steps;
  1. You can use the output file of Example #1.  
Or, you can use any txt file.  
I will create a new txt file and put some text

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <fcntl.h>
5 #include <string.h>
6
7 #define BUFSIZE 4096
8 #define SEEKSIZE -10
9
10 ► int main(int argc, char *argv[]) {
11     int RWFileDescriptor;
12     long int n;
13     char buf[BUFSIZE];
14     const char lseekMSG[] = "THIS IS NEW MSG FROM LSEEK!\0";
15
16     if (argc != 2){
17         printf("Usage: %s <filename>\n", argv[0]);
18         exit (-1);
19     }
20
21     RWFileDescriptor = open(argv[1], O_RDONLY);
22
23     if (RWFileDescriptor == -1){
24         printf("Error with file open\n");
25         exit (-1);
26     }
27
```

- 2. Use lseek to read last 10 bytes of file and print it on console.
  - The *lseek* function takes three arguments: file descriptor, the file offset, and the base address from which the offset is to be implemented (often referred to as *whence*).
  - In this example, we are trying to read the last 10 bytes in the file, so we set the whence to the end of the file using SEEK\_END and specify the offset as 10.
  - You can look at the man page for *lseek* to find out other predefined whence values: SEEK\_SET, SEEK\_CUR, etc.
  - The *lseek* function returns the new file offset on a successful and returns -1 when there is an error.

```
28     if (lseek(RWFileDescriptor, SEEKSIZE, SEEK_END) >= 0){
29         if((n = read(RWFileDescriptor, buf, BUFFSIZE)) > 0){
30             if (write(STDOUT_FILENO, buf, n) != n) {
31                 printf("Error writing to file\n");
32                 exit (-1);
33             }
34         } else {
35             printf("Error reading file\n");
36             exit (-1);
37         }
38     } else {
39         printf("lseek error (Part 1)\n");
40         exit (-1);
41     }
42     close(RWFileDescriptor);
```

- 3. Use lseek to write a string character “THIS IS NEW MSG FROM LSEEK!” at the beginning of the file.

```
43
44     RWFileDescriptor = open(argv[1], O_WRONLY);
45     if (lseek(RWFileDescriptor, 0, SEEK_SET) >= 0){
46         if (write(RWFileDescriptor, lseekMSG, strlen(lseekMSG)) != strlen(lseekMSG))
47             printf("Error writing to file\n");
48     }
49 } else {
50     printf("lseek error (Part 2)\n";
51 }
52
53 close(RWFileDescriptor);
54
55 return 0;
56 }
```

# Run the exercise 2

```
(base) mahmutunan@MacBook-Pro Desktop % gcc fileseek.c -o exercise2  
(base) mahmutunan@MacBook-Pro Desktop % cat testfile.txt  
THIS IS NEW MSG FROM LSEEK!s a test message. Do you see it?%
```

```
(base) mahmutunan@MacBook-Pro Desktop % ./exercise2 testfile.txt
```

```
(base) mahmutunan@MacBook-Pro Desktop % cat testfile.txt  
THIS IS NEW MSG FROM LSEEK!s a test message. Do you see it?%
```

# Lab 4 - exercise

- Check out what values are printed if you change the *offset* and *whence* to the following values when you are using the *lseek* function with the *readFileDescriptor*:

<b>offset</b>	<b>whence</b>
0	SEEK_SET
0	SEEK_END
-1	SEEK_END
-10	SEEK_CUR

# Exercise 3

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <fcntl.h>
5 #define BUF_SIZE 1024
6 |
7 int main(int argc, char *argv[]) {
8     if (argc != 2) {
9         printf("Usage: %s <filename>\n", argv[0]);
10        exit(-1);
11    }
12    char *file_Name = argv[1];
13
14    int writeFd;
```

```
15
16     fprintf(stdout, "Opening file :%s\n", file_Name);
17     writeFd = open(file_Name, O_RDWR, O_NONBLOCK, O_APPEND);
18
19     if (writeFd < 0) {
20         printf("Error with file open\n");
21         exit(-1);
22     }
23
24     fprintf(stdout, "Seeking the beginning of the file \n");
```

```
25
26     if (lseek(writeFd, 0, SEEK_SET) >= 0) {
27         fprintf(stdout, "Writing the message into %s\n", file_Name);
28         char buffer[BUF_SIZE] = "Message from Exercise 3\n";
29         write(writeFd, buffer, BUF_SIZE);
30         close(writeFd);
31
32         return 0;
33     }
34 }
```

# output

```
(base) mahmutunan@MacBook-Pro Desktop % gcc exercise3.c -o exercise3
(base) mahmutunan@MacBook-Pro Desktop % ./exercise3 output.txt
Opening file :output.txt
Seeking the beginning of the file
Writing the message into output.txt
(base) mahmutunan@MacBook-Pro Desktop %
```

# *Make Utility*

- *make* is a utility that is used to automatically detect which program need to be recompiled while working on a large number of source programs and will recompile only those programs that have been modified.
- The *make* utility uses a *Makefile* to describe the rules for determining the dependencies between the various programs and the compiler and compiler options to use for compiling the programs.
- In case of C programs, an executable is created from object files (\*.o files) and object files are created from source files.
- Source files are often divided into header files (\*.h files) and actual source files (\*.c files).

- The simplest way to organize the code compilation
- Make figures out automatically which files it needs to update, based on which source files have changed.

# Exercise 1

```
1 #include <stdio.h>
2
3 int main (int argc, char *argv[]){
4
5     char charArr[20] = "Hello CS330";
6     printf("%s",charArr);
7     return 0;
8
9 }
```

```
1 myExecutable: main.c
2     gcc -o myExecutable main.c
3
```

```
(base) mahmutunan@MacBook-Pro exercise1 % ./myExecutable
(base) mahmutunan@MacBook-Pro exercise1 % ls
Makefile      main.c
(base) mahmutunan@MacBook-Pro exercise1 % make
gcc -o myExecutable main.c
(base) mahmutunan@MacBook-Pro exercise1 % ./myExecutable
Hello CS330%
(base) mahmutunan@MacBook-Pro exercise1 %
```

# Exercise 2

```
1 #include "someFunc.h"
2
3 int main (int argc, char *argv[]){
4
5     printHellofunc();
6     return 0;
7
8 }
```

- main.c

```
1 #include <stdio.h>
2 #include "someFunc.h"
3
4 void printHellofunc(){
5
6     printf("Hello CS330");
7
8 }
```

- someFunc.c

```
1 void printHellofunc();
2
```

- someFunc.h

# Exercise 2

- we can compile multiple file using command line

```
(base) mahmutunan@MacBook-Pro exercise2 % ls  
main.c          someFunc.c      someFunc.h  
(base) mahmutunan@MacBook-Pro exercise2 % gcc -o exercise2 main.c someFunc.c -I  
(base) mahmutunan@MacBook-Pro exercise2 % ./exercise2  
Hello CS330%  
(base) mahmutunan@MacBook-Pro exercise2 %
```

- The -I. is included so that gcc will look in the current directory (.) for the include file someFunc.h

# Exercise 2

- We can use the make file to automate the compilation process

```
Makefile  
1 exercise2withMake: main.c someFunc.c  
2     gcc -o exercise2withMake main.c someFunc.c -I  
3
```

```
(base) mahmutunan@MacBook-Pro exercise2 % ls  
Makefile      exercise2      main.c      someFunc.c      someFunc.h  
(base) mahmutunan@MacBook-Pro exercise2 % make  
gcc -o exercise2withMake main.c someFunc.c -I  
(base) mahmutunan@MacBook-Pro exercise2 % ./exercise2withMake  
Hello CS330%  
(base) mahmutunan@MacBook-Pro exercise2 %
```

## Makefile

```
1 CC=gcc
2 CFLAGS=-I
3 DEPS=someFunc.h
4 OBJ=main.o someFunc.o
5
6 %.o: %.c $(DEPS)
7     $(CC) -c -o $@ $< $(CFLAGS)
8
9 exercise2withMake: $(OBJ)
10    $(CC) -o $@ $^ $(CFLAGS)
```

# Exercise 2

- Let's modify the make file a little bit

```
[base] mahmutunan@MacBook-Pro exercise2 % ls
Makefile          main.c          someFunc.c      someFunc.h
[base] mahmutunan@MacBook-Pro exercise2 % make
gcc -c -o main.o main.c -I
gcc -c -o someFunc.o someFunc.c -I
gcc -o exercise2withMake main.o someFunc.o -I
[base] mahmutunan@MacBook-Pro exercise2 % ls
Makefile          main.o          someFunc.o
exercise2withMake
main.c           someFunc.c
someFunc.h
[base] mahmutunan@MacBook-Pro exercise2 % ./exercise2withMake
Hello CS330%
[base] mahmutunan@MacBook-Pro exercise2 %
```

# Exercise - Lab04

- To illustrate the use of make, let us consider adding a new function to measure the time taken by the insertion sort program that we wrote in Lab 2.
- Instead of adding this method to the same file as the insertion sort, let us create a new file and create a header file that has the method prototype.

# insertionsort.c

```
1 #include<stdio.h>
2 #include<stdlib.h>
3
4 /* main method */
5 int main(int args, char** argv){
6     int N, i;
7     printf("Please enter number of elements in array: ");
8     scanf("%d", &N);
9
10    float arr[N];
11
12    for (i=0; i<N; i++){
13        printf("Please enter element %d of array: ", (i+1));
14        scanf("%f", &arr[i]);
15    }
16
17    printf("Given array is: ");
18    printf("[");
19    for (i=0; i < N-1; i++){
20        printf("%f, ", arr[i]);
21    }
22    printf("%f]\n", arr[N-1]);
23
24    float temp;
25    int currLoc;
26    for (i=1; i < N; i++){
27        currLoc = i;
28        while (currLoc > 0 && arr[currLoc-1] > arr[currLoc]){
29            temp = arr[currLoc];
30            arr[currLoc] = arr[currLoc-1];
31            arr[currLoc-1] = temp;
32            currLoc--;
33        }
34    }
35
36    printf("Sorted array is: ");
37    printf("[");
38    for (i=0; i < N-1; i++){
39        printf("%f, ", arr[i]);
40    }
41    printf("%f]\n", arr[N-1]);
42
43    return 0;
44 }
```

# gettime.h

```
gettime.h
```

```
1 #ifndef _GETTIME_H_
2 #include <stdio.h>
3 #include <sys/time.h>
4 double gettime(void);
5 #endif
```

# gettime.c

gettime.c

```
1 #include "gettime.h"
2
3 double gettime(void) {
4     struct timeval tval;
5     gettimeofday(&tval, NULL);
6     return((double)tval.tv_sec + (double)tval.tv_usec/1000000.0);
7 }
8
```

- Note that we can compile the file `gettime.c` separately and link the object file with any other program that uses the ***gettime*** function.
- To use the `gettime` function in the insertion sort program, we have to include the file `gettime.h` and invoke the ***gettime*** function before and after the call to ***insertionsort*** function

- Here are the steps involved in incrementally compiling and linking these two different files:

```
(base) mahmutunan@MacBook-Pro Desktop % cd lecture11
(base) mahmutunan@MacBook-Pro lecture11 % ls
gettme.c      gettime.h      insertionsort.c
(base) mahmutunan@MacBook-Pro lecture11 % gcc -c gettime.c
(base) mahmutunan@MacBook-Pro lecture11 % gcc -c insertionsort.c
(base) mahmutunan@MacBook-Pro lecture11 % gcc -o insertionsort insertionsort.o gettime.o
(base) mahmutunan@MacBook-Pro lecture11 % ./insertionsort
Please enter number of elements in array: 7
Please enter element 1 of array: 21
Please enter element 2 of array: 22
Please enter element 3 of array: 44
Please enter element 4 of array: 55
Please enter element 5 of array: 32
Please enter element 6 of array: 56
Please enter element 7 of array: 2
Given array is: [21.000000, 22.000000, 44.000000, 55.000000, 32.000000, 56.000000, 2.000000]
Sorted array is: [2.000000, 21.000000, 22.000000, 32.000000, 44.000000, 55.000000, 56.000000]
(base) mahmutunan@MacBook-Pro lecture11 %
```

gcc -c compiles source files without linking

```
$ gcc -c myfile.c
```

This compilation generates *myfile.o* object file.

- Also note that we don't have to recompile `gettime.c` if we are only making changes to the file `insertionsort.c`.
- These dependencies is what we can describe in a make file and let the make utility determine which files what been updated and recompile those files.

# makefile

Makefile

```
1 # Sample Makefile to compile C programs
2 CC = gcc
3 CFLAGS = -Wall -g #replace -g with -O when not debugging
4 DEPS = gettimeofday.h Makefile
5 OBJS = gettimeofday.o insertionsort.o
6 EXECS = insertionsort
7
8 all: $(EXECS)
9
10 %.o: %.c $(DEPS)
11         $(CC) $(CFLAGS) -c -o $@ $<
12
13 insertionsort: $(OBJS)
14         $(CC) $(CFLAGS) -o $@ $^
15
16 clean:
17         /bin/rm -i *.o $(EXECS)
18
19
20
```

- If the Makefile is saved as Makefile or makefile, you can invoke make utility by typing make.
- If you use a different file name other than Makefile or makefile then you have to specify the makefile using the -f option to make. If you type make, you should see the following output:

```
(base) mahmutunan@MacBook-Pro lecture11 % make  
gcc -Wall -g -c -o gettime.o gettime.c  
gcc -Wall -g -c -o insertionsort.o insertionsort.c  
gcc -Wall -g -o insertionsort gettime.o insertionsort.o
```

- If you change `gettime.h` then you should see all files recompiled and the following output:

```
(base) mahmutunan@MacBook-Pro lecture11 % touch gettime.h
(base) mahmutunan@MacBook-Pro lecture11 % make
gcc -Wall -g -c -o gettime.o gettime.c
gcc -Wall -g -c -o insertionsort.o insertionsort.c
gcc -Wall -g -o insertionsort gettime.o insertionsort.o
```

If you change `gettime.c` then you should see the following output:

```
(base) mahmutunan@MacBook-Pro lecture11 % touch gettime.c
(base) mahmutunan@MacBook-Pro lecture11 % make
gcc -Wall -g -c -o gettime.o gettime.c
gcc -Wall -g -o insertionsort gettime.o insertionsort.o
```

- However, if you only change `insertionsort.c` you will see the following output:

```
(base) mahmutunan@MacBook-Pro lecture11 % make  
gcc -Wall -g -c -o insertionsort.o insertionsort.c  
gcc -Wall -g -o insertionsort gettime.o insertionsort.o
```

- If you have not modified any files, if you execute `make`, you will see that following output:

```
(base) mahmutunan@MacBook-Pro lecture11 % make  
make: Nothing to be done for `all'.
```

# stat    lstat

- We'll start with the *stat* and *lstat* functions.
- Both function return a structure called *stat*, and members of stat structure provide information about the file or directory which was provided as the argument to these functions.

# stat lstat

- stat, fstat, lstat, fstatat - get file status

```
int stat(const char * pathname, struct stat  
* statbuf);
```

```
int fstat(int fd, struct stat * statbuf);
```

```
int lstat(const char * pathname, struct stat  
* statbuf);
```

- `stat()` and `fstatat()` retrieve information about the file pointed to by *pathname*
- `lstat()` is identical to `stat()`, except that if *pathname* is a symbolic link, then it returns information about the link itself, not the file that the link refers to.
- `fstat()` is identical to `stat()`, except that the file about which information is to be retrieved is specified by the file descriptor *fd*.

# The stat structure

- All of these system calls return a *stat structure*, which contains the following fields:

```
struct stat {  
    dev_t      st_dev;          /* ID of device containing file */  
    ino_t      st_ino;          /* Inode number */  
    mode_t     st_mode;         /* File type and mode */  
    nlink_t    st_nlink;        /* Number of hard links */  
    uid_t      st_uid;          /* User ID of owner */  
    gid_t      st_gid;          /* Group ID of owner */  
    dev_t      st_rdev;         /* Device ID (if special file) */  
    off_t      st_size;         /* Total size, in bytes */  
    blksize_t   st_blksize;       /* Block size for filesystem I/O */  
    blkcnt_t   st_blocks;        /* Number of 512B blocks allocated */  
  
    /* Since Linux 2.6, the kernel supports nanosecond  
     precision for the following timestamp fields.  
     For the details before Linux 2.6, see NOTES. */  
  
    struct timespec st_atim;    /* Time of last access */  
    struct timespec st_mtim;    /* Time of last modification */  
    struct timespec st_ctim;    /* Time of last status change */  
  
    #define st_atime st_atim.tv_sec      /* Backward compatibility */  
    #define st_mtime st_mtim.tv_sec  
    #define st_ctime st_ctim.tv_sec  
};
```

```
(base) mahmutunan@MacBook-Pro lecture12 % touch someNewFile.txt
(base) mahmutunan@MacBook-Pro lecture12 % echo "some text into file" > someNewFile.txt
(base) mahmutunan@MacBook-Pro lecture12 % cat someNewFile.txt
some text into file
(base) mahmutunan@MacBook-Pro lecture12 % stat -x someNewFile.txt
  File: "someNewFile.txt"
    Size: 20          FileType: Regular File
      Mode: (0644/-rw-r--r--)      Uid: ( 501/mahmutunan)  Gid: ( 20/ staff)
Device: 1,4   Inode: 10604554   Links: 1
Access: Mon Sep 21 11:44:20 2020
Modify: Mon Sep 21 11:44:18 2020
Change: Mon Sep 21 11:44:18 2020
(base) mahmutunan@MacBook-Pro lecture12 %
```

# File Types

- Regular File
- Directory File
- Block Special File
- Character Special File
- FIFO
- Socket
- Symbolic links

# The type of the file

Macro	Type of file
<code>S_ISREG()</code>	regular file
<code>S_ISDIR()</code>	directory file
<code>S_ISCHR()</code>	character special file
<code>S_ISBLK()</code>	block special file
<code>S_ISFIFO()</code>	pipe or FIFO
<code>S_ISLNK()</code>	symbolic link
<code>S_ISSOCK()</code>	socket

**Figure 4.1** File type macros in `<sys/stat.h>`

# Exercise 1 - printstat.c

```
1  /* function to print stat data structure */
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/types.h>
5  #include <sys/stat.h>
6  #include <unistd.h>
7  #include <time.h>
8
9  void printstat(struct stat sb) {
10     /* copied from the lstat man page example as is */
11     printf("File type:          ");
12
13     switch (sb.st_mode & S_IFMT) {
14         case S_IFBLK:   printf("block device\n");           break;
15         case S_IFCHR:   printf("character device\n");       break;
16         case S_IFDIR:   printf("directory\n");             break;
17         case S_IFIFO:   printf("FIFO/pipe\n");            break;
18         case S_IFLNK:   printf("symlink\n");               break;
19         case S_IFREG:   printf("regular file\n");          break;
20         case S_IFSOCK:  printf("socket\n");                break;
21         default:        printf("unknown?\n");             break;
22     }
```

# Exercise 1 - printstat.c /2

```
23     printf("I-node number:          %ld\n", (long) sb.st_ino);
24
25     printf("Mode:                  %lo (octal)\n",
26            (unsigned long) sb.st_mode);
27
28
29     printf("Link count:           %ld\n", (long) sb.st_nlink);
30     printf("Ownership:             UID=%ld    GID=%ld\n",
31            (long) sb.st_uid, (long) sb.st_gid);
32
33     printf("Preferred I/O block size: %ld bytes\n",
34            (long) sb.st_blksize);
35     printf("File size:              %lld bytes\n",
36            (long long) sb.st_size);
37     printf("Blocks allocated:       %lld\n",
38            (long long) sb.st_blocks);
39
40     printf("Last status change:    %s", ctime(&sb.st_ctime));
41     printf("Last file access:       %s", ctime(&sb.st_atime));
42     printf("Last file modification: %s", ctime(&sb.st_mtime));
43 }
44
45 }
```

# Exercise 1 - lstat.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/stat.h>
5 #include <unistd.h>
6
7 void printstat(struct stat statbuf);
8
9 int main(int argc, char **argv) {
10     int i;
11     struct stat buf;
12     char *ptr;
13
14     for (i = 1; i < argc; i++) {
15         printf("%s: ", argv[i]);
16         if (lstat(argv[i], &buf) < 0) {
17             printf("lstat error");
18             continue;
19     } |
```

# Exercise 1 - lstat.c /2

```
20         if (S_ISREG(buf.st_mode))
21             ptr = "regular";
22         else if (S_ISDIR(buf.st_mode))
23             ptr = "directory";
24         else if (S_ISCHR(buf.st_mode))
25             ptr = "character special";
26         else if (S_ISBLK(buf.st_mode))
27             ptr = "block special";
28         else if (S_ISFIFO(buf.st_mode))
29             ptr = "fifo";
30         else if (S_ISLNK(buf.st_mode))
31             ptr = "symbolic link";
32         else if (S_ISSOCK(buf.st_mode))
33             ptr = "socket";
34         else
35             ptr = "** unknown mode **";
36         printf("%s\n", ptr);
37
38     printstat(buf);
39 }
40 exit(0);
41 }
```

# Example 1 -compile&run

```
(base) mahmutunan@MacBook-Pro lecture12 % gcc -o exercise1 printstat.c lstat.c
(base) mahmutunan@MacBook-Pro lecture12 % ln -s /Users/mahmutunan/Desktop/ABEt_links.txt aSymbolicLink
(base) mahmutunan@MacBook-Pro lecture12 % mkdir newFolder
(base) mahmutunan@MacBook-Pro lecture12 % ls
aSymbolicLink  exercise1      lstat.c      newFolder      printstat.c      someNewFile.txt
(base) mahmutunan@MacBook-Pro lecture12 % ./exercise1 aSymbolicLink someNewFile.txt newFolder
```

```
aSymbolicLink: symbolic link
File type:           symlink
I-node number:      10610993
Mode:                120755 (octal)
Link count:          1
Ownership:           UID=501   GID=20
Preferred I/O block size: 4096 bytes
File size:            40 bytes
Blocks allocated:    0
Last status change:  Mon Sep 21 12:38:35 2020
Last file access:    Mon Sep 21 12:38:35 2020
Last file modification: Mon Sep 21 12:38:35 2020
```

# Example 1 -compile&run /2

```
someNewFile.txt: regular
File type:          regular file
I-node number:     10604554
Mode:              100644 (octal)
Link count:        1
Ownership:         UID=501   GID=20
Preferred I/O block size: 4096 bytes
File size:          20 bytes
Blocks allocated:  8
Last status change: Mon Sep 21 12:29:58 2020
Last file access:   Mon Sep 21 12:29:58 2020
Last file modification: Mon Sep 21 11:44:18 2020
```

```
newFolder: directory
File type:          directory
I-node number:     10611008
Mode:              40755 (octal)
Link count:        2
Ownership:         UID=501   GID=20
Preferred I/O block size: 4096 bytes
File size:          64 bytes
Blocks allocated:  0
Last status change: Mon Sep 21 12:38:39 2020
Last file access:   Mon Sep 21 12:38:39 2020
Last file modification: Mon Sep 21 12:38:39 2020
(base) mahmutunan@MacBook-Pro lecture12 %
```

# Open & Read the directories

- Till now we talked about how filesystem store files and directories information and access details. Now it's time to learn how to open and read the directories and traverse the file system. To achieve this task, you need to learn about three functions:
- *opendir* - this function will allow us to open a directory with the given path
- *readdir* - this function will read what's inside the directory
- *closedir* - this will close the open directory

# opendir

- opendir, fdopendir - open a directory

```
DIR *opendir(const char *name);  
DIR *fdopendir(int fd);
```

The `opendir()` function opens a directory stream corresponding to the directory *name*, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.

The `fdopendir()` function is like `opendir()`, but returns a directory stream for the directory referred to by the open file descriptor *fd*. After a successful call to `fdopendir()`, *fd* is used internally by the implementation, and should not otherwise be used by the application.

# readdir

- readdir - read a directory

```
struct dirent *readdir(DIR *dirp);
```

The readdir() function returns a pointer to a dirent structure representing the next directory entry in the directory stream pointed to by dirp.

On success, readdir() returns a pointer to a dirent structure. (This structure may be statically allocated; do not attempt to free(3) it.)

# closedir

- closedir — close a directory stream

```
int closedir(DIR *dirp);
```

- The closedir() function shall close the directory stream referred to by the argument dirp. Upon return, the value of dirp may no longer point to an accessible object of the type DIR. If a file descriptor is used to implement type DIR, that file descriptor shall be closed.

# Exercise 2 - readdir.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <dirent.h>
4
5 int main (int argc, char **argv) {
6     struct dirent *dirent;
7     DIR *parentDir;
8
9     if (argc < 2) {
10         printf ("Usage: %s <dirname>\n", argv[0]);
11         exit(-1);
12     }
13     parentDir = opendir (argv[1]);
14     if (parentDir == NULL) {
15         printf ("Error opening directory '%s'\n", argv[1]);
16         exit (-1);
17     }
18     int count = 1;
19     while((dirent = readdir(parentDir)) != NULL){
20         printf ("[%d] %s\n", count, (*dirent).d_name);
21         count++;
22     }
23     closedir (parentDir);
24     return 0;
25 }
```

# Exercise 2 - compile & run

```
[base] mahmutunan@MacBook-Pro lecture12 % gcc -o exercise2 readdir.c
[base] mahmutunan@MacBook-Pro lecture12 % ./exercise2
Usage: ./exercise2 <dirname>
[base] mahmutunan@MacBook-Pro lecture12 % ./exercise2 ./
[1] .
[2] ..
[3] someNewFile.txt
[4] .DS_Store
[5] exercise2
[6] readdir.c
[7] exercise1
[8] aSymbolicLink
[9] printstat.c
[10] newFolder
[11] lstat.c
[base] mahmutunan@MacBook-Pro lecture12 % ./exercise2 ./newFolder
[1] .
[2] ..
[base] mahmutunan@MacBook-Pro lecture12 %
```

# Exercise 3 - readdir\_v2.c

```
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <dirent.h>
8
9  char *filetype(unsigned char type) {
10    char *str;
11    switch(type) {
12      case DT_BLK: str = "block device"; break;
13      case DT_CHR: str = "character device"; break;
14      case DT_DIR: str = "directory"; break;
15      case DT_FIFO: str = "named pipe (FIFO)"; break;
16      case DT_LNK: str = "symbolic link"; break;
17      case DT_REG: str = "regular file"; break;
18      case DT SOCK: str = "UNIX domain socket"; break;
19      case DT_UNKNOWN: str = "unknown file type"; break;
20      default: str = "UNKNOWN";
21    }
22    return str;
23 }
```

# Exercise 3 - readdir\_v2.c / 2

```
24
25     int main (int argc, char **argv) {
26         struct dirent *dirent;
27         DIR *parentDir;
28
29         if (argc < 2) {
30             printf ("Usage: %s <dirname>\n", argv[0]);
31             exit(-1);
32         }
33         parentDir = opendir (argv[1]);
34         if (parentDir == NULL) {
35             printf ("Error opening directory '%s'\n", argv[1]);
36             exit (-1);
37         }
38         int count = 1;
39         while((dirent = readdir(parentDir)) != NULL){
40             printf ("[%d] %s (%s)\n", count, dirent->d_name, filetype(dirent->d_type));
41             count++;
42         }
43         closedir (parentDir);
44         return 0;
45     }
```

# Exercise 3 - readdir\_v2.c / compile&run

```
[base] mahmutunan@MacBook-Pro lecture13 % gcc -o exercise3 readdir_v2.c
[base] mahmutunan@MacBook-Pro lecture13 % ./exercise3 ./
[1] . (directory)
[2] .. (directory)
[3] someNewFile.txt (regular file)
[4] writetestat.c (regular file)
[5] .DS_Store (regular file)
[6] exercise2 (regular file)
[7] readstat.c (regular file)
[8] exercise3 (regular file)
[9] readdir.c (regular file)
[10] exercise1 (regular file)
[11] aSymbolicLink (symbolic link)
[12] printstat.c (regular file)
[13] newFolder (directory)
[14] funcptr.c (regular file)
[15] lstat.c (regular file)
[16] readdir_v2.c (regular file)
(base) mahmutunan@MacBook-Pro lecture13 %
```

# Exercise 4

- We can use the read/write system calls from Lab-04 to write the stat structure and read the stat structure.
- Note that data is written as a binary file.

# writestats.c

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

void printstat(struct stat statbuf);

int main(int argc, char** argv) {
    int fd;
    struct stat statbuf;

    if (lstat(argv[1], &statbuf) < 0) {
        printf("Error reading file/directory %s\n", argv[1]);
        perror("lstat");
        exit(-1);
    }
    printstat(statbuf);

    if ((fd = open(argv[2], O_CREAT | O_WRONLY, 0755)) == -1) {
        printf("Error opening file %s\n", argv[2]);
        perror("open");
        exit(-1);
    }
    write(fd, &statbuf, sizeof(struct stat));
    close(fd);

    return 0;
}
```

# readstats.c

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

void printstat(struct stat sb);

int main(int argc, char** argv) {
    int fd;
    struct stat statbuf;

    if ((fd = open(argv[1], O_RDONLY)) == -1) {
        printf("Error opening file %s\n", argv[1]);
        perror("open");
        exit(-1);
    }
    read(fd, &statbuf, sizeof(struct stat));
    close(fd);

    printstat(statbuf);

    return 0;
}
```

# Exercise 4 - compile&run

```
(base) mahmutunan@MacBook-Pro lecture13 % gcc -Wall -o exercise4 writestat.c printstat.c
(base) mahmutunan@MacBook-Pro lecture13 % ./exercise4 lstat.c stat.out
File type: regular file
I-node number: 10642103
Mode: 100644 (octal)
Link count: 1
Ownership: UID=501 GID=20
Preferred I/O block size: 4096 bytes
File size: 1041 bytes
Blocks allocated: 8
Last status change: Mon Sep 21 19:40:17 2020
Last file access: Mon Sep 21 19:40:19 2020
Last file modification: Mon Sep 21 12:23:14 2020
(base) mahmutunan@MacBook-Pro lecture13 % gcc -Wall -o exercise4_read readstat.c printstat.c
(base) mahmutunan@MacBook-Pro lecture13 % ./exercise4_read stat.out
File type: regular file
I-node number: 10642103
Mode: 100644 (octal)
Link count: 1
Ownership: UID=501 GID=20
Preferred I/O block size: 4096 bytes
File size: 1041 bytes
Blocks allocated: 8
Last status change: Mon Sep 21 19:40:17 2020
Last file access: Mon Sep 21 19:40:19 2020
Last file modification: Mon Sep 21 12:23:14 2020
(base) mahmutunan@MacBook-Pro lecture13 %
```

# Hints for the HW2

- You can find a more elaborate example in Figure 4.22 in the textbook.
- This program takes as input a directory name, traverses a file hierarchy, counts the different types of files in the given file hierarchy, and prints the summary (as shown in Figure 4.4).
- This program uses function pointers i.e., you can pass a function as an argument to a function similar to passing variables of different type.
- This enables us to perform different operations on a file as we traverse the file hierarchy.

# Function Pointers - recall

- In the C environment, like normal data pointers, we can have pointers to functions

```
1  void someFunction(int x)
2  {
3      printf("Square of x is %d\n", (x*x));
4  }
5
6  ► int main()
7  {
8      void (*functionPointer)(int) = someFunction;
9      functionPointer(10);
10     return 0;
11 }
```

# Passing Pointers to functions

```
1 #include <stdio.h>
2 ← void test(int a);
3 ► □ int main(void)
4 {
5     void (*ptr)(int a);
6     ptr = test;
7     (*ptr)(10);
8     return 0;
9 }
10 ← □ void test(int a)
11 {
12     printf("%d\n", 2*a);
13 }
```



20

```
1 #include <stdio.h>
2 ↵ int addTwoNumbers(int a, int b);
3 ↵ int subtractTwoNumbers(int a, int b);
4
5 ► int main(void)
6 {
7     int (*ptr[2])(int a, int b);
8     int i, j, result;
9     ptr[0] = addTwoNumbers;
10    ptr[1] = subtractTwoNumbers;
11
12    printf("Enter two integer numbers: ");
13    scanf("%d %d", &i, &j);
14    if(i > 0 && i < 25)
15        result = ptr[0](i, j);
16    else
17        result = ptr[1](i, j);
18    printf("Result : %d\n", result);
19    return 0;
20}
21 ↵ int addTwoNumbers(int a, int b)
22 ↵ {    return a+b;    }
23 ↵ int subtractTwoNumbers(int a, int b)
24 ↵ {    return a-b;    }
```



```
1 #include <stdio.h>
2 ↵ int addTwoNumbers(int a, int b);
3 ↵ int subtractTwoNumbers(int a, int b);
4
5 ► ↵ int main(void)
6 {
7     int (*ptr[2])(int a, int b);
8     int i, j, result;
9     ptr[0] = addTwoNumbers;
10    ptr[1] = subtractTwoNumbers;
11
12    printf("Enter two integer numbers: ");
13    scanf("%d %d", &i, &j);
14    if(i > 0 && i < 25)
15        result = ptr[0](i, j);
16    else
17        result = ptr[1](i, j);
18    printf("Result : %d", result);
19    return 0;
20 }
21 ↵ int addTwoNumbers(int a, int b)
22 { return a+b; }
23 ↵ int subtractTwoNumbers(int a, int b)
24 { return a-b; }
```



Enter two integer numbers: 20 30

Result : 50

Enter two integer numbers: 45 20

Result : 25

# Exercise 5

```
1  /* Sample program to illustrate how to use function pointers */
2  #include <stdio.h>
3  typedef int MYFUNC(int a, int b);
4
5  int add(int a, int b) {
6      printf("This is the add function\n");
7      return a + b;
8 }
9
10 int sub(int a, int b) {
11     printf("This is the subtraction function\n");
12     return a - b;
13 }
14
15 int opfunc(int a, int b, MYFUNC *f) {
16     return f(a, b);
17 }
18
19 int main(int argc, char *argv[]) {
20     int a = 10, b = 5;
21     printf("Passing add function....\n");
22     printf("Result = %d\n", opfunc(a, b, add));
23     printf("Passing sub function....\n");
24     printf("Result = %d\n", opfunc(a, b, sub));
25     return 0;
26 }
```

- In this example we define a function *opfunc* that takes as input a pointer to a function that takes two integer arguments and returns an integer value.
- We use *typedef* to define the function signature so that we can use this as a type in the function definition.
- Then we can have different functions with the given type signature and these functions can perform different operations. In this example, we define two functions to perform addition and subtraction on the two arguments passed to the function.
- Now we can invoke the *opfunc* by providing two integer values and the corresponding function to perform the required operation.

- Example 4.22 uses this mechanism through which we can define different functions to perform different operations on a given file as we traverse the file hierarchy.
- In this example, we just count the different files types, however, we could perform other operations such as check the file size or file permission or any other user-defined operation.

# Standard I/O Library

- This library is specified by ISO C standard because it has been implemented on many OS
- The standard I/O library handles such details as buffer allocation and performing I/O in optimal-sized chunks, obviating our need to worry about using the correct block size

# Recall - File Descriptor

- When a file opened
  - nonnegative int assigned
  - this int is used in all operations
- Streams
  - with the standard I/O library, the discussion centers on streams
  - Open or Create a file → associate with a stream

# I/O Stream

- *Open I/O Stream:* The standard I/O stream allows you to open a file in read, write, or append modes. This mode can be combined in a single open function call (see Figure 5.2 in Section 5.5 of the textbook for a complete list of options that can be specified). For example:

```
FILE *fptr;  
fptr = fopen ("listings.csv", "r+");
```

<i>type</i>	Description	<i>open(2)</i> Flags
r or rb	open for reading	O_RDONLY
w or wb	truncate to 0 length or create for writing	O_WRONLY   O_CREAT   O_TRUNC
a or ab	append; open for writing at end of file, or create for writing	O_WRONLY   O_CREAT   O_APPEND
r+ or r+b or rb+	open for reading and writing	O_RDWR
w+ or w+b or wb+	truncate to 0 length or create for reading and writing	O_RDWR   O_CREAT   O_TRUNC
a+ or a+b or ab+	open or create for reading and writing at end of file	O_RDWR   O_CREAT   O_APPEND

**Figure 5.2** The *type* argument for opening a standard I/O stream

# Input / Output Stream:

- ***Input Stream:*** The standard I/O stream allows us to read from the open file. These functions allow us to read a file character by character – getchar(), line by line – fgets(), or with specific size – fread()
- ***Output Stream:*** The standard I/O stream allow you to write to an open file. These functions allow you to write to a file character by character – putchar(), line by line – fputs(), or with specific size – fwrite()

# Exercise 1

- Now let's use these functions and write a program. We will use APIs available in Linux and C to develop different versions of this program

# getc()

```
int getc(FILE *stream)
```

- Gets the next character (an unsigned char) from the specified stream
- Advances the position indicator for the stream.

# getline1.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int getLine(FILE *fp, char *line);
5
6  int main(int argc, char** argv) {
7      char *str;
8      FILE *fp;
9      int n;
10
11     str = malloc(sizeof(char)*BUFSIZ);
12     fp = fopen( filename: argv[1], mode: "r");
13     if (fp == NULL) {
14         printf("Error opening file %s\n", argv[1]);
15         exit(-1);
16     }
17     while ( (n = getLine(fp, str)) > 0)
18         printf("%d: %s\n", n, str);
19     fclose(fp);
20     return 0;
21 }
22
23 int getLine(FILE *fp, char *line) {
24     int c, i=0;
25     while ((c = getc(fp)) != '\n' && c != EOF)
26         line[i++] = c;
27     line[i] = '\0';
28     return i;
29 }
```



Some text line 1  
Line 2  
l3

```
(base) mahmutunan@MacBook-Pro lecture15 % gcc -o getline1 getline1.c
(base) mahmutunan@MacBook-Pro lecture15 % ./getline1 test.txt
16: Some text line 1
6: Line 2
2: l3
```

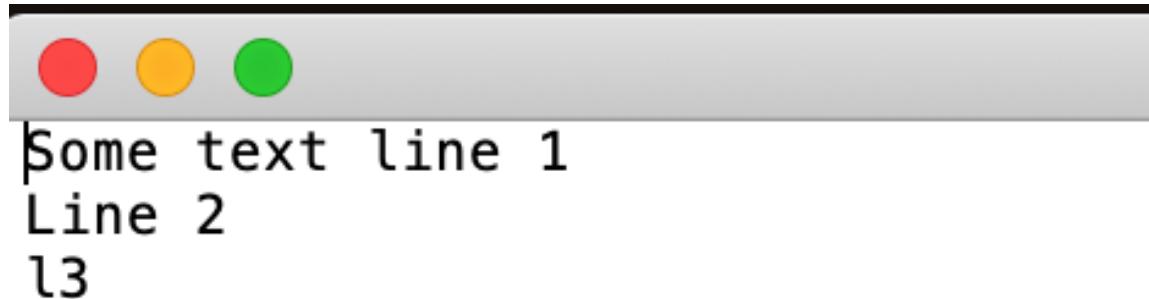
# getline()

```
ssize_t getline(char **lineptr, size_t *n, FILE  
*stream);
```

- **getline()** reads an entire line from *stream*, storing the address of the buffer containing the text into *\*lineptr*. The buffer is null-terminated and includes the newline character, if one was found.
- On success, **getline()** returns the number of characters read, including the delimiter character, but not including the terminating null byte ('\0')

# getline2.c

```
1
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main(int argc, char** argv) {
6     char *line=NULL;
7     FILE *fp;
8     size_t maxlen=0;
9     ssize_t n;
10
11    printf("BUFSIZ = %d\n", BUFSIZ);
12
13    if ((fp = fopen( filename: argv[1], mode: "r")) == NULL) {
14        printf("Error opening file %s\n", argv[1]);
15        exit(-1);
16    }
17    while ( (n = getline(&line, &maxlen, fp)) > 0)
18        printf("%ld[%ld]: %s\n", n, maxlen, line);
19
20    fclose(fp);
21    return 0;
22}
```



```
(base) mahmutunan@MacBook-Pro lecture15 % ./getline2 test.txt
BUFSIZ = 1024
17[32]: Some text line 1

7[32]: Line 2

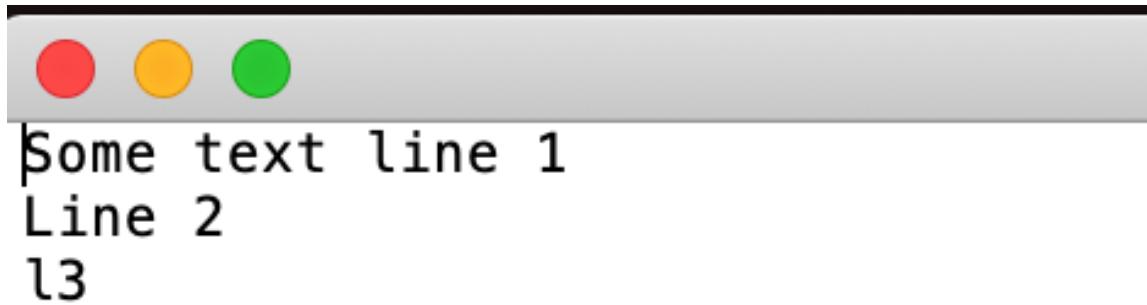
2[32]: l3
```

# getdelim()

- `ssize_t getdelim(char **lineptr,  
size_t *n, int delim, FILE *stream);`
- `getdelim()` works like `getline()`, except that a line delimiter other than newline can be specified as the *delimiter* argument.
- `getdelim()` returns the number of characters read, including the delimiter character, but not including the terminating null byte

# getline3.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char** argv) {
5     char *line=NULL;
6     FILE *fp;
7     size_t maxlen=0;
8     ssize_t n;
9
10    if ((fp = fopen(filename: argv[1], mode: "r")) == NULL) {
11        printf("Error opening file %s\n", argv[1]);
12        exit(-1);
13    }
14    while ( (n = getdelim(&line, &maxlen, delimiter: ' ', fp)) > 0)
15        printf("%ld: %s\n", n, line);
16
17    fclose(fp);
18    return 0;
19}
20
```



Some text line 1  
Line 2  
l3

```
(base) mahmutunan@MacBook-Pro lecture15 % gcc -o getline3 getline3.c
(base) mahmutunan@MacBook-Pro lecture15 % ./getline3 test.txt
5: Some
5: text
5: line
7: 1
Line
4: 2
l3
(base) mahmutunan@MacBook-Pro lecture15 %
```

# gets()

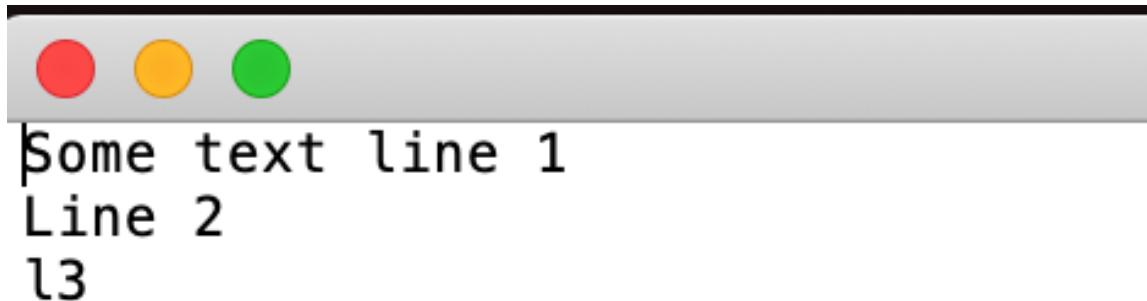
- `char *gets(char *str)`
- The C library function `char *gets(char *str)` reads a line from `stdin` and stores it into the string pointed to by `str`.
- It stops when either the newline character is read or when the end-of-file is reached, whichever comes first.
- It reads string from standard input and prints the entered string, but it suffers from Buffer Overflow as `gets()` doesn't do any array bound testing.
- `gets()` keeps on reading until it sees a newline character.
- To avoid Buffer Overflow, `fgets()` should be used instead of `gets()` as `fgets()` makes sure that not more than `MAX_LIMIT` characters are read.

# fgets()

- `char *fgets(char *str, int n, FILE *stream)`
- **str** – This is the pointer to an array of chars where the string read is stored.
- **n** – This is the maximum number of characters to be read (including the final null-character). Usually, the length of the array passed as str is used.
- **stream** – This is the pointer to a FILE object that identifies the stream where characters are read from.
- On success, the function returns the same str parameter. If the End-of-File is encountered and no characters have been read, the contents of str remain unchanged and a null pointer is returned

# getline4.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 int main(int argc, char** argv) {
6     char *line;
7     FILE *fp;
8
9     line = malloc(sizeof(char)*BUFSIZ);
10
11    if ((fp = fopen( filename: argv[1], mode: "r")) == NULL) {
12        fprintf(stderr,"Error opening file %s\n", argv[1]);
13        exit(-1);
14    }
15    while ( fgets(line, BUFSIZ, fp) != NULL )
16        fprintf(stdout,"%ld: %s\n", strlen(line), line);
17
18    fclose(fp);
19    return 0;
20 }
```



```
[base] mahmutunan@MacBook-Pro lecture15 % gcc -o getline4 getline4.c
[base] mahmutunan@MacBook-Pro lecture15 % ./getline4 test.txt
17: Some text line 1
7: Line 2
2: l3
[base] mahmutunan@MacBook-Pro lecture15 %
```

# fscanf()

- `int fscanf(FILE *stream, const char *format, ...)`

Conversion type	Description
d, i	signed decimal
o	unsigned octal
u	unsigned decimal
x, X	unsigned hexadecimal
f, F	double floating-point number
e, E	double floating-point number in exponential format
g, G	interpreted as f, F, e, or E, depending on value converted
a, A	double floating-point number in hexadecimal exponential format
c	character (with 1 length modifier, wide character)
s	string (with 1 length modifier, wide character string)
p	pointer to a void
n	pointer to a signed integer into which is written the number of characters written so far
%	a % character
C	wide character (XSI option, equivalent to 1c)
S	wide character string (XSI option, equivalent to 1s)

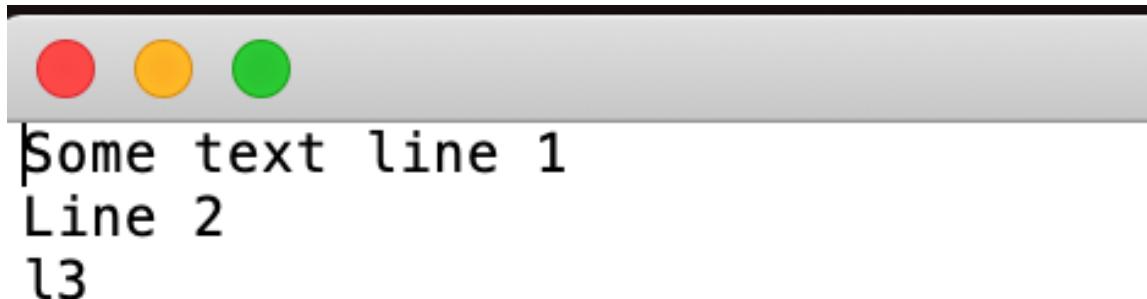
**Figure 5.9** The conversion type component of a conversion specification

Conversion type	Description
d	signed decimal, base 10
i	signed decimal, base determined by format of input
o	unsigned octal (input optionally signed)
u	unsigned decimal, base 10 (input optionally signed)
x,X	unsigned hexadecimal (input optionally signed)
a,A,e,E,f,F,g,G	floating-point number
c	character (with 1 length modifier, wide character)
s	string (with 1 length modifier, wide character string)
[	matches a sequence of listed characters, ending with ]
[^	matches all characters except the ones listed, ending with ]
p	pointer to a void
n	pointer to a signed integer into which is written the number of characters read so far
%	a % character
C	wide character (XSI option, equivalent to 1c)
S	wide character string (XSI option, equivalent to 1s)

**Figure 5.10** The conversion type component of a conversion specification

# getline5.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(int argc, char** argv) {
6      char *line;
7      FILE *fp;
8
9      line = malloc(sizeof(char)*BUFSIZ);
10
11     if ((fp = fopen( filename: argv[1], mode: "r")) == NULL) {
12         printf("Error opening file %s\n", argv[1]);
13         exit(-1);
14     }
15     while ( fscanf(fp, "%s", line) != EOF )
16         printf("%ld: %s\n", strlen(line), line);
17
18     fclose(fp);
19     return 0;
20 }
```



```
[base] mahmutunan@MacBook-Pro lecture15 % gcc -o getline5 getline5.c  
[base] mahmutunan@MacBook-Pro lecture15 % ./getline5 test.txt  
4: Some  
4: text  
4: line  
1: 1  
4: Line  
1: 2  
2: l3
```

# fprintf()

```
int fprintf(FILE *stream, const char *format, ...)
```

## stream

The stream where the output will be written.

## format

Describes the output as well as provides a placeholder to insert the formatted string. Here are a few examples:

Format	Explanation	Example
%d	Display an integer	10
%f	Displays a floating-point number in fixed decimal format	10.500000
.1f	Displays a floating-point number with 1 digit after the decimal	10.5
%e	Display a floating-point number in exponential (scientific notation)	1.050000e+01
%g	Display a floating-point number in either fixed decimal or exponential format depending on the size of the number (will not display trailing zeros)	10.5

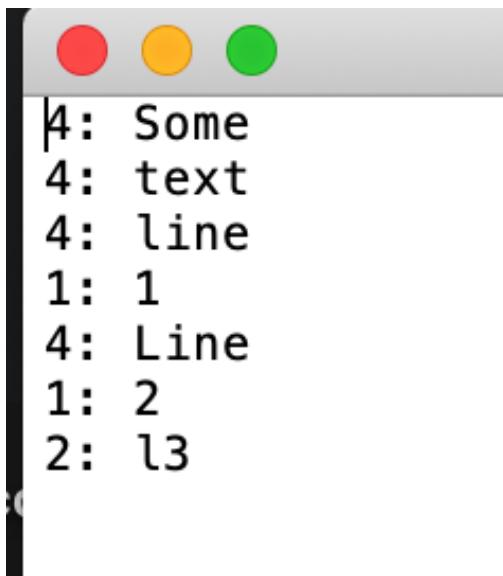
# getline6.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 int main(int argc, char** argv) {
6     char *line;
7     FILE *fp, *fpout;
8
9     line = malloc(sizeof(char)*BUFSIZ);
10
11    if ((fp = fopen(filename: argv[1], mode: "r")) == NULL) {
12        fprintf(stderr,"Error opening file %s\n", argv[1]);
13        exit(-1);
14    }
15    if ((fpout = fopen(filename: argv[2], mode: "w")) == NULL) {
16        fprintf(stderr,"Error opening file %s\n", argv[2]);
17        exit(-1);
18    }
19    while ( fscanf(fp, "%s", line) != EOF )
20        fprintf(fpout,"%ld: %s\n", strlen(line), line);
21
22    fclose(fp);
23    fclose(fpout);
24    return 0;
25 }
```



```
Some text line 1  
Line 2  
l3
```

```
(base) mahmutunan@MacBook-Pro lecture15 % gcc -o getline6 getline6.c  
(base) mahmutunan@MacBook-Pro lecture15 % ./getline6 test.txt output.txt  
(base) mahmutunan@MacBook-Pro lecture15 % _
```



# Example 2

- We will now write a program to read a comma separated file (“listing.csv”) and use the C structures to store and display the data on the console.
- The sample input file used is :

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	id	host_id	host_name	neighbourhood	neighbourhood_group	latitude	longitude	room_type	price	minimum_nights	number_of_reviews	calculated_host_listings_count	availability_365
2	2015	2217	Ian	Mitte	Brunnenstr. 9	52.5345373	13.4025569	Entire home/apt	60	4	118	4	141
3	2695	2986	Michael	Pankow	Prenzlauer Berg	52.5485128	13.4045528	Private room	17	2	6	1	0
4	3176	3718	Britta	Pankow	Prenzlauer Berg	52.5349962	13.4175787	Entire home/apt	90	62	143	1	220
5	3309	4108	Jana	Tempelhof - Schöneberg	Schöneberg	52.4988549	13.3490645	Private room	26	5	25	1	297
6	7071	17391	Bright	Pankow	Helmholtzplatz	52.5431573	13.4150911	Private room	42	2	197	1	26
7	9991	33852	Philipp	Pankow	Prenzlauer Berg	52.5330308	13.4160468	Entire home/apt	180	6	6	1	137
8	14325	55531	Chris + Olive	Pankow	Prenzlauer Berg	52.5478464	13.4055622	Entire home/apt	70	90	23	3	129
9	16401	59666	Melanie	Friedrichshain-Kreuzberg	Frankfurter Allee	52.510514	13.4578502	Private room	120	30	0	1	365
10	16644	64696	Rene	Friedrichshain-Kreuzberg	Neukölln	52.5047923	13.4351019	Entire home/apt	90	60	48	2	159
11	17409	67590	Wolfram	Pankow	Prenzlauer Berg	52.5290709	13.4128434	Private room	45	3	279	1	42
12	17904	68997	Matthias	Neukölln	Reuterstraße	52.4954763	13.4218213	Entire home/apt	49	5	223	1	232
13	20858	71331	Marc	Pankow	Prenzlauer Berg	52.5369524	13.407615	Entire home/apt	129	3	56	1	166
14	21869	64696	Rene	Friedrichshain-Kreuzberg	Neukölln	52.5027333	13.4346199	Entire home/apt	70	60	60	2	129
15	22415	86068	Kiki	Friedrichshain-Kreuzberg	Neukölln	52.4948506	13.4285006	Entire home/apt	98	3	61	2	257
16	22677	87357	Ramfis	Mitte	Brunnenstraße	52.5343484	13.4055765	Entire home/apt	160	3	223	1	228
17	23834	94918	Tanja	Friedrichshain-Kreuzberg	Tempelhofer Vorstadt	52.4897144	13.3797476	Entire home/apt	65	60	96	1	275
18	24569	99662	Dominik	Pankow	Prenzlauer Berg	52.5307909	13.4180844	Entire home/apt	90	3	18	2	3
19	25653	99662	Dominik	Pankow	Prenzlauer Berg	52.5302587	13.419467	Entire home/apt	90	4	5	2	15
20	26543	112675	Terri	Pankow	Helmholtzplatz	52.5440624	13.4213765	Entire home/apt	197	3	163	1	336
21	28156	55531	Chris + Olive	Pankow	Prenzlauer Berg	52.5467194	13.405117	Entire home/apt	70	90	28	3	191
22	28268	121580	Elena	Friedrichshain-Kreuzberg	Neukölln	52.5133852	13.4699475	Entire home/apt	90	5	30	1	55
23	28711	84157	Emanuela	Neukölln	Reuterstraße	52.4861061	13.434817	Entire home/apt	60	2	1	10	341
24	29279	54283	Marine	Pankow	Helmholtzplatz	52.5417876	13.4238832	Entire home/apt	130	60	69	3	221

# listing.c

- The given file has 13 different attributes, and these attributes can be divided into three different datatypes: integer, character array, and float.
- And collectively we can create a C structure to represent these attributes and then create an array of such structures to store multiple entities.
- 1. Define a structure called listing with all attributes as individual members of the struct listing.

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

#define LINESIZE 1024

struct listing {
    int id, host_id, minimum_nights, number_of_reviews, calculated_host_listings_count, availability_365;
    char *host_name, *neighbourhood_group, *neighbourhood, *room_type;
    float latitude, longitude, price;
};
```

# *strtok*

- `char *strtok(char *str, const char *delim);`
- `char *strtok_r(char *str, const char *delim, char **saveptr);`
- strtok, strtok\_r - extract tokens from strings
- The `strtok()` function breaks a string into a sequence of zero or more nonempty tokens. On the first call to `strtok()`, the string to be parsed should be specified in `str`. In each subsequent call that should parse the same string, `str` must be `NULL`.

# atoi

- atoi, atol, atoll - convert a string to an integer
- int atoi(const char \**nptr*) ;
- long atol(const char \**nptr*) ;
- long long atoll(const char \**nptr*) ;
- Return value: The converted value or 0 on error.

<https://man7.org/linux/man-pages/man3/atoi.3.html>

# atof

- **atof** - convert a string to a double

```
double atof(const char *nptr);
```

**RETURN VALUE** : The converted value

<https://man7.org/linux/man-pages/man3/atof.3.html>

# strdup

- strdup- duplicate a string

```
char *strdup(const char *s);
```

On success, the strdup() function returns a pointer to the duplicated string. It returns NULL if insufficient memory was available, with errno set to indicate the cause of the error.

<https://www.man7.org/linux/man-pages/man3/strndup.3.html#:~:text=DESCRIPTION%20top,copies%20at%20most%20n%20bytes.>

# listing.c

- 2. Define a function which can help to parse each line in the file and return the above defined structure.
- For this task you need to learn the string tokenizer function (*strtok*) which is available in *<string.h>* header file.
- You can find out more about the *strtok* function by typing *man strtok*.
- You will notice that when you invoke the *strtok* function for the first time you provide the pointer to the character array and on subsequent invocations of *strtok* we use *NULL* as the argument.

# listing.c

```
15  struct listing getfields(char* line){
16      struct listing item;
17
18      item.id = atoi(strtok(line, ","));
19      item.host_id = atoi(strtok(NULL, ","));
20      item.host_name = strdup(strtok(NULL, ","));
21      item.neighbourhood_group = strdup(strtok(NULL, ","));
22      item.neighbourhood = strdup(strtok(NULL, ","));
23      item.latitude = atof(strtok(NULL, ","));
24      item.longitude = atof(strtok(NULL, ","));
25      item.room_type = strdup(strtok(NULL, ","));
26      item.price = atof(strtok(NULL, ","));
27      item.minimum_nights = atoi(strtok(NULL, ","));
28      item.number_of_reviews = atoi(strtok(NULL, ","));
29      item.calculated_host_listings_count = atoi(strtok(NULL, ","));
30      item.availability_365 = atoi(strtok(NULL, ","));
31
32      return item;
33 }
```

# listing.c

Now, create a function to display the items in the struct

```
36 void displayStruct(struct listing item) {  
37     printf("ID : %d\n", item.id);  
38     printf("Host ID : %d\n", item.host_id);  
39     printf("Host Name : %s\n", item.host_name);  
40     printf("Neighbourhood Group : %s\n", item.neighbourhood_group);  
41     printf("Neighbourhood : %s\n", item.neighbourhood);  
42     printf("Latitude : %f\n", item.latitude);  
43     printf("Longitude : %f\n", item.longitude);  
44     printf("Room Type : %s\n", item.room_type);  
45     printf("Price : %f\n", item.price);  
46     printf("Minimum Nights : %d\n", item.minimum_nights);  
47     printf("Number of Reviews : %d\n", item.number_of_reviews);  
48     printf("Calculated Host Listings Count : %d\n", item.calculated_host_listings_count);  
49     printf("Availability_365 : %d\n", item.availability_365);  
50 }
```

# listing.c

- 3. Now use *fopen* function to open file in read only mode. Notice that the *fopen* function returns a pointer of *FILE* type (file pointer) unlike the *open* function that returns an *integer* value as the file descriptor.

```
52     int main(int argc, char* args[]) {
53         struct listing list_items[22555];
54         char line[LINESIZE];
55         int i, count;
56
57         FILE *fptr = fopen("listings.csv", "r");
58         if(fptr == NULL){
59             printf("Error reading input file listings.csv\n");
60             exit (-1);
61     }
```

# listing.c

- 4. Then loop through till the end of file (use fgets function) and store all data in the array of structures
- We can close the file using fclose function

```
62
63     count = 0;
64     while (fgets(line, LINESIZE, fptr) != NULL){
65         list_items[count++] = getfields(line);
66     }
67     fclose(fptr);
```

# listing.c

- 5. Now invoke the function to display the structure in a loop.

```
69     for (i=0; i<count; i++)
70         displayStruct(list_items[i]);
71
72     return 0;
73 }
74
```

# Process Management

# Process

- Fundamental to the structure of operating systems

A *process* can be defined as:

A program in execution

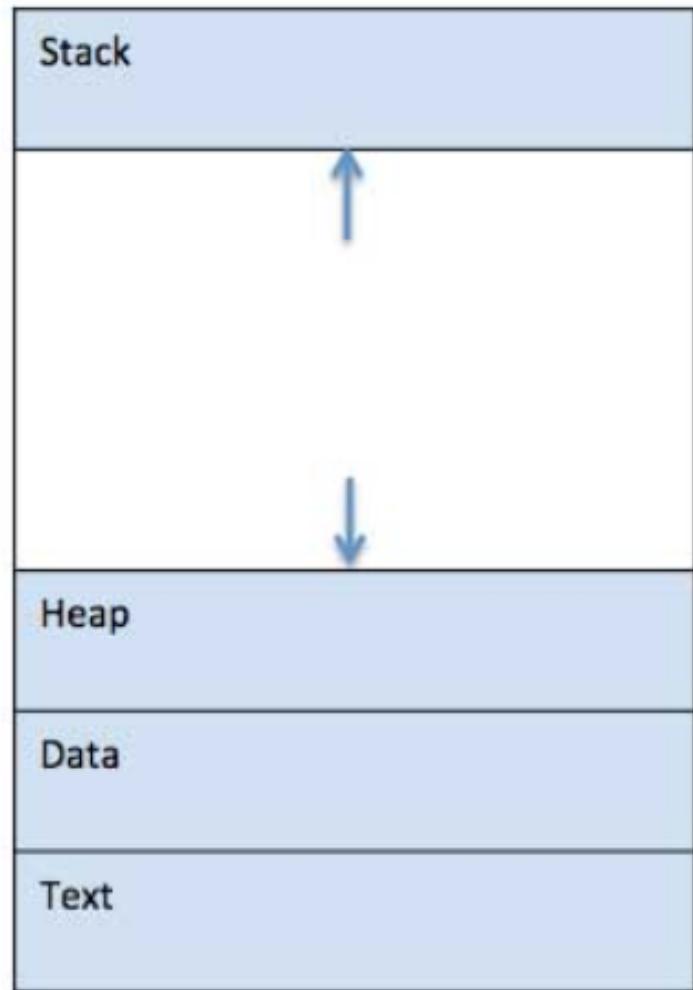
An instance of a running program

The entity that can be assigned to, and executed on, a processor

A unit of activity characterized by a single sequential thread of execution, a current state, and an associated set of system resources

# Process

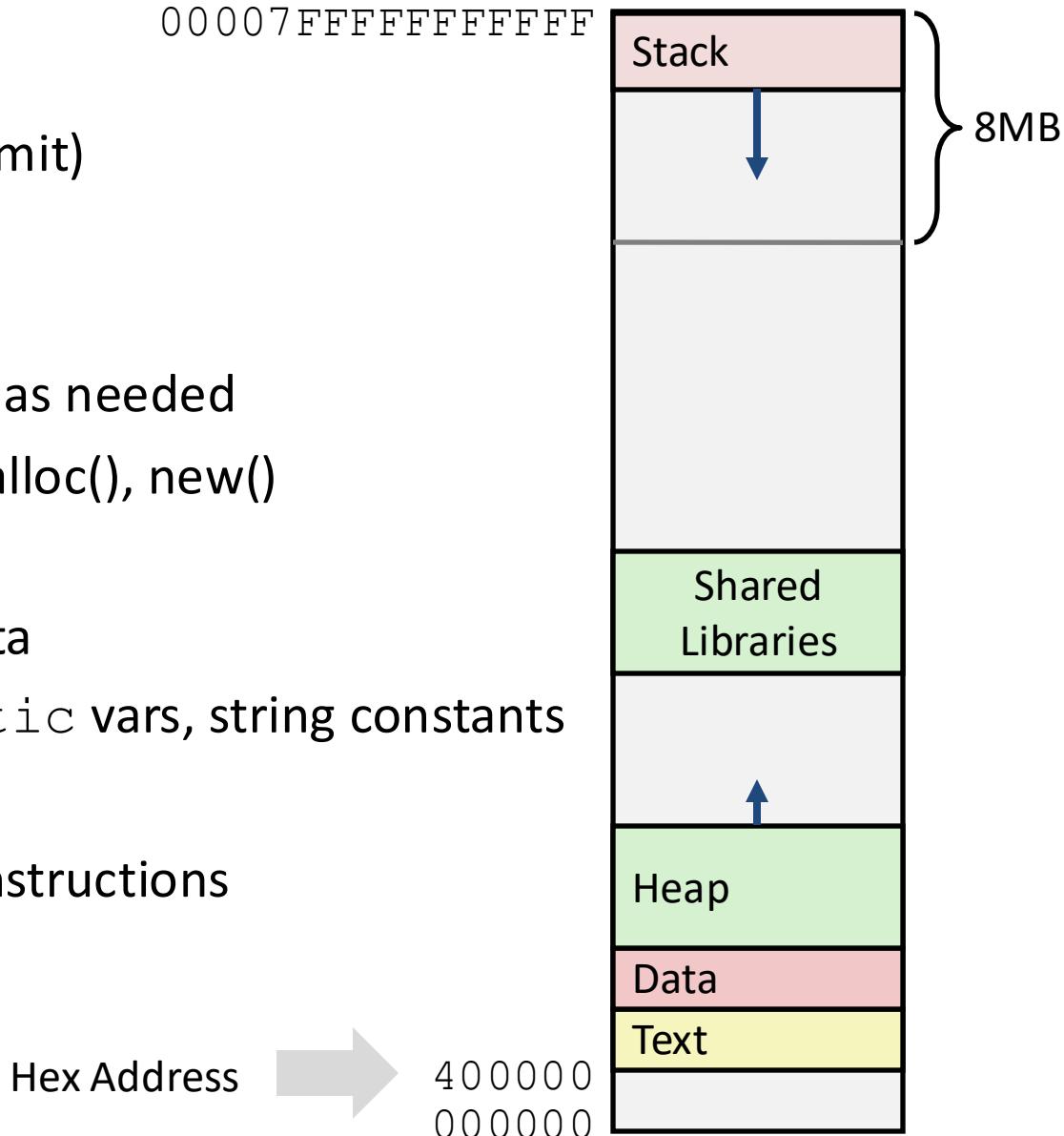
- When a program is loaded into the memory and it becomes a process, it can be divided into four sections
  - stack
  - heap
  - text
  - data.



# x86-64 Linux Memory Layout

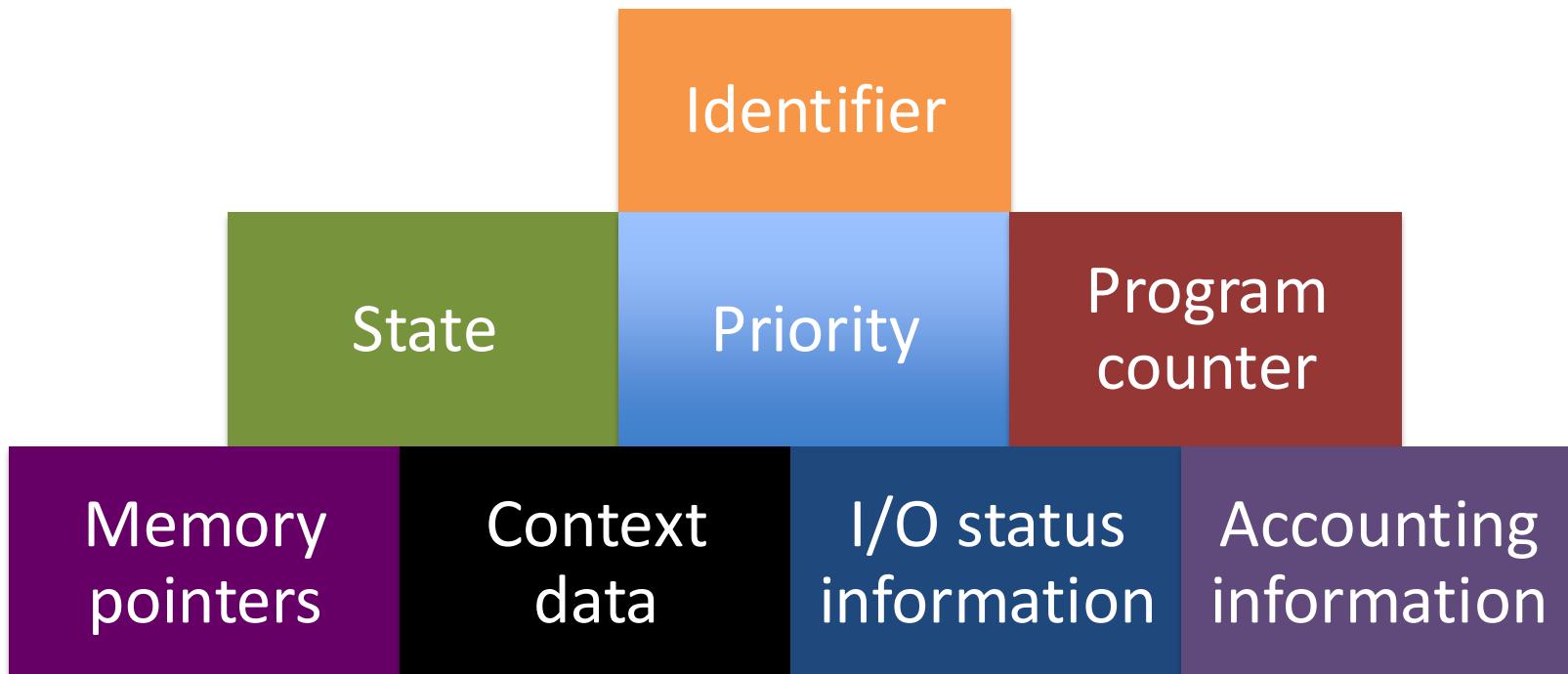
*not drawn to scale*

- Stack
  - Runtime stack (8MB limit)
  - E. g., local variables
- Heap
  - Dynamically allocated as needed
  - When call `malloc()`, `calloc()`, `new()`
- Data
  - Statically allocated data
  - E.g., global vars, static vars, string constants
- Text / Shared Libraries
  - Executable machine instructions
  - Read-only

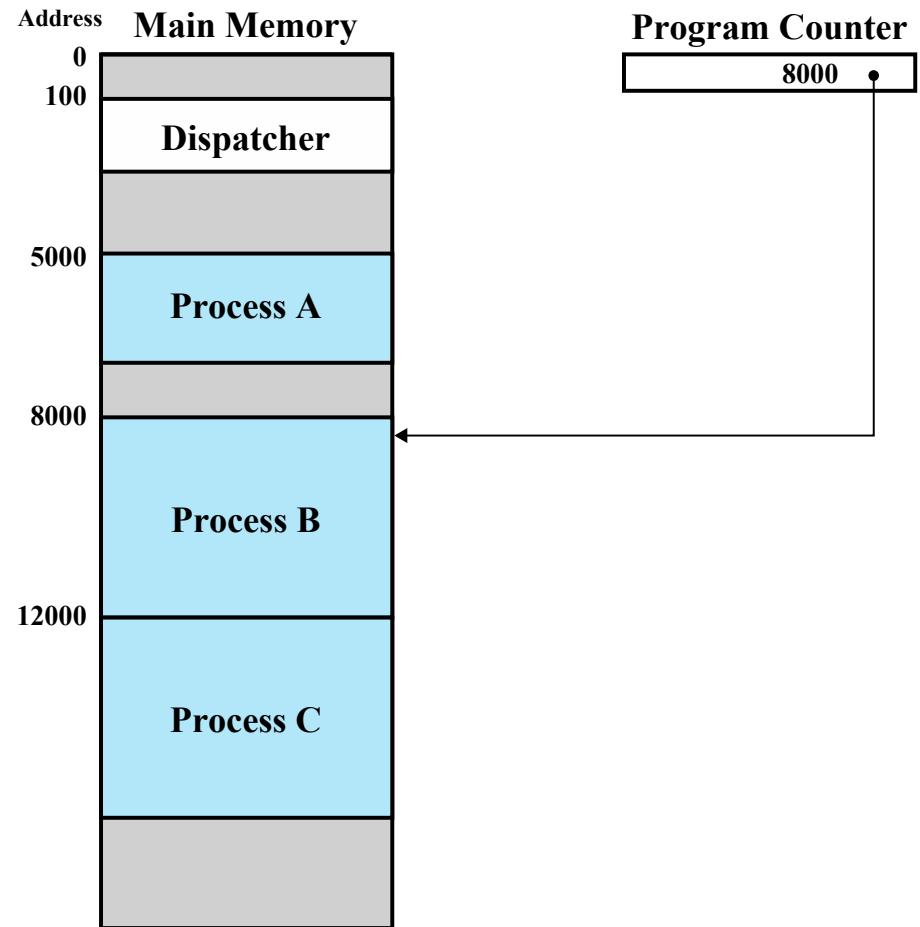


# Process Elements

- While the program is executing, this process can be uniquely characterized by a number of elements, including:



# Process Execution



**Figure 3.2 Snapshot of Example Execution (Figure 3.4) at Instruction Cycle 13**

5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

**(a) Trace of Process A**

**(b) Trace of Process B**

**(c) Trace of Process C**

5000 = Starting address of program of Process A

8000 = Starting address of program of Process B

12000 = Starting address of program of Process C

**Figure 3.3 Traces of Processes of Figure 3.2**

1	5000		
2	5001		
3	5002		
4	5003		
5	5004		
6	5005		
		----- Timeout	
7	100		
8	101		
9	102		
10	103		
11	104		
12	105		
13	8000		
14	8001		
15	8002		
16	8003		
		----- I/O Request	
17	100		
18	101		
19	102		
20	103		
21	104		
22	105		
23	12000		
24	12001		
25	12002		
26	12003		
		----- Timeout	
27	12004		
28	12005		
		----- Timeout	
29	100		
30	101		
31	102		
32	103		
33	104		
34	105		
35	5006		
36	5007		
37	5008		
38	5009		
39	5010		
40	5011		
		----- Timeout	
41	100		
42	101		
43	102		
44	103		
45	104		
46	105		
47	12006		
48	12007		
49	12008		
50	12009		
51	12010		
52	12011		
		----- Timeout	

100 = Starting address of dispatcher program

Shaded areas indicate execution of dispatcher process;  
first and third columns count instruction cycles;  
second and fourth columns show address of instruction being executed

**Figure 3.4 Combined Trace of Processes of Figure 3.2**

# Process Creation

*Process  
spawning*

- When the OS creates a process at the explicit request of another process

*Parent  
process*

- Is the original, creating, process

*Child process*

- Is the new process

# Process Termination

- There must be a means for a process to indicate its completion
- A batch job should include a HALT instruction or an explicit OS service call for termination
- For an interactive application, the action of the user will indicate when the process is completed (e.g. log off, quitting an application)

# Modes of Execution

## User Mode

- Less-privileged mode
- User programs typically execute in this mode

## System Mode

- More-privileged mode
- Also referred to as control mode or kernel mode
- Kernel of the operating system

# Unix Processes

- The OS tracks processes through a five-digit ID number
  - **pid** or **process ID.**
- Each process in the system has a unique **pid**.
- If you want to list the running processes, use the **ps** (process status) command
  - to display the full option

`ps -f`

UID, PID, PPID, C, STIME, TTY, CMD, TIME.....

# Unix Processes

- to stop a process
  - kill
  - kill PID
  - kill -9 PID
- Init Process
  - PID 1 and PPID 0
- Foreground Process
- Background Process

# fork()

- fork - create a child process

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

fork() creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process.

# fork()

- We can use the fork() system call to create a new process (referred to as the child process) which is an identical image of the calling process (referred to as the parent process).
- Here is the C interface for the fork() system call:

```
#include <unistd.h>  
  
pid_t fork(void);
```

# fork()

- If the fork() call is successful, then it returns the process ID of the child process to the parent process and returns 0 in the child process.
- fork() returns a negative value in the parent process and sets the corresponding errno variable (external variable defined in *errno.h*) if there is any error in process creation and the child process is not created.
- We can use perror() function (defined in *stdio.h*) to print the corresponding system error message. Look at the man page for perror to find out more about the perror() function.

# fork()

- Once the parent process creates the child process, the parent process continues with its normal execution.
- If the parent process exits before the child process completes its execution and terminates, the child process will become a zombie process (*i.e.*, a process without a parent process).
- Alternatively, the parent process could wait for the child process to terminate using the `wait()` function.
- The `wait()` system call will suspend the execution of the calling process until one of the child process terminates and if there are no child processes available the `wait()` function returns immediately.

# Process Control

- Process creation is by means of the kernel system call, *fork ()*
- When a process issues a fork request, the OS performs the following functions:
  - 1 • Allocates a slot in the process table for the new process
  - 2 • Assigns a unique process ID to the child process
  - 3 • Makes a copy of the process image of the parent, with the exception of any shared memory
  - 4 • Increments counters for any files owned by the parent, to reflect that an additional process now also owns those files
  - 5 • Assigns the child process to the Ready to Run state
  - 6 • Returns the ID number of the child to the parent process, and a 0 value to the child process

# Hello World!

```
int main() {  
    printf("Hello, World!\n");  
    return 0;  
}
```

Hello, World!

# Hello World!

```
int main() {  
    printf("Hello, World!\n");  
    return 0;  
}
```

Hello, World!

```
int main() {  
    fork();  
    printf("Hello, World!\n");  
    return 0;  
}
```

Hello, World!  
Hello, World!

# Hello World!

```
int main() {  
    int processId = fork();  
    printf("Hello, World! from= %d\n",processId);  
    return 0;  
}
```

```
Hello, World! from= 96037  
Hello, World! from= 0
```

```
int main() {  
    printf("This is before the fork statement\n");  
    fork();  
    printf("After the FIRST fork\n");  
    fork();  
    printf("After the SECOND fork \n");  
    fork();  
    printf("After the THIRD fork \n");  
    return 0;  
}
```

```
After the FIRST fork  
After the SECOND fork  
After the FIRST fork  
After the THIRD fork  
After the SECOND fork  
After the SECOND fork  
After the SECOND fork  
After the THIRD fork
```

```
int main() {  
  
    fork();  
    fork();  
    fork();  
    fork();  
    printf("4 forks will work 16 times\n");  
    return 0;  
}
```

```
4 forks will work 16 times  
4 forks will work 16 times
```

# getpid()

```
int main() {  
  
    printf("before calling the fork %d\n",getpid());  
    fork();  
    printf("after calling the FIRST fork %d\n",getpid());  
    fork();  
    printf("after calling the SECOND fork %d\n",getpid());  
}
```

```
before calling the fork 96717  
after calling the FIRST fork 96717  
after calling the SECOND fork 96717  
after calling the FIRST fork 96718|  
after calling the SECOND fork 96719  
after calling the SECOND fork 96718  
after calling the SECOND fork 96720
```

# `wait()`

`wait`, `waitpid`, `waitid` - wait for process to change state

```
pid_t wait(int *wstatus);  
pid_t waitpid(pid_t pid, int *wstatus, int  
options);  
int waitid(idtype_t idtype, id_t id, siginfo_t  
*infop, int options);
```

<https://www.man7.org/linux/man-pages/man2/waitid.2.html>

# wait()

- The wait() call returns the PID of the child process that terminated when successful, otherwise, it returns -1.
- The wait() call also sets an integer value that is passed as an argument to the function which can be inspected with various macros provided in <sys/wait.h> to determine how the child process completed (e.g., terminated normally, terminated by a signal).

# `wait()` `waitpid()`

- If the calling process created more than one child process, we can use the `waitpid()` system call to wait on a specific child process to change state.
- A state change could be any one of the following events: the child was terminated; the child was stopped by a signal; or the child was resumed by a signal. Similar to `wait()`, `waitpid()` returns the PID of the child process that changed state when successful, otherwise, it returns -1.

# wait() waitpid()

- Here are the C APIs for the `wait()` and `waitpid()` system calls:

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int
options);
```

```
int main() {  
    int processId = fork();  
    int count;  
    fflush(stdout);  
  
    if (processId==0){  
        count=1;  
    }else{  
        count=6;  
    }  
    if (processId!=0){  
        wait();  
    }  
    int i;  
    for (i=count;i<count+5;i++){  
        printf("%d",i);  
        fflush( stdout );  
    }  
}
```

```
12345678910  
Process finished with exit code 0
```

# Example 1

- We will create a sample program to illustrate how to use fork() to create a child process, wait for the child process to terminate, and display the parent and child process ID in both processes.

# fork.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/types.h>
5 #include <sys/wait.h>
6
7 int main(int argc, char **argv) {
8     pid_t pid;
9     int status;
10
11     pid = fork();
12     if (pid == 0) { /* this is child process */
13         printf("This is the child process, my PID is %ld and my parent PID is %ld\n",
14                (long)getpid(), (long)getppid());
15     } else if (pid > 0) { /* this is the parent process */
16         printf("This is the parent process, my PID is %ld and the child PID is %ld\n",
17                (long)getpid(), (long)pid);
18
19         printf("Wait for the child process to terminate\n");
20 }
```

# fork.c

```
18     printf("Wait for the child process to terminate\n");
19     wait(&status); /* wait for the child process to terminate */
20     if (WIFEXITED(status)) { /* child process terminated normally */
21         printf("Child process exited with status = %d\n", WEXITSTATUS(status));
22     } else { /* child process did not terminate normally */
23         printf("ERROR: Child process did not terminate normally!\n");
24         /* look at the man page for wait (man 2 wait) to
25            determine how the child process was terminated */
26     }
27 }
28 } else { /* we have an error in process creation */
29     perror("fork");
30     exit(EXIT_FAILURE);
31 }
32
33 printf("[%ld]: Exiting program .....\\n", (long)getpid());
34
35 return 0;
36 }
37 }
```

# fork.c

```
[base] mahmutunan@MacBook-Pro lecture17 % ./exercise1
This is the parent process, my PID is 90695 and the child PID is 90696
Wait for the child process to terminate
This is the child process, my PID is 90696 and my parent PID is 90695
[90696]: Exiting program .....
Child process exited with status = 0
[90695]: Exiting program .....
(base) mahmutunan@MacBook-Pro lecture17 %
```

# exec()

- execl, execlp, execle, execv, execvp, execvpe - execute a file
- The **exec()** family of functions replaces the current process image with a new process image.

# exec()

- Note that the child process is a copy of the parent process and control is split at the invocation of the fork() call between the parent and the child process.
- If we like the child process to execute a different program other than making a copy of the parent process, we can use the exec family of system calls to replace the current process image with a new one.

- Here is the C APIs for the exec family of system calls:

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

```
int execl(const char *pathname, const char
          *arg, ...);
int execlp(const char *filename, const char
           *arg, ...);
int execle(const char *pathname, const char
           *arg, ..., char * const envp[]);
int execv(const char *pathname, char *const
          argv[]);
int execvp(const char *filename, char *const
          argv[]);
int execvpe(const char *filename, char *const
            argv[], char *const envp[]);
```

- We will use the `execl()` to replace the child process created by `fork()`.
- The `execl()` function takes as arguments the full pathname of the executable along with a pointer to an array of characters for each argument.
- Since we can have a variable number of arguments, the last argument is a null pointer.

# p1.c

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char **argv) {
    printf("Hello from p1, process id= () %d\n", getpid());
    char *args[]={ "Hello", "CS", "332", NULL};
    execv( path: "./p2", args);
    printf("we are not supposed to see this text");
    return 0;
}
```

# p2

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

int main(int argc, char **argv) {
    printf("Hello from p2, process id= () %d\n", getpid());
    printf("The arguments are %s %s %s\n", argv[0], argv[1], argv[2]);
    printf("Now, the child process will terminate\n");
    return 0;
}
```

# compile & run

```
[base] mahmutunan@MacBook-Pro lecture18 % gcc -Wall p1.c -o p1
[base] mahmutunan@MacBook-Pro lecture18 % gcc -Wall p2.c -o p2
[base] mahmutunan@MacBook-Pro lecture18 % ./p1
Hello from p1, process id= () 30983
Hello from p2, process id= () 30983
The arguments are Hello CS 332
Now, the child process will terminate
(base) mahmutunan@MacBook-Pro lecture18 %
```

# p1.c

- Let's modify it a little bit and fork the process

```
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char **argv) {
    printf("Hello from p1, process id= () %d\n", getpid());
    int pid=fork();
    int status;
    if (pid == 0) {
        char *args[] = {"Hello", "CS", "332", NULL};
        execv( path: "./p2", args);
        printf("we are not supposed to see this text");
    }
    else if(pid>0){
        wait(&status);
    }
    return 0;
}
```

# p2.c

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

int main(int argc, char **argv) {
    printf("Hello from p2, process id= () %d\n", getpid());
    printf("The arguments are %s %s %s\n", argv[0], argv[1], argv[2]);
    printf("Now, the child process will terminate\n");
    return 0;
}
```

# compile&run

```
(base) mahmutunan@MacBook-Pro lecture18 % gcc -Wall p1.c -o p1
(base) mahmutunan@MacBook-Pro lecture18 % gcc -Wall p2.c -o p2
(base) mahmutunan@MacBook-Pro lecture18 % ./p1
Hello from p1, process id= () 30922
Hello from p2, process id= () 30923
The arguments are Hello CS 332
Now, the child process will terminate
(base) mahmutunan@MacBook-Pro lecture18 %
```

# forkexec.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/types.h>
5  #include <sys/wait.h>
6
7  int main(int argc, char **argv) {
8      pid_t pid;
9      int status;
10
11      pid = fork();
```

```

12     if (pid == 0) { /* this is child process */
13         execl(path: "/usr/bin/uname", arg0: "uname", "-a", (char *)NULL);
14         printf("If you see this statement then execl failed ;-(\n");
15         perror("execl");
16         exit(-1);
17     } else if (pid > 0) { /* this is the parent process */
18         printf("Wait for the child process to terminate\n");
19         wait(&status); /* wait for the child process to terminate */
20         if (WIFEXITED(status)) { /* child process terminated normally */
21             printf("Child process exited with status = %d\n", WEXITSTATUS(status));
22         } else { /* child process did not terminate normally */
23             printf("Child process did not terminate normally!\n");
24             /* look at the man page for wait (man 2 wait) to determine
25              how the child process was terminated */
26         }
27     } else { /* we have an error */
28         perror("fork"); /* use perror to print the system error message */
29         exit(EXIT_FAILURE);
30     }
31
32     printf("[%ld]: Exiting program .....\\n", (long)getpid());
33
34     return 0;
35 }
```

# compile & run

```
[base] mahmutunan@MacBook-Pro lecture18 % gcc -Wall forkexec1.c -o exercise1
[base] mahmutunan@MacBook-Pro lecture18 % ./exercise1
Wait for the child process to terminate
Darwin MacBook-Pro.local 19.6.0 Darwin Kernel Version 19.6.0: Thu Jun 18 20:49:00 PDT
2020; root:xnu-6153.141.1~1/RELEASE_X86_64 x86_64
Child process exited with status = 0
[1290]: Exiting program .....
(base) mahmutunan@MacBook-Pro lecture18 %
```

- Let us look at the different versions of the exec functions. There are two classes of exec functions based on whether the argument is a list of separate values (l versions) or the argument is a vector (v versions):
- functions that take a variable number of command-lines arguments each as an array of characters terminated with a null character and the last argument is a null pointer –  
*(char \*)NULL (execl, execlp, and execle)*
- functions that take the command-line arguments as a pointer to an array of pointers to the arguments, similar to argv parameter used by the main method (execv, execvp, and execvpe)

- Functions that have *p* in the name use *filename* as the first argument while functions without *p* use the *pathname* as the first argument. If the filename contains a slash character (*/*), it is considered as a pathname, otherwise, all directories specified by the PATH environment variable are searched for the executable.
- Functions that end in *e* have an additional argument – a pointer to an array of pointers to the environment strings.

# forkexecv.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/types.h>
5 #include <sys/wait.h>
6
7 int main(int argc, char **argv) {
8     pid_t pid;
9     int status;
10    char *args[] = {"uname", "-a", (char *)NULL};
11
12    pid = fork();
13    if (pid == 0) { /* this is child process */
14        execv( path: "/usr/bin/uname", args);
15        printf("If you see this statement then execl failed ;-(\n");
16        perror("execv");
17        exit(-1);
18    } else if (pid > 0) { /* this is the parent process */
```

# forkexecv.c

```
18 } else if (pid > 0) { /* this is the parent process */
19     printf("Wait for the child process to terminate\n");
20     wait(&status); /* wait for the child process to terminate */
21     if (WIFEXITED(status)) { /* child process terminated normally */
22         printf("Child process exited with status = %d\n", WEXITSTATUS(status));
23     } else { /* child process did not terminate normally */
24         printf("Child process did not terminate normally!\n");
25         /* look at the man page for wait (man 2 wait) to determine
26             how the child process was terminated */
27     }
28 } else { /* we have an error */
29     perror("fork"); /* use perror to print the system error message */
30     exit(EXIT_FAILURE);
31 }
32
33 printf("[%ld]: Exiting program .....%n", (long)getpid());
34
35 return 0;
36 }
37 }
```

# compile & run

```
(base) mahmutunan@MacBook-Pro lecture18 % gcc -Wall forkexecv.c -o exercise2
(base) mahmutunan@MacBook-Pro lecture18 % ./exercise2
Wait for the child process to terminate
Darwin MacBook-Pro.local 19.6.0 Darwin Kernel Version 19.6.0: Thu Jun 18 20:49:00 PDT
2020; root:xnu-6153.141.1~1/RELEASE_X86_64 x86_64
Child process exited with status = 0
[30193]: Exiting program .....
(base) mahmutunan@MacBook-Pro lecture18 %
```

In order to see the difference between execl and execv, here is a line of code executing a

```
ls -l -R -a
```

**with execl :**

```
execl("/bin/ls", "ls", "-l", "-R", "-a", NULL);
```

**with execv :**

```
char* arr[] = {"ls", "-l", "-R", "-a", NULL};  
execv("/bin/ls", arr);
```

The array of char\* sent to execv will be passed to /bin/ls as argv  
(in `int main(int argc, char **argv)`)

# forkexecvp.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/types.h>
5  #include <sys/wait.h>
6
7  ► int main(int argc, char **argv) {
8      pid_t pid;
9      int status;
10
11     if (argc < 2) {
12         printf("Usage: %s <command> [args]\n", argv[0]);
13         exit(-1);
14     }
15
16     pid = fork();
17     if (pid == 0) { /* this is child process */
18         execvp(argv[1], &argv[1]);
19         printf("If you see this statement then exec failed ;-(\n");
20         perror("execvp");
21         exit(-1);
```

# forkexecvp.c

```
22 } else if (pid > 0) { /* this is the parent process */
23     printf("Wait for the child process to terminate\n");
24     wait(&status); /* wait for the child process to terminate */
25     if (WIFEXITED(status)) { /* child process terminated normally */
26         printf("Child process exited with status = %d\n", WEXITSTATUS(status));
27     } else { /* child process did not terminate normally */
28         printf("Child process did not terminate normally!\n");
29         /* look at the man page for wait (man 2 wait) to determine
30            how the child process was terminated */
31     }
32 } else { /* we have an error */
33     perror("fork"); /* use perror to print the system error message */
34     exit(EXIT_FAILURE);
35 }
36
37 printf("[%ld]: Exiting program ....\n", (long)getpid());
38
39 return 0;
40 }
41 }
```

# hello.c

```
#include <stdio.h>
int main(int argc, char **argv) {
    printf("Hello from the execvp()\n");
    return 0;
}
```

# compile & run

```
(base) mahmutunan@MacBook-Pro lecture18 % gcc -Wall forkexecvp.c -o exercise3
(base) mahmutunan@MacBook-Pro lecture18 % gcc -Wall hello.c -o hello
(base) mahmutunan@MacBook-Pro lecture18 % ./exercise3 ./hello
Wait for the child process to terminate
Hello from the execvp()
Child process exited with status = 0
[30387]: Exiting program .....
(base) mahmutunan@MacBook-Pro lecture18 %
```

# forkexecvp2.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/types.h>
5 #include <sys/wait.h>
6
7 int main(int argc, char **argv) {
8     pid_t pid;
9     int status;
10
11    if (argc < 2) {
12        printf("Usage: %s <command> [args]\n", argv[0]);
13        exit(-1);
14    }
15
16    pid = fork();
17    if (pid == 0) { /* this is child process */
18        execvp(argv[1], &argv[1]);
19        printf("If you see this statement then exec failed ;-(\n");
20        perror("execvp");
21        exit(-1);
```

```
22 } else if (pid > 0) { /* this is the parent process */
23     char name[BUFSIZ];
24
25     printf("[%d]: Please enter your name: ", getpid());
26     scanf("%s", name);
27     printf("[%d-stdout]: Hello %s!\n", getpid(), name);
28     fprintf(stderr, "[%d-stderr]: Hello %s!\n", getpid(), name);
29
30     wait(&status); /* wait for the child process to terminate */
31     if (WIFEXITED(status)) { /* child process terminated normally */
32         printf("Child process exited with status = %d\n", WEXITSTATUS(status));
33     } else { /* child process did not terminate normally */
34         printf("Child process did not terminate normally!\n");
35         /* look at the man page for wait (man 2 wait) to determine
36          how the child process was terminated */
37     }
38 } else { /* we have an error */
39     perror("fork"); /* use perror to print the system error message */
40     exit(EXIT_FAILURE);
41 }
42
43 printf("[%ld]: Exiting program ....\n", (long)getpid());
44
45 return 0;
46 }
47 }
```

- Here is the C APIs for the exec family of system calls:

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

```
int execl(const char *pathname, const char
          *arg, ...);
int execlp(const char *filename, const char
           *arg, ...);
int execle(const char *pathname, const char
           *arg, ..., char * const envp[]);
int execv(const char *pathname, char *const
          argv[]);
int execvp(const char *filename, char *const
          argv[]);
int execvpe(const char *filename, char *const
            argv[], char *const envp[]);
```

# ls -lh

```
[base] mahmutunan@MacBook-Pro lecture19 % ls -lh
total 240
-rw-r--r--@ 1 mahmutunan  staff  239B Oct  9 12:56 execl.c
-rw-r--r--@ 1 mahmutunan  staff  341B Oct  9 13:13 execle.c
-rw-r--r--@ 1 mahmutunan  staff  221B Oct  9 12:58 execlp.c
-rw-r--r--@ 1 mahmutunan  staff  194B Oct  9 12:59 execv.c
-rw-r--r--@ 1 mahmutunan  staff  326B Oct  9 13:19 execve.c
-rw-r--r--@ 1 mahmutunan  staff  198B Oct  9 13:01 execvp.c
-rwxr-xr-x  1 mahmutunan  staff   12K Oct  9 12:56 exercise1
-rwxr-xr-x  1 mahmutunan  staff   12K Oct  9 12:58 exercise2
-rwxr-xr-x  1 mahmutunan  staff   12K Oct  9 13:00 exercise3
-rwxr-xr-x  1 mahmutunan  staff   12K Oct  9 13:02 exercise4
-rwxr-xr-x  1 mahmutunan  staff   12K Oct  9 13:13 exercise5
-rwxr-xr-x  1 mahmutunan  staff   12K Oct  9 13:19 exercise6
```

# execl

```
int main(void) {
    char *binaryPath = "/bin/ls";
    char *arg0="ls";
    char *arg1 = "-lh";
    char *arg2 = "/Users/mahmutunan/Desktop/lecture19";

    execl(binaryPath, arg0, arg1, arg2, NULL);

    return 0;
}
```

```
[base] mahmutunan@MacBook-Pro lecture19 % gcc execl.c -o exercise1
[base] mahmutunan@MacBook-Pro lecture19 % ./exercise1
total 160
```

```
-rw-r--r--@ 1 mahmutunan  staff   239B Oct  9 12:56 execl.c
-rw-r--r--@ 1 mahmutunan  staff   220B Oct  9 12:33 execlp.c
-rw-r--r--@ 1 mahmutunan  staff   192B Oct  9 12:29 execv.c
-rw-r--r--@ 1 mahmutunan  staff   196B Oct  9 12:37 execvp.c
-rwxr-xr-x  1 mahmutunan  staff   12K Oct  9 12:56 exercise1
-rwxr-xr-x  1 mahmutunan  staff   12K Oct  9 12:33 exercise2
-rwxr-xr-x  1 mahmutunan  staff   12K Oct  9 12:34 exercise3
-rwxr-xr-x  1 mahmutunan  staff   12K Oct  9 12:37 exercise4
```

```
(base) mahmutunan@MacBook-Pro lecture19 %
```

# execp

```
1 #include <unistd.h>
2
3 int main(void) {
4     char *commandName = "ls";
5     char *arg1 = "-lh";
6     char *arg2 = "/Users/mahmutunan/Desktop/lecture19";
7
8     execlp(commandName, commandName, arg1, arg2, NULL);
9
10    return 0;
11 }
```

```
[base] mahmutunan@MacBook-Pro lecture19 % gcc execp.c -o exercise2
[base] mahmutunan@MacBook-Pro lecture19 % ./exercise2
total 160
-rw-r--r--@ 1 mahmutunan  staff   239B Oct  9 12:56 exec.c
-rw-r--r--@ 1 mahmutunan  staff   221B Oct  9 12:58 execp.c
-rw-r--r--@ 1 mahmutunan  staff   192B Oct  9 12:29 execv.c
-rw-r--r--@ 1 mahmutunan  staff   196B Oct  9 12:37 execvp.c
-rwxr-xr-x  1 mahmutunan  staff   12K Oct  9 12:56 exercise1
-rwxr-xr-x  1 mahmutunan  staff   12K Oct  9 12:58 exercise2
-rwxr-xr-x  1 mahmutunan  staff   12K Oct  9 12:34 exercise3
-rwxr-xr-x  1 mahmutunan  staff   12K Oct  9 12:37 exercise4
[base] mahmutunan@MacBook-Pro lecture19 %
```

# execv

```
1 #include <unistd.h>
2
3 int main(void) {
4     char *binaryPath = "/bin/ls";
5     char *args[] = {"ls", "-lh", "/Users/mahmutunan/Desktop/lecture19", NULL};
6
7     execv(binaryPath, args);
8
9     return 0;
10 }
```

```
[base] mahmutunan@MacBook-Pro lecture19 % gcc execv.c -o exercise3
[base] mahmutunan@MacBook-Pro lecture19 % ./exercise3
total 160
-rw-r--r--@ 1 mahmutunan  staff   239B Oct  9 12:56 execl.c
-rw-r--r--@ 1 mahmutunan  staff   221B Oct  9 12:58 execlp.c
-rw-r--r--@ 1 mahmutunan  staff   194B Oct  9 12:59 execv.c
-rw-r--r--@ 1 mahmutunan  staff   196B Oct  9 12:37 execvp.c
-rwxr-xr-x  1 mahmutunan  staff   12K Oct  9 12:56 exercise1
-rwxr-xr-x  1 mahmutunan  staff   12K Oct  9 12:58 exercise2
-rwxr-xr-x  1 mahmutunan  staff   12K Oct  9 13:00 exercise3
-rwxr-xr-x  1 mahmutunan  staff   12K Oct  9 12:37 exercise4
[base] mahmutunan@MacBook-Pro lecture19 %
```

# execvp

```
1 #include <unistd.h>
2
3 int main(void) {
4     char *commandName= "ls";
5     char *args[] = {commandName, "-lh", "/Users/mahmutunan/Desktop/lecture19", NULL};
6
7     execvp(commandName, args);
8
9     return 0;
10 }
```

```
[base] mahmutunan@MacBook-Pro lecture19 % gcc execvp.c -o exercise4
[base] mahmutunan@MacBook-Pro lecture19 % ./exercise4
total 160
-rw-r--r--@ 1 mahmutunan  staff   239B Oct  9 12:56 execl.c
-rw-r--r--@ 1 mahmutunan  staff   221B Oct  9 12:58 execlp.c
-rw-r--r--@ 1 mahmutunan  staff   194B Oct  9 12:59 execv.c
-rw-r--r--@ 1 mahmutunan  staff   198B Oct  9 13:01 execvp.c
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 12:56 exercise1
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 12:58 exercise2
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 13:00 exercise3
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 13:02 exercise4
[base] mahmutunan@MacBook-Pro lecture19 %
```

# Environment / Environment Variable

- An important Unix concept is the **environment**, which is defined by environment variables. Some are set by the system, others by you, yet others by the shell, or any program that loads another program.
- A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.
- When you log in to the system, the shell undergoes a phase called **initialization** to set up the environment. Environment variables allow you to customize how the system works and the behavior of the applications on the system.

# Environment / Environment Variable

Sr.No.	Variable & Description
1	<b>DISPLAY</b> Contains the identifier for the display that <b>X11</b> programs should use by default.
2	<b>HOME</b> Indicates the home directory of the current user: the default argument for the cd <b>built-in</b> command.
3	<b>IFS</b> Indicates the <b>Internal Field Separator</b> that is used by the parser for word splitting after expansion.
4	<b>LANG</b> LANG expands to the default system locale; LC_ALL can be used to override this. For example, if its value is <b>pt_BR</b> , then the language is set to (Brazilian) Portuguese and the locale to Brazil.
5	<b>LD_LIBRARY_PATH</b> A Unix system with a dynamic linker, contains a colon-separated list of directories that the dynamic linker should search for shared objects when building a process image after exec, before searching in any other directories.

6	<b>PATH</b> Indicates the search path for commands. It is a colon-separated list of directories in which the shell looks for commands.
7	<b>PWD</b> Indicates the current working directory as set by the cd command.
8	<b>RANDOM</b> Generates a random integer between 0 and 32,767 each time it is referenced.

# execle

```
1 #include <unistd.h>
2
3 int main(void) {
4     char *binaryPath= "/bin/bash";
5     char *arg1 = "-c";
6     char *arg2 = "echo \"Visit $HOSTNAME:$PORT from your browser.\"";
7     char *const env[] = {"HOSTNAME=https://www.uab.edu/cas/computerscience/", "PORT=8080", NULL};
8
9     execle(binaryPath, binaryPath,arg1, arg2, NULL, env);
10
11    return 0;
12 }
```

```
[(base) mahmutunan@MacBook-Pro lecture19 % gcc execle.c -o exercise5
(base) mahmutunan@MacBook-Pro lecture19 % ./exercise5
Visit https://www.uab.edu/cas/computerscience/:8080 from your browser.
(base) mahmutunan@MacBook-Pro lecture19 % _
```

# execve

```
1 #include <unistd.h>
2
3 int main(void) {
4     char *binaryPath= "/bin/bash";
5     char *const args[] = {binaryPath,"-c","echo \"Visit $HOSTNAME:$PORT from your browser.\\"",NULL};
6     char *const env[] = {"HOSTNAME=https://www.uab.edu/cas/computerscience/", "PORT=8080", NULL};
7
8     execve(binaryPath, args, env);
9
10    return 0;
11}
```

```
[base] mahmutunan@MacBook-Pro lecture19 % gcc execve.c -o exercise6
[base] mahmutunan@MacBook-Pro lecture19 % ./exercise6
Visit https://www.uab.edu/cas/computerscience/:8080 from your browser.
(base) mahmutunan@MacBook-Pro lecture19 %
```

# forkexec.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/types.h>
5  #include <sys/wait.h>
6
7  int main(int argc, char **argv) {
8      pid_t pid;
9      int status;
10
11      pid = fork();
```

```
12     if (pid == 0) { /* this is child process */
13         execl(path: "/usr/bin/uname", arg0: "uname", "-a", (char *)NULL);
14         printf("If you see this statement then execl failed ;-(\n");
15         perror("execl");
16         exit(-1);
17     } else if (pid > 0) { /* this is the parent process */
18         printf("Wait for the child process to terminate\n");
19         wait(&status); /* wait for the child process to terminate */
20         if (WIFEXITED(status)) { /* child process terminated normally */
21             printf("Child process exited with status = %d\n", WEXITSTATUS(status));
22         } else { /* child process did not terminate normally */
23             printf("Child process did not terminate normally!\n");
24             /* look at the man page for wait (man 2 wait) to determine
25              how the child process was terminated */
26         }
27     } else { /* we have an error */
28         perror("fork"); /* use perror to print the system error message */
29         exit(EXIT_FAILURE);
30     }
31
32     printf("[%ld]: Exiting program .....\\n", (long)getpid());
33
34     return 0;
35 }
```

# compile & run

```
[base] mahmutunan@MacBook-Pro lecture18 % gcc -Wall forkexec1.c -o exercise1
[base] mahmutunan@MacBook-Pro lecture18 % ./exercise1
Wait for the child process to terminate
Darwin MacBook-Pro.local 19.6.0 Darwin Kernel Version 19.6.0: Thu Jun 18 20:49:00 PDT
2020; root:xnu-6153.141.1~1/RELEASE_X86_64 x86_64
Child process exited with status = 0
[1290]: Exiting program .....
(base) mahmutunan@MacBook-Pro lecture18 %
```

- Let us look at the different versions of the exec functions. There are two classes of exec functions based on whether the argument is a list of separate values (l versions) or the argument is a vector (v versions):
- functions that take a variable number of command-lines arguments each as an array of characters terminated with a null character and the last argument is a null pointer –  
*(char \*)NULL (execl, execlp, and execle)*
- functions that take the command-line arguments as a pointer to an array of pointers to the arguments, similar to argv parameter used by the main method (execv, execvp, and execvpe)

- Functions that have *p* in the name use *filename* as the first argument while functions without *p* use the *pathname* as the first argument. If the filename contains a slash character (*/*), it is considered as a pathname, otherwise, all directories specified by the PATH environment variable are searched for the executable.
- Functions that end in *e* have an additional argument – a pointer to an array of pointers to the environment strings.

# forkexecv.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/types.h>
5 #include <sys/wait.h>
6
7 int main(int argc, char **argv) {
8     pid_t pid;
9     int status;
10    char *args[] = {"uname", "-a", (char *)NULL};
11
12    pid = fork();
13    if (pid == 0) { /* this is child process */
14        execv( path: "/usr/bin/uname", args);
15        printf("If you see this statement then execl failed ;-(\n");
16        perror("execv");
17        exit(-1);
18    } else if (pid > 0) { /* this is the parent process */
```

# forkexecv.c

```
18 } else if (pid > 0) { /* this is the parent process */
19     printf("Wait for the child process to terminate\n");
20     wait(&status); /* wait for the child process to terminate */
21     if (WIFEXITED(status)) { /* child process terminated normally */
22         printf("Child process exited with status = %d\n", WEXITSTATUS(status));
23     } else { /* child process did not terminate normally */
24         printf("Child process did not terminate normally!\n");
25         /* look at the man page for wait (man 2 wait) to determine
26             how the child process was terminated */
27     }
28 } else { /* we have an error */
29     perror("fork"); /* use perror to print the system error message */
30     exit(EXIT_FAILURE);
31 }
32
33 printf("[%ld]: Exiting program .....%n", (long)getpid());
34
35 return 0;
36 }
37 }
```

# compile & run

```
(base) mahmutunan@MacBook-Pro lecture18 % gcc -Wall forkexecv.c -o exercise2
(base) mahmutunan@MacBook-Pro lecture18 % ./exercise2
Wait for the child process to terminate
Darwin MacBook-Pro.local 19.6.0 Darwin Kernel Version 19.6.0: Thu Jun 18 20:49:00 PDT
2020; root:xnu-6153.141.1~1/RELEASE_X86_64 x86_64
Child process exited with status = 0
[30193]: Exiting program .....
(base) mahmutunan@MacBook-Pro lecture18 %
```

In order to see the difference between execl and execv, here is a line of code executing a

```
ls -l -R -a
```

**with execl :**

```
execl("/bin/ls", "ls", "-l", "-R", "-a", NULL);
```

**with execv :**

```
char* arr[] = {"ls", "-l", "-R", "-a", NULL};  
execv("/bin/ls", arr);
```

The array of char\* sent to execv will be passed to /bin/ls as argv  
(in `int main(int argc, char **argv)`)

# forkexecvp.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/types.h>
5  #include <sys/wait.h>
6
7  ► int main(int argc, char **argv) {
8      pid_t pid;
9      int status;
10
11     if (argc < 2) {
12         printf("Usage: %s <command> [args]\n", argv[0]);
13         exit(-1);
14     }
15
16     pid = fork();
17     if (pid == 0) { /* this is child process */
18         execvp(argv[1], &argv[1]);
19         printf("If you see this statement then exec failed ;-(\n");
20         perror("execvp");
21         exit(-1);
```

# forkexecvp.c

```
22 } else if (pid > 0) { /* this is the parent process */
23     printf("Wait for the child process to terminate\n");
24     wait(&status); /* wait for the child process to terminate */
25     if (WIFEXITED(status)) { /* child process terminated normally */
26         printf("Child process exited with status = %d\n", WEXITSTATUS(status));
27     } else { /* child process did not terminate normally */
28         printf("Child process did not terminate normally!\n");
29         /* look at the man page for wait (man 2 wait) to determine
30            how the child process was terminated */
31     }
32 } else { /* we have an error */
33     perror("fork"); /* use perror to print the system error message */
34     exit(EXIT_FAILURE);
35 }
36
37 printf("[%ld]: Exiting program ....\n", (long)getpid());
38
39 return 0;
40 }
41 }
```

# hello.c

```
#include <stdio.h>
int main(int argc, char **argv) {
    printf("Hello from the execvp()\n");
    return 0;
}
```

# compile & run

```
(base) mahmutunan@MacBook-Pro lecture18 % gcc -Wall forkexecvp.c -o exercise3
(base) mahmutunan@MacBook-Pro lecture18 % gcc -Wall hello.c -o hello
(base) mahmutunan@MacBook-Pro lecture18 % ./exercise3 ./hello
Wait for the child process to terminate
Hello from the execvp()
Child process exited with status = 0
[30387]: Exiting program .....
(base) mahmutunan@MacBook-Pro lecture18 %
```

# forkexecvp2.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/types.h>
5 #include <sys/wait.h>
6
7 int main(int argc, char **argv) {
8     pid_t pid;
9     int status;
10
11    if (argc < 2) {
12        printf("Usage: %s <command> [args]\n", argv[0]);
13        exit(-1);
14    }
15
16    pid = fork();
17    if (pid == 0) { /* this is child process */
18        execvp(argv[1], &argv[1]);
19        printf("If you see this statement then exec failed ;-(\n");
20        perror("execvp");
21        exit(-1);
```

```
22 } else if (pid > 0) { /* this is the parent process */
23     char name[BUFSIZ];
24
25     printf("[%d]: Please enter your name: ", getpid());
26     scanf("%s", name);
27     printf("[%d-stdout]: Hello %s!\n", getpid(), name);
28     fprintf(stderr, "[%d-stderr]: Hello %s!\n", getpid(), name);
29
30     wait(&status); /* wait for the child process to terminate */
31     if (WIFEXITED(status)) { /* child process terminated normally */
32         printf("Child process exited with status = %d\n", WEXITSTATUS(status));
33     } else { /* child process did not terminate normally */
34         printf("Child process did not terminate normally!\n");
35         /* look at the man page for wait (man 2 wait) to determine
36            how the child process was terminated */
37     }
38 } else { /* we have an error */
39     perror("fork"); /* use perror to print the system error message */
40     exit(EXIT_FAILURE);
41 }
42
43 printf("[%ld]: Exiting program ....\n", (long)getpid());
44
45 return 0;
46 }
47 }
```

# compile & run

```
[base] mahmutunan@MacBook-Pro lecture19 % gcc -Wall forkexecvp2.c -o exercise7
[base] mahmutunan@MacBook-Pro lecture19 % ./exercise7 ls -lh
[65577]: Please enter your name: total 280
-rw-r--r--@ 1 mahmutunan  staff   239B Oct  9 12:56 execl.c
-rw-r--r--@ 1 mahmutunan  staff   341B Oct  9 13:13 execle.c
-rw-r--r--@ 1 mahmutunan  staff   221B Oct  9 12:58 execlp.c
-rw-r--r--@ 1 mahmutunan  staff   194B Oct  9 12:59 execv.c
-rw-r--r--@ 1 mahmutunan  staff   326B Oct  9 13:19 execve.c
-rw-r--r--@ 1 mahmutunan  staff   198B Oct  9 13:01 execvp.c
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 12:56 exercise1
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 12:58 exercise2
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 13:00 exercise3
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 13:02 exercise4
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 13:13 exercise5
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 13:19 exercise6
-rwxr-xr-x  1 mahmutunan  staff   13K Oct  9 13:51 exercise7
-rw-r--r--@ 1 mahmutunan  staff   1.8K Oct  5 19:48 forkexecvp2.c
mahmut
[65577-stdout]: Hello mahmut!
[65577-stderr]: Hello mahmut!
Child process exited with status = 0
[65577]: Exiting program .....
(base) mahmutunan@MacBook-Pro lecture19 %
```

# Linux

- Linux and Unix are different, but they do have a relationship with each other
  - Linux is derived from Unix.
- Linux is just the kernel and not the complete OS. This Linux kernel is generally packaged in Linux distributions which thereby makes it a complete OS
- There are many different versions of Linux, and the core of Linux is free to distribute and use.
- Normally, distributions are made for specific reasons and have been tailored to address a series of concerns

# Processes in a Linux environment

- User processes in a Linux environment could be in one of the following three states: foreground, background, or suspended.
- Most interactive applications that take input from the keyboard or command-line argument and display output in a terminal are considered as foreground processes. Till now, we have been executed all our programs as foreground processes by typing the name of the command in the bash shell.

- Non-interactive processes that are typically not connected to a terminal and execute in the background are considered as background processes.
- You can execute a program in the background by typing the program name followed by the symbol & at the end.
- You will notice that the shell returns the command-prompt with the background process number and the corresponding process identifier (PID) of the process that was created.
- For example:

```
$ nano myprog.c &
[1] 16946
$
```

- You can display various jobs that are currently running in the background using the *jobs* command. It will show the job number, the current state of the job, and the job name. For example:

```
$ jobs  
[1]+  Stopped                  nano myprog.c  
$
```

- If you like to list additional information such as the PID of the job you can use the -l option. For example:

```
$ jobs -l  
[1]+ 16946 Stopped (tty output) nano myprog.c  
$
```

In case of the above example, we have invoked an editor and it has been stopped since it requires terminal to display the output and continue. If we had created a non-interactive job in the background, then it would be in the running state.

For example:

```
$ sleep 20 &  
[2] 17513  
$ jobs  
[1]+  Stopped nano myprog.c  
[2]-  Running sleep 20 &  
$
```

- We can bring a job that is running in the background to foreground by using the *fg* command. For example, to switch the *sleep* process to foreground, we specify the job number after the % symbol:

```
$ fg %2  
sleep 20  
$ jobs  
[1]+  Stopped                 nano myprog.c  
$
```

- You will notice that the sleep process will return the command prompt in the terminal when it completes execution and when you type *jobs* again, it will only show one process. You can use *fg* command to switch to the editor using: *fg %1*.
- You can continue with your edits, save the file, and exit the editor.
- After you exit, if you type *jobs* again, you will notice that it does not list any processes since the *nano* process is no longer executing.
- If you like to suspend a foreground process then type *Control-Z* when the program is executing and that process will be suspended

- For example, if we started the sleep process in foreground and would like to suspend it, then type *Control-Z* and you will see a message similar to what you saw when you started a process in background:

```
$  
$ jobs  
$ sleep 100  
^Z  
[1]+  Stopped                 sleep 100  
$ jobs  
[1]+  Stopped                 sleep 100  
$
```

- However, notice that the sleep process is stopped (it is not running) unlike the previous case when it was running in the background. If you like the sleep process to continue then you have to use the *bg* command as follows:

```
$ bg %1  
[1]+ sleep 100 &  
$ jobs  
[1]+ Running sleep 100 &  
$  
[1]+ Done sleep 100  
$ jobs
```

- Now you notice that the sleep process is running and when it is done you will see the message *Done* in your terminal.
- There are special background processes that are started at system startup and they continue to run till the system is shutdown.
- These special background processes are called ***daemons***.
- These processes typically end in “d” and some examples are: systemd, crond, ntpd, nfsd, sshd, httpd, named.
- If you like to terminate a process that is executing in the foreground, you use Control-C to kill it.
- If you like to terminate a process in the background, you could bring it to foreground and then use Control-C or use the *kill* command to terminate the background process directly

- For example:

- \$ jobs

```
$ sleep 100 &  
[1] 1519
```

```
$ jobs
```

```
[1]+  Running
```

```
sleep 100 &
```

```
$ kill %1
```

```
[1]+  Terminated
```

```
sleep 100
```

```
$
```

```
$ jobs
```

\$ You can also provide the PID as the argument to *kill* command to terminate a process.

## Monitor processes in Linux environment

- You can use the *ps* command to display information about various processes running on a Linux system. Login to one of CS Linux systems and enter the *ps* command, you will see the following information displayed:

```
~~{2.00}~{unan@vulcan18:~}~~
[$ ps
   PID TTY          TIME CMD
 30544 pts/1        00:00:00 bash
 30692 pts/1        00:00:00 ps
```

# ps -man page

- ps - report a snapshot of the current processes.
- **ps [options]**

To see every process on the system using standard syntax:

```
ps -e  
ps -ef  
ps -eF  
ps -ely
```

To see every process on the system using BSD syntax:

```
ps ax  
ps axu
```

To print a process tree:

```
ps -ejH  
ps axjf
```

To get info about threads:

```
ps -eLf  
ps axms
```

To get security info:

```
ps -eo euser,ruser,suser,fuser,f,comm,label  
ps axZ  
ps -eM
```

<https://man7.org/linux/man-pages/man1/ps.1.html>

- By default, *ps* lists processes for the current user that are associated with the terminal that invoked the command and the output is unsorted.
- The following information is shown above: the process ID (PID), the terminal associated with the process (TTY), the cumulative CPU time in hh:mm:ss format (TIME), and the executable name (CMD).
- You will notice that there are two processes currently executing – bash and ps (the command you just executed) along with their corresponding process ID (in the first column under PID).

- The *ps* command has a large number of options that you can use to get more detailed information about the various processes currently running. We will look at some of these options below, you can find out more about these options using *man ps*.
- The *-u username* option lists all processes that belong to the user *username*:

```
$ ps -u unan
 PID TTY          TIME CMD
 30543 ?        00:00:00 sshd
 30544 pts/1    00:00:00 bash
 30716 pts/1    00:00:00 ps
```

- You can select all processes owned by you (runner of the **ps command**, root in this case), type:

```
ps -x
```

- We can also view every process running with **root** user privileges (real & effective ID) in user format.

```
ps -U root -u root
```

- The command below allows you to view the PID, PPID, username and command of a process.

```
$ ps -eo pid,ppid,user,cmd
```

- To select a specific process by its name, use the -C flag, this will also display all its child processes.

```
$ ps -C sshd
```

- Now you notice that there is an additional process (sshd) that belongs to you and there is no terminal associated with that process (hence the ? under the TTY column). This process was started by the OS when you connected to this computer using an SSH client. You can use the -f or -F option to get a full listing:

```
~~{2.00}~{unan@vulcan18:~}~~
$ ps -fu unan
UID          PID  PPID   C STIME  TTY          TIME CMD
unan        30543 30532   0 13:59 ?          00:00:00 sshd: unan@pts/1
unan        30544 30543   0 13:59 pts/1      00:00:00 -bash
unan        30731 30544   0 14:00 pts/1      00:00:00 ps -fu unan
~~{2.00}~{unan@vulcan18:~}~~
```

```
~~[Z:~]~$ unan@vulcan10:~]$ ~~~  
$ ps -Fu unan  
UID          PID  PPID   C   SZ   RSS  PSR STIME TTY          TIME CMD  
unan        30543 30532  0 49950  2784  0 13:59 ?          00:00:00 sshd: unan@pts/1  
unan        30544 30543  0 33493  3748  0 13:59 pts/1        00:00:00 -bash  
unan        30746 30544  0 42042  1940  0 14:00 pts/1        00:00:00 ps -Fu unan
```

Now we see several additional fields displayed such the user ID, parent PI, process with command-line options, etc.

By looking at the PID and PPID we can identify that the *ps* command was created by the *bash* process and the *bash* process was in turn created by the *sshd* process. We can display the process tree with the *ps* command using the --forest option:

```
~~{2.00}~{unan@vulcan18:~}~~  
$ ps -fu unan --forest  
UID          PID  PPID   C STIME TTY          TIME CMD  
unan        30543 30532  0 13:59 ?          00:00:00 sshd: unan@pts/1  
unan        30544 30543  0 13:59 pts/1       00:00:00 \_ -bash  
unan        30789 30544  0 14:01 pts/1       00:00:00      \_ ps -fu unan --forest
```

You can use *ps* to list every process currently running (not just processes that belong to you) using the *-e* option

```
ps -e | more  
ps -ef | more  
ps -eF | more  
ps -ely | more
```

You have to press the **spacebar** to scroll through the list.

- You can send the output to the `wc` command to determine the total number of processes currently running on the system, for example:

```
~~{2.00}~{unan@vulcan18:~}~~  
[$ ps -e | wc -l  
158
```

At this the above command was executed on one of the CS Linux system, we had 158 processes currently running on the system, even though there are only three processes that belong to you. What are all these processes? Who is running these processes?

- You can also use the *pstree* command to display the process tree (you can find out more about the various options supported by *pstree* using *man pstree*).
  - For example:

```
pstree -np | more
```

```
[\$ ps -e | more
```

PID	TTY	TIME	CMD
1	?	00:14:34	systemd
2	?	00:00:03	kthreadd
4	?	00:00:00	kworker/0:0H
6	?	00:00:29	ksoftirqd/0
7	?	00:00:05	migration/0
8	?	00:00:00	rcu_bh
9	?	00:10:59	rcu_sched
10	?	00:00:00	lru-add-drain
11	?	00:00:34	watchdog/0
12	?	00:00:29	watchdog/1
13	?	00:00:06	migration/1
14	?	00:00:23	ksoftirqd/1
16	?	00:00:00	[kernel]

- Similarly, you can use the `top` command (instead of `ps`) to display all processes currently running (you have to enter `q` to quit `top`).
- The `top` command provides a real-time update on the various processes running on the system.
- You can look at processes that belong to a specific user by typing `u` and the user name when `top` is running or start `top` with the `-u user` option.
- It provides a dynamic real-time view of the running system. Usually, this command shows the summary information of the system and the list of processes or threads which are currently managed by the Linux Kernel.
- <https://man7.org/linux/man-pages/man1/top.1.html>

# kill

- To terminate a process, we can use the *kill* command. The *kill* command can take the PID or the command name and terminate a specific process with the given PID or terminate all processes with the specified command name. We can also specify the type of signal, either as a signal name (e.g., KILL for kill) or signal number (9 for kill), to send to a process with the *kill* command. Here is an example that shows how to use the *kill* command to terminate a process using PID.

# kill - man page

- kill - terminate a process
- The command **kill** sends the specified *signal* to the specified processes or process groups.
- If no signal is specified, the TERM signal is sent. The default action for this signal is to terminate the process. This signal should be used in preference to the KILL signal (number 9), since a process may install a handler for the TERM signal in order to perform clean-up steps before terminating in an orderly fashion
- <https://man7.org/linux/man-pages/man1/kill.1.html>

```
~~{2.00}~{unan@vulcan18:~}~~
$ sleep 100 &
[1] 30898
~~{2.00}~{unan@vulcan18:~}~~
$ kill -TERM 30898
[1]+ Terminated                 sleep 100
~~{2.00}~{unan@vulcan18:~}~~
$
```

You can find the complete list of signal names and numbers using the `-l` option of *kill* command.

We will discuss more about signals in the later labs

```
kill SIGNAL PID
```

- Where SIGNAL is the signal to be sent and PID is the Process ID to be killed.
- Let's say we need to terminate the process 3827. We need to send the kill signal;

```
kill -9 3827
```

To display all the available signals, we should use below command option

```
kill -l
```

```
~~~ i ~~~~~~ turan@evulcan9: ~~~~~~  
$ kill -l  
 1) SIGHUP      2) SIGINT      3) SIGQUIT      4) SIGILL      5) SIGTRAP  
 6) SIGABRT     7) SIGBUS      8) SIGFPE       9) SIGKILL     10) SIGUSR1  
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM  
16) SIGSTKFLT    17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP  
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ  
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR  
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8  
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13  
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12  
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2  
63) SIGRTMAX-1  64) SIGRTMAX
```

# Explore the /proc file system

- The proc file system (procfs) is a virtual file system that is created by the OS at system boot time to provide an interface between the kernel space and user space.
- It is commonly mounted at /proc.
- It provides information of processes currently running on the system and tools such as *ps* use this to display information about these processes

# ls -l /proc

```
~~{1.06}~{unan@vulcan16:~}~~
$ ls -l /proc
total 0
dr-xr-xr-x 9 root          root          0 Apr 28 14:54 1
dr-xr-xr-x 9 root          root          0 Sep 23 22:27 10
dr-xr-xr-x 9 root          root          0 Oct  1 21:17 10596
dr-xr-xr-x 9 root          root          0 Oct  1 21:17 10597
dr-xr-xr-x 9 root          root          0 Sep 23 22:27 11
dr-xr-xr-x 9 root          root          0 Sep 23 22:27 111
dr-xr-xr-x 9 root          root          0 Oct  1 21:16 1175
dr-xr-xr-x 9 root          root          0 Oct  1 21:16 1176
dr-xr-xr-x 9 root          root          0 Oct  1 21:16 1177
dr-xr-xr-x 9 root          root          0 Oct  1 21:16 1179
dr-xr-xr-x 9 root          root          0 Sep 23 22:27 12
dr-xr-xr-x 9 root          root          0 Oct  1 21:16 1218
dr-xr-xr-x 9 root          root          0 Oct  1 21:16 1296
dr-xr-xr-x 9 postfix        postfix        0 Oct  1 21:16 1298
dr-xr-xr-x 9 root          root          0 Oct  1 21:16 1299
dr-xr-xr-x 9 root          root          0 Sep 23 22:27 13
dr-xr-xr-x 9 root          root          0 Oct  1 21:16 1313
```

File	Description
/proc/cpuinfo	information about the CPU architecture, used by <i>lscpu</i> command
/proc/loadavg	system load averages data, used by <i>uptime</i> command
/proc/meminfo	memory usage statistics on the system, used by <i>free</i> command
/proc/stat	kernel/system statistics
/proc/version	version of the kernel currently running on the system, used by <i>uname</i> command
/proc/[pid]	a subdirectory for each running process
/proc/[pid]/cmdline	command string of the process along with arguments separated by null ('\0') character
/proc/[pid]/cmd	a symbolic link to the current working directory of the process
/proc/[pid]/environ	environment variables and values used by the process separated by null ('\0') character (use strings /proc/[pid]/environ to display the environment variable and values)
/proc/[pid]/maps	information on memory mapped regions of the process
/proc/[pid]/mem	information to access pages of the process through I/O calls
/proc/[pid]/stat	status information about the process, used by <i>ps</i> command
/proc/[pid]/statm	status of memory used by the process, measured in pages

- Let's take a look at one of the files :

```
# cat /proc/meminfo
```

```
~~{      }~{unan@vulcan0:~}~~  
$ cat /proc/meminfo  
MemTotal:           3880088 kB  
MemFree:            2366368 kB  
MemAvailable:       3141080 kB  
Buffers:             0 kB  
Cached:              841836 kB  
SwapCached:          7264 kB  
Active:              687940 kB  
Inactive:            181940 kB  
Active(anon):        93296 kB  
Inactive(anon):      113528 kB  
Active(file):        594644 kB  
Inactive(file):      68412 kB  
Unevictable:         120 kB  
Mlocked:             120 kB  
SwapTotal:            6160380 kB  
SwapFree:             6083580 kB  
Dirty:                404 kB  
Writeback:             0 kB  
AnonPages:            25520 kB
```

- You can list the contents of /proc using the *ls* command and view files using the *cat* command.
- For files that contain strings separated with null characters you have use the *strings* command to display the contents of such files correctly.
- Here is an example to look at the *environ* file for the *bash* process:

```
$ strings /proc/$$/environ
```

- Note: \$\$ in bash refers to the process ID of the current process, you can replace it with the actual PID of bash and test the above command.

# Processes and Threads

- The unit of dispatching is referred to as a *thread* or *lightweight process*
- The unit of resource ownership is referred to as a *process* or *task*
- **Multithreading** - The ability of an OS to support multiple, concurrent paths of execution within a single process

# Threads

- Each thread belongs to exactly one process and no thread can exist outside a process.
- Each thread represents a separate flow of control.
- Threads have been successfully used in implementing network servers and web server.
- They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors.

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

# Single Threaded Approaches

- A single thread of execution per process, in which the concept of a thread is not recognized, is referred to as a single-threaded approach
- MS-DOS is an example

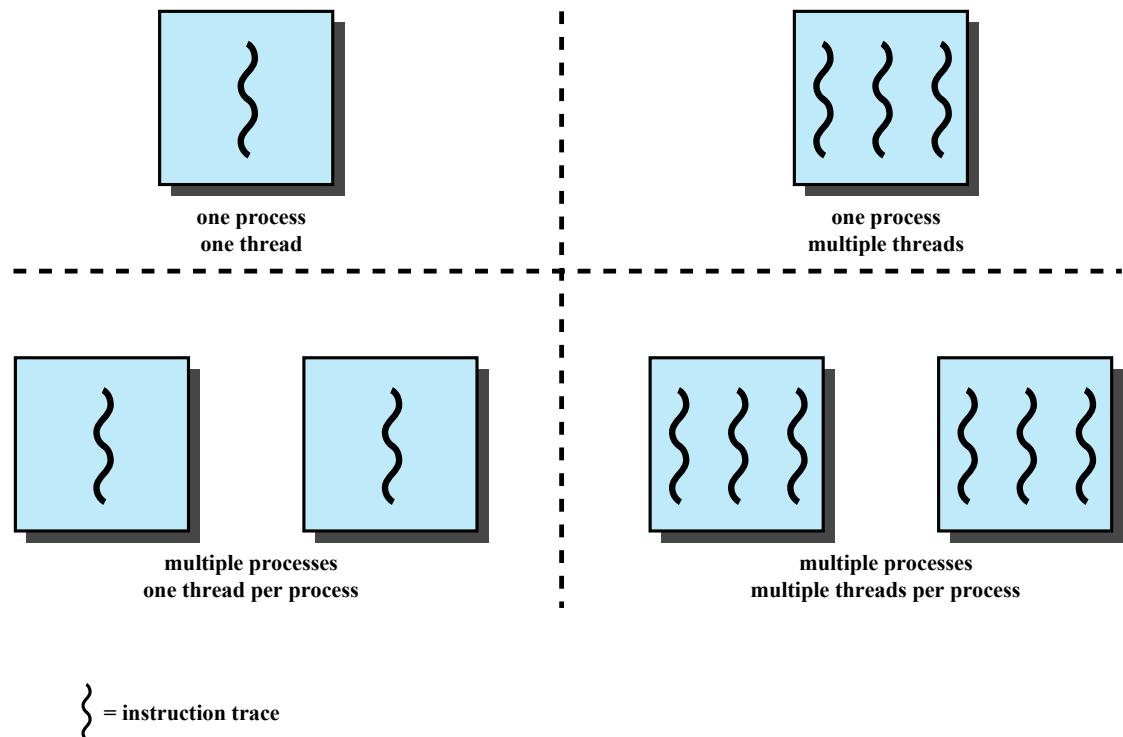


Figure 4.1 Threads and Processes

# Multithreaded Approaches

- The right half of the figure depicts multithreaded approaches
- A Java run-time environment is an example of a system of one process with multiple threads

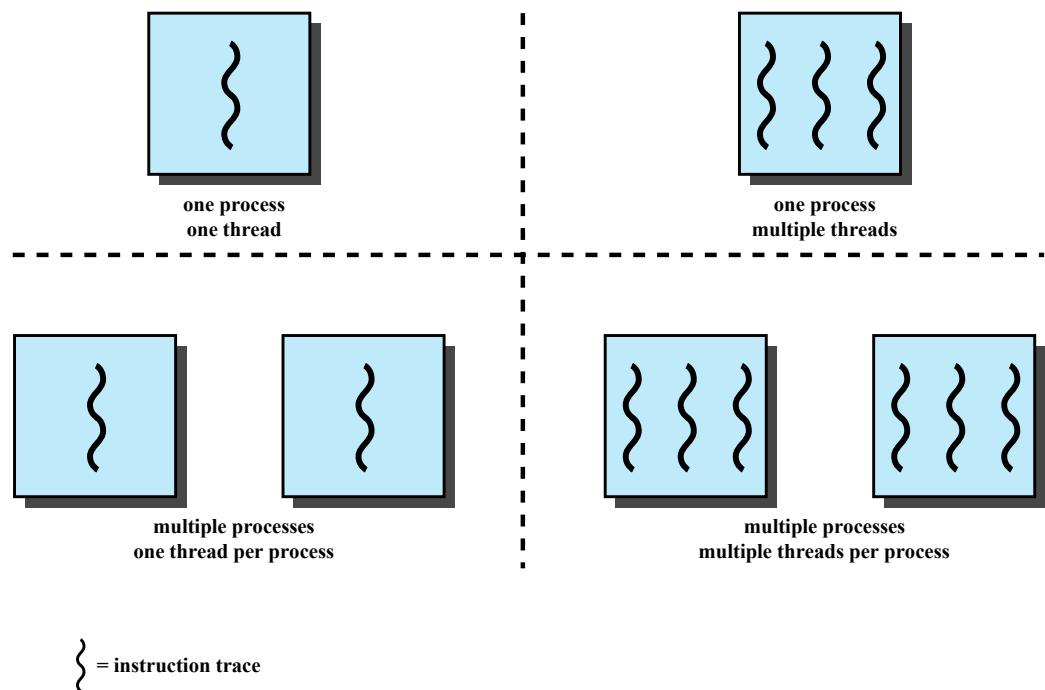
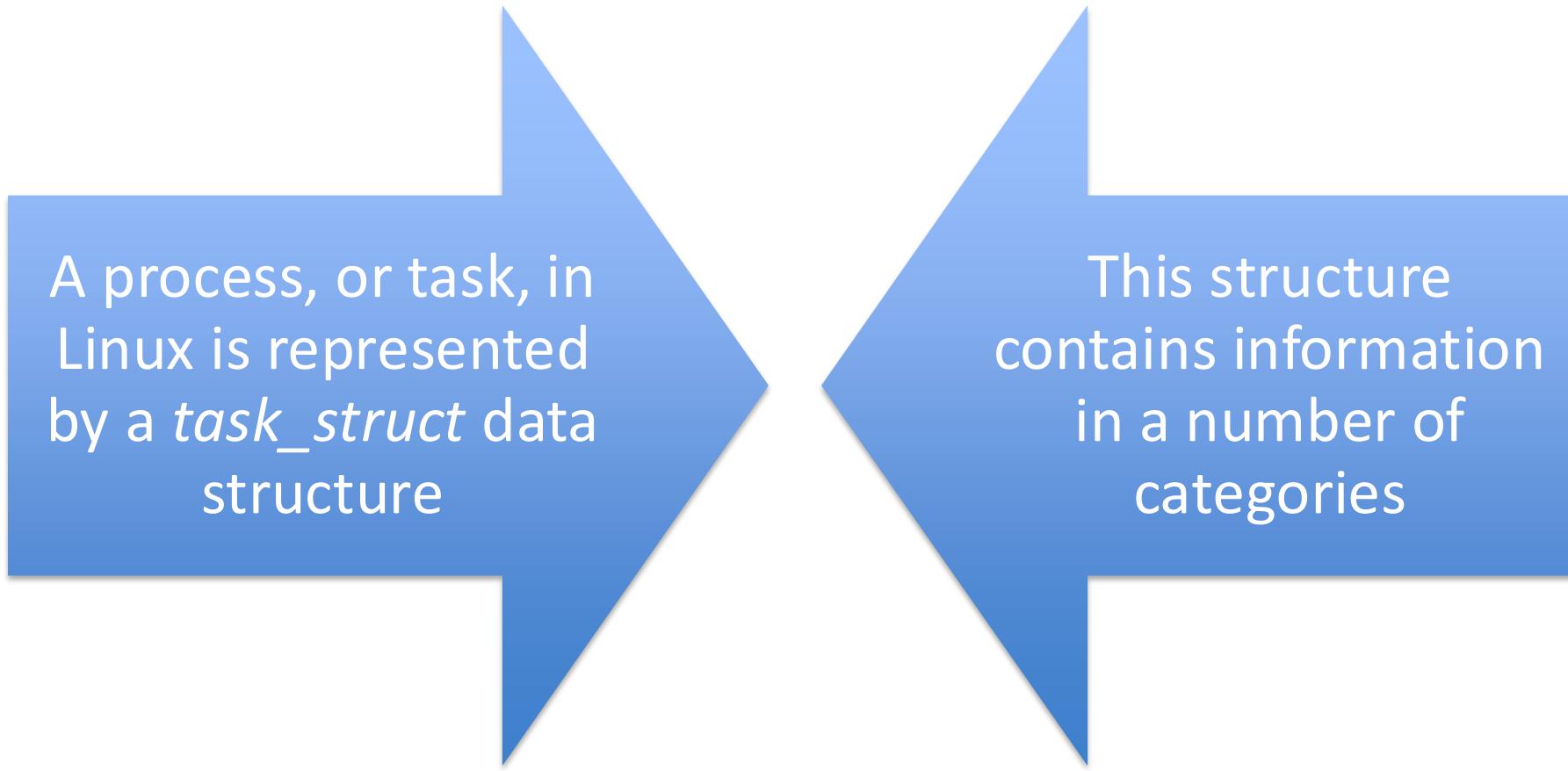


Figure 4.1 Threads and Processes

# Linux Tasks



A process, or task, in Linux is represented by a *task\_struct* data structure

This structure contains information in a number of categories

- State, • Scheduling information, • Identifiers, • Interprocess communication,
- Links, • Times and timers, • File system, • Address space, • Processor-specific context:

# Concurrency

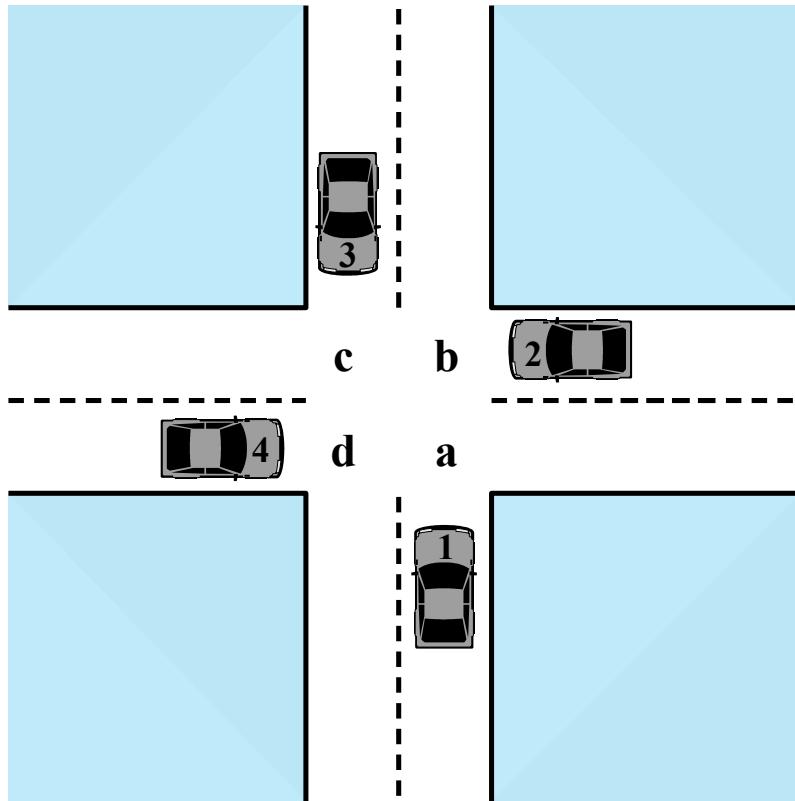
- Fundamental to all of these areas, and fundamental to OS design, is **concurrency**.
- Concurrency encompasses a host of design issues, including communication among processes, sharing of and competing for resources (such as memory, files, and I/O access), synchronization of the activities of multiple processes, and allocation of processor time to processes.
- We shall see that these issues arise not just in multiprocessing and distributed processing environments but even in single-processor multiprogramming systems.

# Race Condition

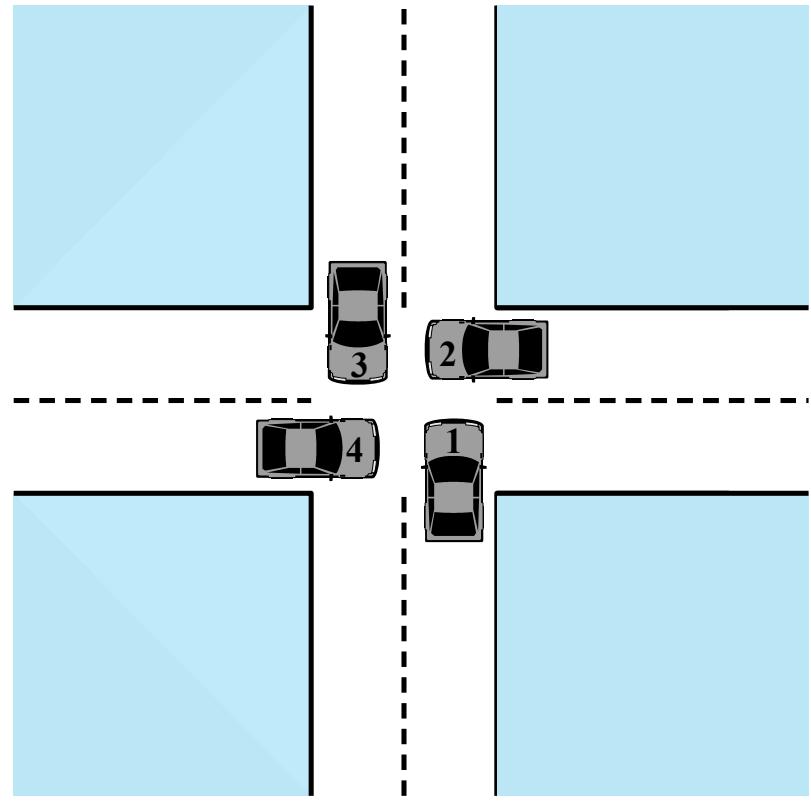
- Occurs when multiple processes or threads read and write data items
- The final result depends on the order of execution
  - The “loser” of the race is the process that updates last and will determine the final value of the variable

# Deadlock

- The *permanent blocking* of a set of processes that either compete for system resources or communicate with each other
- A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set
- Permanent because none of the events is ever triggered
- No efficient solution in the general case



(a) Deadlock possible



(b) Deadlock

**Figure 6.1 Illustration of Deadlock**

# Deadlock Approaches

- There is no single effective strategy that can deal with all types of deadlock
- Three approaches are common:
  - **Deadlock avoidance**
    - Do not grant a resource request if this allocation might lead to deadlock
  - **Deadlock prevention**
    - Disallow one of the three necessary conditions for deadlock occurrence, or prevent circular wait condition from happening
  - **Deadlock detection**
    - Grant resource requests when possible, but periodically check for the presence of deadlock and take action to recover

# UNIX Concurrency Mechanisms

- UNIX provides a variety of mechanisms for interprocessor communication and synchronization including:

Pipes

Messages

Shared  
memory

Semaphores

Signals

# Signals

- A software mechanism that informs a process of the occurrence of asynchronous events
  - Similar to a hardware interrupt, but does not employ priorities
- A signal is delivered by updating a field in the process table for the process to which the signal is being sent
- A process may respond to a signal by:
  - Performing some default action
  - Executing a signal-handler function
  - Ignoring the signal

# Signals

- Signals are software interrupts that provide a mechanism to deal with asynchronous events (e.g., a user pressing Control-C to interrupt a program that the shell is currently executing).
- A signal is a notification to a process that an event has occurred that requires the attention of the process (this typically interrupts the normal flow of execution of the interrupted process).
- Signals can also be used as a synchronization technique or even as a simple form of interprocess communication.
- Signals could be generated by hardware interrupts, the program itself, other programs, the OS kernel, or by the user.

- All these signals have a unique symbolic name and starts with SIG. The standard signals (also called POSIX reliable signals) are defined in the header file *<signal.h>*.
- Here are some examples:
- SIGINT : Interrupt a process from keyboard (e.g., pressing Control-C). The process is terminated.
- SIGQUIT : Interrupt a process to quit from keyboard (e.g., pressing Control-/>. The process is terminated and a core file is generated.
- SIGTSTP : Interrupt a process to stop from keyboard (e.g., pressing Control-Z). The process is stopped from executing.
- SIGUSR1 and SIGUSR2: These are user-defined signals, for use in application programs.

- **NOTE:** Please see Section 10.2 and Figure 10.1 in the textbook for a complete list of UNIX System signals. You can also find more details using *man 7 signal* on any CS UNIX system (*man signal* on Mac).
- After a signal is generated, it is delivered to a process to perform some action in response to this signal.
- Since signals are asynchronous events, a process has to decide ahead of time how to respond when the particular signal is delivered.
- There are three different options possible when a signal is delivered to a process:

- Perform the default action. Most signals have a default action associated with them, if the process does not change the default behavior then the default action specific to that particular signal will occur.
  - The predefined default signal is specified as `SIG_DFL`.
- Ignore the signal. If a signal is ignored, then the default action is performed.
  - The predefined ignore signal handler is specified as `SIG_IGN`.
- Catch and Handle the signal. It is also possible to override the default action and invoke a specific user-defined task when a signal is received.
  - This is usually performed by invoking a signal handler using `signal()` or `sigaction()` system calls.
- However, the signals `SIGKILL` and `SIGSTOP` cannot be caught, blocked, or ignored.

# signal()

- signal - ANSI C signal handling
- ```
typedef void (*sighandler_t) (int);
```
- ```
sighandler_t signal(int signum, sighandler_t handler);
```
- **signal()** sets the disposition of the signal *signum* to *handler*, which is either **SIG\_IGN**, **SIG\_DFL**, or the address of a programmer-defined function (a "signal handler").

# Sending Signals Using The Keyboard

- Ctrl-C to send an INT signal (SIGINT) to the running process.
  - This signal causes the process to immediately terminate.
- Ctrl-Z to send a TSTP signal (SIGTSTP) to the running process.
  - This signal causes the process to suspend execution.
- Ctrl-\ to send a ABRT signal (SIGABRT) to the running process.
  - This signal causes the process to immediately terminate.
  - Ctrl-\ doing the same as Ctrl-C but it gives us some better flexibility.

# infinite loop

```
1 #include<stdio.h>
2 #include<signal.h>
3 #include <unistd.h>
4
5 void handleSignINT(int sig)
6 {
7     printf("the signal caught = %d\n", sig);
8 }
9
10 int main()
11 {
12     signal(SIGINT, handleSignINT);
13     while (1==1)
14     {
15         printf("Hello CS332 \n");
16         sleep(1);
17     }
18     return 0;
19 }
20
```

# compile & run

```
[base] mahmutunan@MacBook-Pro lecture23 % gcc helloLoop.c -o helloLoop
[base] mahmutunan@MacBook-Pro lecture23 % ./helloLoop
Hello CS332
^Cthe signal caught =  2
Hello CS332
Hello CS332
^zsh: quit      ./helloLoop
```

- Now, let's write a program to handle a simple signal that catches either of the two user-defined signals and prints the signal number (see Figure 10.2 in the textbook).
- 1. Create a user-defined function (signal handler) that will be invoked when a signal is received:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <signal.h>
5
6 static void sig_usr(int signo) {
7     if (signo == SIGUSR1) {
8         printf("received SIGUSR1\n");
9     } else if (signo == SIGUSR2) {
10        printf("received SIGUSR2\n");
11    } else {
12        printf("received signal %d\n", signo);
13    }
14}
```

- 2. Now let's call above user-defined signal.

```
16 int main(void) {
17     if (signal(SIGUSR1, sig_usr) == SIG_ERR) {
18         printf("can't catch SIGUSR1\n");
19         exit(-1);
20     }
21     if (signal(SIGUSR2, sig_usr) == SIG_ERR) {
22         printf("can't catch SIGUSR2\n");
23         exit(-1);
24     }
25     for ( ; ; )
26         pause();
27
28     return 0;
29 }
```

```
[base] mahmutunan@MacBook-Pro lecture23 % gcc -Wall sigusr.c -o sigusr
[base] mahmutunan@MacBook-Pro lecture23 % ls
sigusr          sigusr.c
[base] mahmutunan@MacBook-Pro lecture23 % ./sigusr &
[1] 8122
[base] mahmutunan@MacBook-Pro lecture23 % jobs
[1] + running      ./sigusr
[base] mahmutunan@MacBook-Pro lecture23 % kill -USR1 %1
received SIGUSR1
[base] mahmutunan@MacBook-Pro lecture23 % kill -USR2 %1
received SIGUSR2
```

```
[base] mahmutunan@MacBook-Pro lecture23 % kill -SIGUSR1 8122
received SIGUSR1
[base] mahmutunan@MacBook-Pro lecture23 % kill -STOP 8122
[1] + suspended (signal) ./sigusr
[base] mahmutunan@MacBook-Pro lecture23 % jobs
[1] + suspended (signal) ./sigusr
[base] mahmutunan@MacBook-Pro lecture23 % kill -USR1 %1
received SIGUSR1
[base] mahmutunan@MacBook-Pro lecture23 % kill -USR2 %1
received SIGUSR2
[base] mahmutunan@MacBook-Pro lecture23 % kill -CONT 8122
[base] mahmutunan@MacBook-Pro lecture23 % jobs
[1] + running ./sigusr
[base] mahmutunan@MacBook-Pro lecture23 % kill -TERM %1
[1] + terminated ./sigusr
[base] mahmutunan@MacBook-Pro lecture23 % jobs
(base) mahmutunan@MacBook-Pro lecture23 %
```

- In the example we used the *kill* command that we used in the previous lecture to generate the signal.
- While we used the kill command in the previous lecture to terminate a process using the TERM signal (the default signal if no signal is specified), the kill command could be used to generate any other signal that is supported by the kernel.

- You can list all signals that can be generated using *kill -l*. For example, on the CS Linux systems you see the following output:

```
$ kill -l
1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL 5) SIGTRAP
6) SIGABRT 7) SIGBUS 8) SIGFPE 9) SIGKILL 10) SIGUSR1
11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO 30) SIGPWR
31) SIGSYS 34) SIGRTMIN 35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX
```

# Exercise 2

- We will now extend the exercise 1 to handle other signal generated from keyboard such as Control-C (SIGINT), Control-Z (SIGTSTP), and Control-\ (SIGQUIT). In this example we will use a single signal handler to handle all these signals using a switch statement (instead of separate signal handlers).

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <signal.h>
5
6 static void sig_usr(int signo) {
7     switch(signo) {
8         case SIGINT:
9             printf("received SIGINT signal %d\n", signo);
10            break;
11        case SIGQUIT:
12            printf("received SIGQUIT signal %d\n", signo);
13            break;
14        case SIGUSR1:
15            printf("received SIGUSR1 signal %d\n", signo);
16            break;
17        case SIGUSR2:
18            printf("received SIGUSR2 signal %d\n", signo);
19            break;
20        case SIGTSTP:
21            printf("received SIGTSTP signal %d\n", signo);
22            break;
23        default:
24            printf("received signal %d\n", signo);
25    }
26 }
```

```
28 int main(void) {
29     if (signal(SIGINT, sig_usr) == SIG_ERR) {
30         printf("can't catch SIGINT\n");
31         exit(-1);
32     }
33     if (signal(SIGQUIT, sig_usr) == SIG_ERR) {
34         printf("can't catch SIGQUIT\n");
35         exit(-1);
36     }
37     if (signal(SIGUSR1, sig_usr) == SIG_ERR) {
38         printf("can't catch SIGUSR1\n");
39         exit(-1);
40     }
41     if (signal(SIGUSR2, sig_usr) == SIG_ERR) {
42         printf("can't catch SIGUSR2\n");
43         exit(-1);
44     }
45     if (signal(SIGTSTP, sig_usr) == SIG_ERR) {
46         printf("can't catch SIGTSTP\n");
47         exit(-1);
48     }
49     for ( ; ; )
50         pause();
51
52     return 0;
53 }
```

# compile & run

```
(base) mahmutunan@MacBook-Pro ~ % cd Desktop/lecture23
(base) mahmutunan@MacBook-Pro lecture23 % ls
sighandler      sighandler.c    sigusr          sigusr.c
(base) mahmutunan@MacBook-Pro lecture23 % gcc -Wall sighandler.c -o sighandler
(base) mahmutunan@MacBook-Pro lecture23 % ./sighandler
^Creceived SIGINT signal 2
^Zreceived SIGTSTP signal 18
^Zreceived SIGTSTP signal 18
^\\received SIGQUIT signal 3
^Zreceived SIGTSTP signal 18
^Creceived SIGINT signal 2
^Creceived SIGINT signal 2
^Zreceived SIGTSTP signal 18
^\\received SIGQUIT signal 3
-
```

# infinite loop

- Notice that we have replaced the default signal handlers to kill this job from the keyboard and the program is in an infinite loop with pause. As a result we cannot terminate this program from the keyboard. We have to login to the specific machine this process is running and kill this process using either *kill* or *top* commands. You can follow the examples from the first part and kill this process.

# Default Action Of Signals

- Each signal has a default action, one of the following:
  - **Term:** The process will terminate.
  - **Core:** The process will terminate and produce a core dump file.
  - **Ign:** The process will ignore the signal.
  - **Stop:** The process will stop.
  - **Cont:** The process will continue from being stopped.
- Default action may be changed using handler function. Some signal's default action cannot be changed. **SIGKILL** and **SIGABRT** signal's default action cannot be changed or ignored
- [https://linuxhint.com/signal\\_handlers\\_c\\_programming\\_language/](https://linuxhint.com/signal_handlers_c_programming_language/)

# Basic Signal handler examples

```
#include<stdio.h>
#include<signal.h>
#include<unistd.h>
void sig_handler(int signum){

    //Return type of the handler function should be void
    printf("\nInside handler function\n");
}

int main(){
    signal(SIGINT,sig_handler); // Register signal handler
    for(int i=1;;i++){        //Infinite loop
        printf("%d : Inside main function\n",i);
        sleep(1); // Delay for 1 second
    }
    return 0;
}
```

```
tump@tump:~/Desktop/c_prog/signal$ gcc Example1.c -o Example1
tump@tump:~/Desktop/c_prog/signal$ ./Example1
1 : Inside main function
2 : Inside main function
^C
Inside handler function
3 : Inside main function
4 : Inside main function
^CQuit (core dumped)
tump@tump:~/Desktop/c_prog/signal$
```

```
#include<stdio.h>
#include<unistd.h>
#include<signal.h>
int main(){
    signal(SIGINT,SIG_IGN); // Register signal handler for ignoring the signal
    for(int i=1;;i++){    //Infinite loop
        printf("%d : Inside main function\n",i);
        sleep(1); // Delay for 1 second
    }
    return 0;
}
```

```
tump@tump:~/Desktop/c_prog/signal$ gcc Example2.c -o Example2
tump@tump:~/Desktop/c_prog/signal$ ./Example2
1 : Inside main function
2 : Inside main function
^C3 : Inside main function
4 : Inside main function
^C5 : Inside main function
^QQuit (core dumped)
tump@tump:~/Desktop/c_prog/signal$
```

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

void sig_handler(int signum){
    printf("\nInside handler function\n");
    signal(SIGINT,SIG_DFL); // Re Register signal handler for default action
}

int main(){
    signal(SIGINT,sig_handler); // Register signal handler
    for(int i=1;;i++){ //Infinite loop
        printf("%d : Inside main function\n",i);
        sleep(1); // Delay for 1 second
    }
    return 0;
}
```

```
tump@tump:~/Desktop/c_prog/signal$ gcc Example3.c -o Example3
tump@tump:~/Desktop/c_prog/signal$ ./Example3
1 : Inside main function
2 : Inside main function
3 : Inside main function
^C
Inside handler function
4 : Inside main function
^C
tump@tump:~/Desktop/c_prog/signal$
```

```
#include<stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include<signal.h>
void sig_handler_parent(int signum){
    printf("Parent : Received a response signal from child \n");
}

void sig_handler_child(int signum){
    printf("Child : Received a signal from parent \n");
    sleep(1);
    kill(getppid(),SIGUSR1);
}

int main(){
    pid_t pid;
    if((pid=fork())<0){
        printf("Fork Failed\n");
        exit(1);
    }
    /* Child Process */
    else if(pid==0){
        signal(SIGUSR1,sig_handler_child); // Register signal handler
        printf("Child: waiting for signal\n");
        pause();
    }
    /* Parent Process */
    else{
        signal(SIGUSR1,sig_handler_parent); // Register signal handler
        sleep(1);
        printf("Parent: sending signal to Child\n");
        kill(pid,SIGUSR1);
        printf("Parent: waiting for response\n");
        pause();
    }
    return 0;
}
```

```

#include<stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include<signal.h>
void sig_handler_parent(int signum){
    printf("Parent : Received a response signal from child \n");
}

void sig_handler_child(int signum){
    printf("Child : Received a signal from parent \n");
    sleep(1);
    kill(getppid(),SIGUSR1);
}

int main(){
    pid_t pid;
    if(tump@tump:~/Desktop/c_prog/signals$ gcc Example6.c -o Example6
        tump@tump:~/Desktop/c_prog/signals$ ./Example6
    } Child: waiting for signal
    /* Parent: sending signal to Child
    else Parent: waiting for response
        Child : Received a signal from parent
    } Parent : Received a response signal from child
    /* tump@tump:~/Desktop/c_prog/signals$
        signal(SIGUSR1,sig_handler_parent); // Register signal handler
        sleep(1);
        printf("Parent: sending signal to Child\n");
        kill(pid,SIGUSR1);
        printf("Parent: waiting for response\n");
        pause();
    }
    return 0;
}

```

# Exercise Sigint

- Till now we used the kill command and the keyboard to generate the signals.
- Now we will generate the signal in a program using the C API for *kill* or *raise* system call.
- In this example, we replace the SIGINT signal handler with our own and in the signal handler asks the user if the process should be terminated and then based on the response continue with either terminating the process or let it continue to run.
- We use read function instead of scanf to emphasize that scanf is not a reentrant function

# sigint

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <signal.h>
5
6 static void sig_int(int signo) {
7     ssize_t n;
8     char buf[2];
9
10    signal(signo, SIG_IGN); /* ignore signal first */
11    printf("Do you really want to do this: [Y/N]? ");
12    fflush(stdout);
13    n = read(STDIN_FILENO, buf, 2);
14    if (buf[0] == 'Y') {
15        raise(SIGTERM); // or kill(0, SIGTERM); // or exit (-1);
16    } else {
17        printf("Ignoring signal, be careful next time!\n");
18        fflush(stdout);
19    }
20    signal(signo, sig_int); /* reinstall the signal handler */
21 }
```

```
23 int main(void) {
24     if (signal(SIGINT, sig_int) == SIG_ERR) {
25         printf("Unable to catch SIGINT\n");
26         exit(-1);
27     }
28     for ( ; ; )
29         pause();
30
31     return 0;
32 }
33
```

```
[base] mahmutunan@MacBook-Pro lecture23 % gcc -Wall sigint.c -o sigint
[base] mahmutunan@MacBook-Pro lecture23 % ./sigint
^CDo you really want to do this: [Y/N]? N
Ignoring signal, be careful next time!
^CDo you really want to do this: [Y/N]?
Ignoring signal, be careful next time!
^CDo you really want to do this: [Y/N]? Y
zsh: terminated  ./sigint
[base] mahmutunan@MacBook-Pro lecture23 %
```

# recall -forkexecvp.c

```
int main(int argc, char **argv) {
    pid_t pid;
    int status;

    if (argc < 2) {
        printf("Usage: %s <command> [args]\n", argv[0]);
        exit(-1);
    }

    pid = fork();
    if (pid == 0) { /* this is child process */
        execvp(argv[1], &argv[1]);
        printf("If you see this statement then execl failed ;-(\n");
        perror("execvp");
        exit(-1);
    } else if (pid > 0) { /* this is the parent process */
        printf("Wait for the child process to terminate\n");
        wait(&status); /* wait for the child process to terminate */
        if (WIFEXITED(status)) { /* child process terminated normally */
            printf("Child process exited with status = %d\n", WEXITSTATUS(status));
        } else { /* child process did not terminate normally */
            printf("Child process did not terminate normally!\n");
            /* look at the man page for wait (man 2 wait) to determine
               how the child process was terminated */
        }
    } else { /* we have an error */
        perror("fork"); /* use perror to print the system error message */
        exit(EXIT_FAILURE);
    }

    printf("[%ld]: Exiting program ....\n", (long)getpid());

    return 0;
}
```

- Let us now consider how signals impact child processes created using fork/exec. We will be running a c program that consist of fork and execvp system calls (hw1.c)
- This code illustrates the use of dynamic memory allocation to create contiguous 2D-matrices and use traditional array indexing. It also illustrate the use of gettimeofday to measure wall clock time.

```
$ ./a.out /home/UAB/puri/CS332/shared/hw1 1000
```

```
Wait for the child process to terminate
```

```
^C
```

```
$ ps -u
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
puri	19202	0.3	0.1	125440	3912	pts/0	Ss	10:05	0:00	-bash
puri	19320	0.0	0.0	161588	1868	pts/0	R+	10:06	0:00	ps -u

```
$
```

```
$ ./a.out /home/UAB/puri/CS332/shared/hw1 1000
Wait for the child process to terminate
^Z
[1]+  Stopped                  ./a.out /home/UAB/puri/CS332/shared/hw1 1000
$
$ jobs
[1]+  Stopped                  ./a.out /home/UAB/puri/CS332/shared/hw1 1000
$ ps -u
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
puri    19202  0.1  0.1 125440  3912 pts/0    Ss   10:05  0:00 -bash
puri    19351  0.0  0.0   4220   352 pts/0    T    10:08  0:00 ./a.out /home/U
puri    19352 29.0  0.6  27800 23844 pts/0    T    10:08  0:01 /home/UAB/puri/
puri    19353  0.0  0.0 161588  1864 pts/0    R+   10:08  0:00 ps -u
$ kill -CONT 19352
$
$ ps -u
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
puri    19202  0.0  0.1 125440  3912 pts/0    Ss   10:05  0:00 -bash
puri    19351  0.0  0.0   4220   352 pts/0    T    10:08  0:00 ./a.out /home/U
puri    19352 17.9  0.6  27800 23844 pts/0    R    10:08  0:04 /home/UAB/puri/
puri    19355  0.0  0.0 161588  1872 pts/0    R+   10:08  0:00 ps -u
$ Time taken for size 1000 = 32.274239 seconds
$ jobs
[1]+  Stopped                  ./a.out /home/UAB/puri/CS332/shared/hw1 1000
$ kill -CONT 19351
$ Child process exited with status = 0
[19351]: Exiting program .....
[1]+  Done                    ./a.out /home/UAB/puri/CS332/shared/hw1 1000
$ jobs
$
```

```
$ ./a.out /home/UAB/puri/CS332/shared/hw1 1000
Wait for the child process to terminate
^Z
[1]+  Stopped                  ./a.out /home/UAB/puri/CS332/shared/hw1 1000
$
$ jobs
[1]+  Stopped                  ./a.out /home/UAB/puri/CS332/shared/hw1 1000
$ ps -u
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
puri    19202  0.1  0.1 125440  3912 pts/0    Ss  10:05  0:00 -bash
puri    19351  0.0  0.0   4220   352 pts/0    T   10:08  0:00 ./a.out /home/U
puri    19352 29.0  0.6  27800 23844 pts/0    T   10:08  0:01 /home/UAB/puri/
puri    19353  0.0  0.0 161588  1864 pts/0    R+  10:08  0:00 ps -u
$ kill -CONT 19352
$
$ ps -u
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
puri    19202  0.0  0.1 125440  3912 pts/0    Ss  10:05  0:00 -bash
puri    19351  0.0  0.0   4220   352 pts/0    T   10:08  0:00 ./a.out /home/U
puri    19352 17.9  0.6  27800 23844 pts/0    R   10:08  0:04 /home/UAB/puri/
puri    19355  0.0  0.0 161588  1872 pts/0    R+  10:08  0:00 ps -u
$ Time taken for size 1000 = 32.274239 seconds
$ jobs
[1]+  Stopped                  ./a.out /home/UAB/puri/CS332/shared/hw1 1000
$ kill -CONT 19351
$ Child process exited with status = 0
[19351]: Exiting program .....
[1]+  Done                    ./a.out /home/UAB/puri/CS332/shared/hw1 1000
$ jobs
$
```

# Linux I/O Streams

- Before we discuss I/O redirection, let us review how I/O is handled in Linux systems.
- Each process in a Linux environment has three different file descriptors available when a process is created: standard input (*stdin* – 0), standard output (*stdout* – 1), and standard error (*stderr* – 2).
- These three file descriptors are created when a process is created.
- We use the *stdin* file descriptor to read input from a keyboard or from another a file or from another program.
- Similarly, we use the *stdout* and *stderr* file descriptors to write output and error messages, respectively, to the terminal.

- Input and output in the Linux environment is distributed across three streams. These streams are:
    - **standard input (stdin)**
    - **standard output (stdout)**
    - **standard error (stderr)**
  - The streams are also numbered:
    - **stdin (0)**
    - **stdout (1)**
    - **stderr (2)**
- During standard interactions between the user and the terminal, standard input is transmitted through the user's keyboard. Standard output and standard error are displayed on the user's terminal as text. Collectively, the three streams are referred to as the *standard streams*.

# cat command

```
[base] mahmutunan@MacBook-Pro lecture26 % cat  
1  
1  
2  
2  
3  
3  
(base) mahmutunan@MacBook-Pro lecture26 % _
```

# Stream Redirection

- **Overwrite**

- > - standard output

- < - standard input

- 2>** - standard error

- **Append**

- >> - standard output

- << - standard input

- 2>>** - standard error

```
[base] mahmutunan@MacBook-Pro lecture26 % cat > outputFile.txt  
1  
2  
3  
4  
[base] mahmutunan@MacBook-Pro lecture26 % cat outputFile.txt  
1  
2  
3  
4
```

```
[base] mahmutunan@MacBook-Pro lecture26 % cat >> outputFile.txt  
a  
b  
c  
[base] mahmutunan@MacBook-Pro lecture26 % cat outputFile.txt  
1  
2  
3  
4  
a  
b  
c
```

```
[base] mahmutunan@MacBook-Pro lecture26 % ls >> listOffiles.txt  
[base] mahmutunan@MacBook-Pro lecture26 % cat listOffiles.txt  
error.txt  
forkexecvp  
forkexecvp.c  
forkexecvp2.c  
hw2.c  
input.txt  
ioredirect.c  
lab7_solution.c  
listOffiles.txt  
myprog  
myprog.c  
output.txt  
output2.txt  
outputFile.txt
```

# stdout stderr

```
(base) mahmutunan@MacBook-Pro lecture26 % echo "some output using stdout"  
some output using stdout  
(base) mahmutunan@MacBook-Pro lecture26 % echo  
  
(base) mahmutunan@MacBook-Pro lecture26 % _
```

```
(base) mahmutunan@MacBook-Pro lecture26 % cat nonexistingfile.txt  
cat: nonexistingfile.txt: No such file or directory  
(base) mahmutunan@MacBook-Pro lecture26 % _
```

- You have already been using these file descriptors when you wrote the insertion sort program in C.
- You read the number of elements and the elements to be sorted from the keyboard using the *scanf* function.
- The *scanf* function was using *stdin* file descriptor to read your keyboard input. In other words, the following two functions are equivalent:

```
scanf ("%d", &N) ;  
fscanf(stdin, "%d", &N) ;
```

- Similarly when you use the *printf* function to print the output of your program you are using the *stdout* file descriptor.

- The two functions below are equivalent:

```
printf ("%d\n", N);  
fprintf (stdout, "%d\n", N);
```

- The file descriptors *stdin*, *stdout*, and *stderr* are defined in the header file *stdio.h*. We typically use the *stderr* file descriptor to write error messages.

- If we need to save the output or error message from a program to a file or read data from a file instead of entering it through the keyboard, we can use the I/O redirection supported by the Linux shell (such as bash).
- In fact, we already used this in one of the earlier labs when we used insertion sort to sort large input values.
- The following examples show how to use I/O redirection in the bash shell to read input from a file (instead of entering it from the keyboard) and send the output and error messages to different files (instead of the terminal)

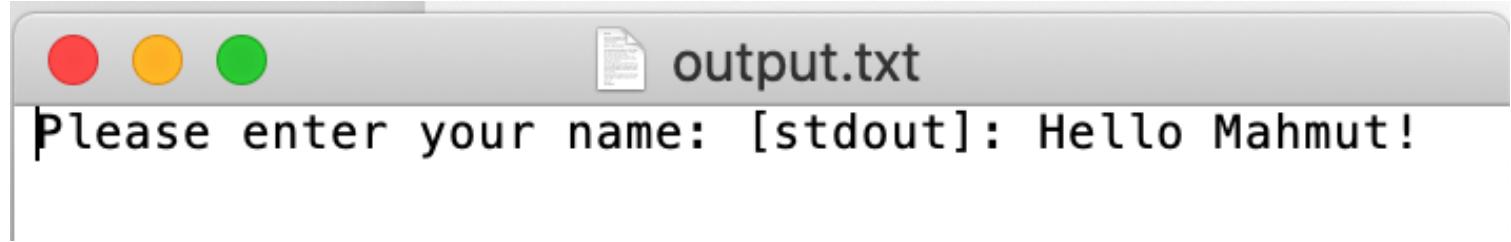
# Exercise 1

```
1 #include <stdio.h>
2
3 int main(int argc, char **argv) {
4     char name[BUFSIZ];
5
6     printf("Please enter your name: ");
7     scanf("%s", name);
8     printf("[stdout]: Hello %s!\n", name);
9     fprintf(stderr, "[stderr]: Hello %s!\n", name);
10
11    return 0;
12 }
```

Compile the program [myprog.c](#),  
create a file called *input.txt*, type you name in the file *input.txt*

# compile & run

```
(base) mahmutunan@MacBook-Pro lecture26 % touch input.txt
(base) mahmutunan@MacBook-Pro lecture26 % echo "Mahmut" > input.txt
(base) mahmutunan@MacBook-Pro lecture26 % cat input.txt
Mahmut
(base) mahmutunan@MacBook-Pro lecture26 % gcc -Wall myprog.c -o myprog
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt
Please enter your name: [stdout]: Hello Mahmut!
[stderr]: Hello Mahmut!
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt > output.txt
[stderr]: Hello Mahmut!
(base) mahmutunan@MacBook-Pro lecture26 %
```



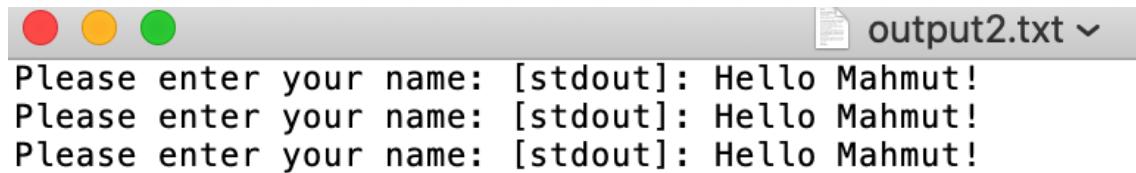
# compile & run

```
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt 2> error.txt
Please enter your name: [stdout]: Hello Mahmut!
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt 2> error.txt >output.txt
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt >output2.txt 2> error.txt
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt & >output2.txt 2> error.txt
[1] 91593
Please enter your name: [stdout]: Hello Mahmut!
[stderr]: Hello Mahmut!
[1] + done      ./myprog < input.txt
^C
(base) mahmutunan@MacBook-Pro lecture26 %
```



# compile & run

```
[base] mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt >> output2.txt
[stderr]: Hello Mahmut!
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt >> output2.txt
[stderr]: Hello Mahmut!
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt >> output2.txt
[stderr]: Hello Mahmut!
(base) mahmutunan@MacBook-Pro lecture26 %
```



You can replace `>` with `>>` if you like to append to the file instead of overwriting the file

# Sharing between parent and child processes

- When we created a new process using fork/exec in the previous labs, we noted that the child process is a copy of the parent process and it inherits several attributes from the parent process such as open file descriptors.
- This duplication of descriptors allowed the child processes to read and write to the standard I/O streams (note that the child process was able to read input from the keyboard and write output to the terminal).
- As a result of this sharing both parent and child processes share the three standard I/O streams: *stdin*, *stdout*, and *stderr*.

# Exercise 2

- Let us use the example from the previous lectures to illustrate this by adding the following lines in the parent process:

```
char name[BUFSIZ];  
  
printf("Please enter your name: ");  
scanf("%s", name);  
printf("[stdout]: Hello %s!\n", name);  
fprintf(stderr, "[stderr]: Hello  
%s!\n", name);
```

# Exercise 2

```
1 |  
2 | #include <stdio.h>  
3 | #include <stdlib.h>  
4 | #include <unistd.h>  
5 | #include <sys/types.h>  
6 | #include <sys/wait.h>  
7 |  
8 | int main(int argc, char **argv) {  
9 |     pid_t pid;  
10 |    int status;  
11 |  
12 |    if (argc < 2) {  
13 |        printf("Usage: %s <command> [args]\n", argv[0]);  
14 |        exit(-1);  
15 |    }  
16 |  
17 |    pid = fork();  
18 |    if (pid == 0) { /* this is child process */  
19 |        execvp(argv[1], &argv[1]);  
20 |        perror("exec");  
21 |        exit(-1);  
22 |    } else if (pid > 0) { /* this is the parent process */
```

# Exercise 2

```
23     char name[BUFSIZ];  
24  
25     printf("[%d]: Please enter your name: ", getpid());  
26     scanf("%s", name);  
27     printf("[stdout]: Hello %s!\n", name);  
28     fprintf(stderr, "[stderr]: Hello %s!\n", name);  
29  
30     wait(&status); /* wait for the child process to terminate */  
31     if (WIFEXITED(status)) { /* child process terminated normally */  
32         printf("Child process exited with status = %d\n", WEXITSTATUS(status));  
33     } else { /* child process did not terminate normally */  
34         printf("Child process did not terminate normally!\n");  
35         /* look at the man page for wait (man 2 wait) to determine  
36             how the child process was terminated */  
37     }  
38 } else { /* we have an error */  
39     perror("fork"); /* use perror to print the system error message */  
40     exit(EXIT_FAILURE);  
41 }  
42  
43     return 0;  
44 }  
45 }
```

# compile & run

- If we compile and execute the program by using *myprog* (used earlier) as the child process, we will notice that the prompt to enter the name is printed twice. If we enter the name, which process is reading the keyboard input?

```
[base] mahmutunan@MacBook-Pro lecture26 % gcc -Wall forkexecvp2.c -o forkexecvp
[base] mahmutunan@MacBook-Pro lecture26 % ./forkexecvp ./myprog
[86530]: Please enter your name: Please enter your name: mahmut
[stdout]: Hello mahmut!
[stderr]: Hello mahmut!
```

# Exercise 2

```
23     char name[BUFSIZ];
24
25     printf("[%d]: Please enter your name: ", getpid());
26     scanf("%s", name);
27     printf("[%d-stdout]: Hello %s!\n", getpid(), name);
28     fprintf(stderr, "[%d-stderr]: Hello %s!\n", getpid(), name);
29
30     wait(&status); /* wait for the child process to terminate */
31     if (WIFEXITED(status)) { /* child process terminated normally */
32         printf("Child process exited with status = %d\n", WEXITSTATUS(status));
33     } else { /* child process did not terminate normally */
34         printf("Child process did not terminate normally!\n");
35         /* look at the man page for wait (man 2 wait) to determine
36            how the child process was terminated */
37     }
38 } else { /* we have an error */
39     perror("fork"); /* use perror to print the system error message */
40     exit(EXIT_FAILURE);
41 }
42
43     return 0;
44 }
```

# compile & run

- We could add the PID in the printf statement to make this explicit. We can update the code above to print the PID and test the program. In any case, this illustrates the result of sharing of the standard I/O streams between the parent and the child processes

```
[base] mahmutunan@MacBook-Pro lecture26 % gcc -Wall forkexecvp2.c -o forkexecvp  
[base] mahmutunan@MacBook-Pro lecture26 % ./forkexecvp ./myprog  
[85953]: Please enter your name: Please enter your name: mahmut  
[85953-stdout]: Hello mahmut!  
[85953-stderr]: Hello mahmut!
```

- If we like to change the behavior of all child processes to use separate files instead of the standard I/O streams we have to replace the standard I/O file descriptors with new file descriptors.
- We will use the dup2() system call to create a copy of this file descriptors and associate separate files to replace the standard I/O streams.

# dup2()

- **Copying file descriptors – dup2() system call**
- The dup2() system call duplicates an existing file descriptor and returns the duplicate file descriptor.
- After the call returns successfully, both file descriptors can be used interchangeably. If there is an error then -1 is returned and the corresponding errno is set (look at the man page for dup2() for more details on the specific error codes returned).
- The prototype for the dup2() system call is shown below:

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

```
int dup2(int oldfd, int newfd);
```

# dup() vs dup2()

- These system calls create a copy of the file descriptor *oldfd*.
- **dup()** uses the lowest-numbered unused descriptor for the new descriptor.
- **dup2()** makes *newfd* be the copy of *oldfd*, closing *newfd* first if necessary, but note the following:
  - If *oldfd* is not a valid file descriptor, then the call fails, and *newfd* is not closed.
  - If *oldfd* is a valid file descriptor, and *newfd* has the same value as *oldfd*, then **dup2()** does nothing, and returns *newfd*.
- After a successful return from one of these system calls, the old and new file descriptors may be used interchangeably. T

# Exercise

- This example illustrate how to use dup2 to replace the stdin and stdout file descriptors with the files stdin.txt and stdout.txt, respectively.
- We use the file forkexecvp.c from the previous lab and the new lines of code are highlighted.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/types.h>
5 #include <sys/wait.h>
6 #include <sys/stat.h>
7 #include <fcntl.h>
8
9 int main(int argc, char **argv) {
10     pid_t pid;
11     int status;
12     int fdin, fdout;
13
14     /* display program usage if arguments are missing */
15     if (argc < 2) {
16         printf("Usage: %s <command> [args]\n", argv[0]);
17         exit(-1);
18     }
19
20     /* open file to read standard input stream,
21      make sure the file stdin.txt exists, even if it is empty */
22     if ((fdin = open("stdin.txt", O_RDONLY)) == -1) {
23         printf("Error opening file stdin.txt for input\n");
24         exit(-1);
25     }
```

```
26
27     /* open file to write standard output stream in append mode.
28      create a new file if the file does not exist. */
29     if ((fdout = open("stdout.txt", O_CREAT | O_APPEND | O_WRONLY, 0755)) == -1) {
30         printf("Error opening file stdout.txt for output\n");
31         exit(-1);
32     }
33
34     pid = fork();
35     if (pid == 0) { /* this is child process */
36         /* replace standard input stream with the file stdin.txt */
37         dup2(fdin, 0);
38
39         /* replace standard output stream with the file stdout.txt */
40         dup2(fdout, 1);
41
42         execvp(argv[1], &argv[1]);
43         /* since stdout is written to stdout.txt and not the terminal,
44            we should write to stderr in case exec fails, we use perror
45            that writes the error message to stderr */
46         perror("exec");
47         exit(-1);
```

```
48 } else if (pid > 0) { /* this is the parent process */
49     /* output from the parent process still goes to stdout :-) */
50     printf("Wait for the child process to terminate\n");
51     wait(&status); /* wait for the child process to terminate */
52     if (WIFEXITED(status)) { /* child process terminated normally */
53         printf("Child process exited with status = %d\n", WEXITSTATUS(status));
54         /* parent process still has the file handle to stdout.txt,
55            now that the child process is done, let us write to
56            the file stdout.txt using the write system call */
57         write(fdout, "Hey! This is the parent process\n", 32);
58         close(fdout);
59         /* since we opened the file in append mode, the above text
60            will be added after the output from the child process */
61     } else { /* child process did not terminate normally */
62         printf("Child process did not terminate normally!\n");
63         /* look at the man page for wait (man 2 wait) to determine
64            how the child process was terminated */
65     }
66 } else { /* we have an error */
67     perror("fork"); /* use perror to print the system error message */
68     exit(EXIT_FAILURE);
69 }
70
71     return 0;
72 }
```

# compile & run

- Compile and run this program using `myprog` as the child process. Note that you have provide input to `myprog` in the file `stdin.txt` and the output of `myprog` will be written to `stdout.txt`. As we did not do anything with the `stderr` stream, the output to `stderr` stream goes to the terminal. You can add the PID in the print statement to confirm which output is sent to the terminal and which output is sent to the files. Here is a terminal session that illustrates this interaction:

# recall myprog.c

```
1 #include <stdio.h>
2
3 int main(int argc, char **argv) {
4     char name[BUFSIZ];
5
6     printf("Please enter your name: ");
7     scanf("%s", name);
8     printf("[stdout]: Hello %s!\n", name);
9     fprintf(stderr, "[stderr]: Hello %s!\n", name);
10
11    return 0;
12 }
```

# compile & run

```
[base] mahmutunan@MacBook-Pro lecture27 % gcc -Wall -o myprog myprog.c
[base] mahmutunan@MacBook-Pro lecture27 % gcc -Wall -o ioredirect ioredirect.c
[base] mahmutunan@MacBook-Pro lecture27 % cat > stdin.txt
Mahmut
[base] mahmutunan@MacBook-Pro lecture27 % ./ioredirect ./myprog
Wait for the child process to terminate
[stderr]: Hello Mahmut!
Child process exited with status = 0
[base] mahmutunan@MacBook-Pro lecture27 % cat stdout.txt
Please enter your name: [stdout]: Hello Mahmut!
Hey! This is the parent process
```

```
(base) mahmutunan@MacBook-Pro lecture27 % cat > stdin.txt  
World  
(base) mahmutunan@MacBook-Pro lecture27 % ./ioreirect ./myprog  
Wait for the child process to terminate  
[stderr]: Hello World!  
Child process exited with status = 0  
(base) mahmutunan@MacBook-Pro lecture27 % cat stdout.txt  
Please enter your name: [stdout]: Hello Mahmut!  
Hey! This is the parent process  
Please enter your name: [stdout]: Hello World!  
Hey! This is the parent process  
(base) mahmutunan@MacBook-Pro lecture27 %
```

```
[base) mahmutunan@MacBook-Pro lecture27 % ./ioreirect uname -a  
Wait for the child process to terminate  
Child process exited with status = 0  
[base) mahmutunan@MacBook-Pro lecture27 % cat stdout.txt  
Please enter your name: [stdout]: Hello Mahmut!  
Hey! This is the parent process  
Please enter your name: [stdout]: Hello World!  
Hey! This is the parent process  
Darwin MacBook-Pro.local 19.6.0 Darwin Kernel Version 19.6.0: Mon Aug 31 22:12:52 PDT 2020; roo  
t:xnu-6153.141.2~1/RELEASE_X86_64 x86_64  
Hey! This is the parent process  
(base) mahmutunan@MacBook-Pro lecture27 %
```

# Pipes

- We have seen the Linux shell support pipes.
- For example:

```
$ ps -elf | grep ssh
```

- The above example redirects the output of the program ps to another program grep (instead of sending the output to standard output).
- Similarly, the program grep uses the output of ps as the input instead of a file name as the argument. The shell implements this redirection using pipes.
- The system call pipe is used to create a pipe and in most Linux systems pipes provide a unidirectional flow of data between two processes.

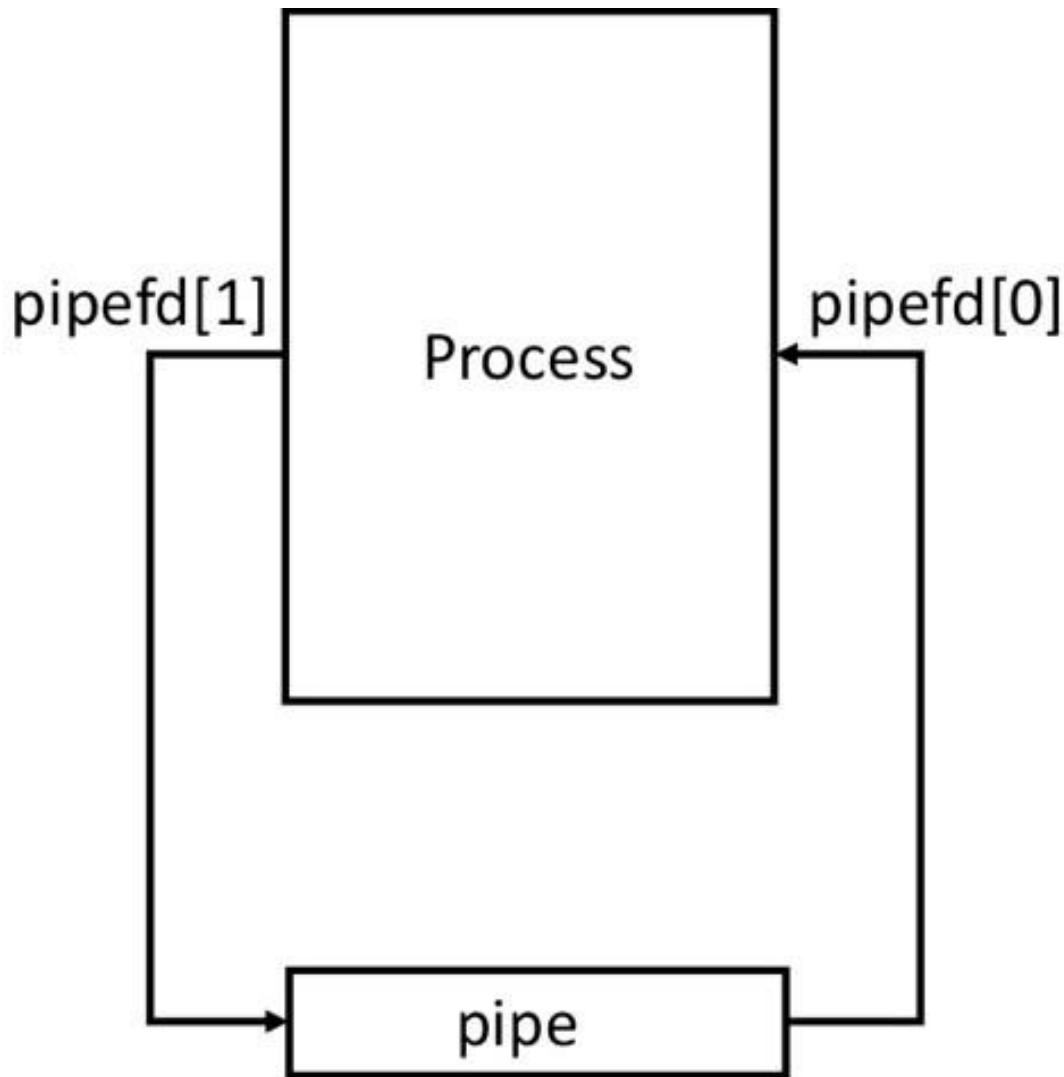
# The C API for the pipe function

- The C API for the pipe function is shown below:

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

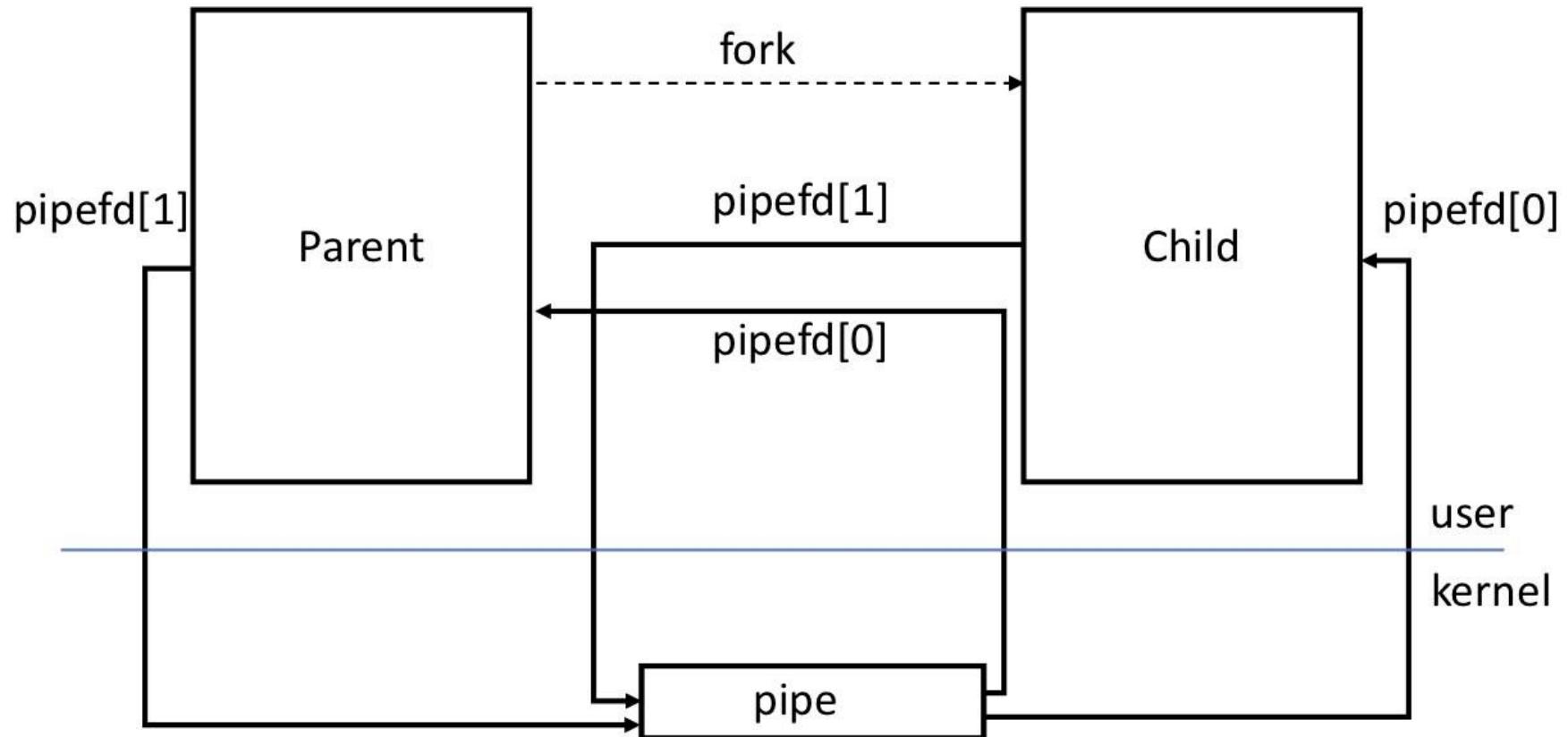
- The pipe call returns two file descriptors corresponding to the read and write ends of the pipe.
- The first file descriptor (*pipefd[0]*) refers to the read end of the pipe (can be used for reading data from the pipe) and the second file descriptor (*pipefd[1]*) refers to the write end of the pipe (can be used for writing data to the pipe).
- The kernel buffers the data written to the pipe until it is read from the read end of the pipe. When there is an error in creating the pipe, it returns -1 and sets the corresponding *errno*, otherwise it returns 0 on success.

- The diagram below illustrates the creation of a pipe in a single process.

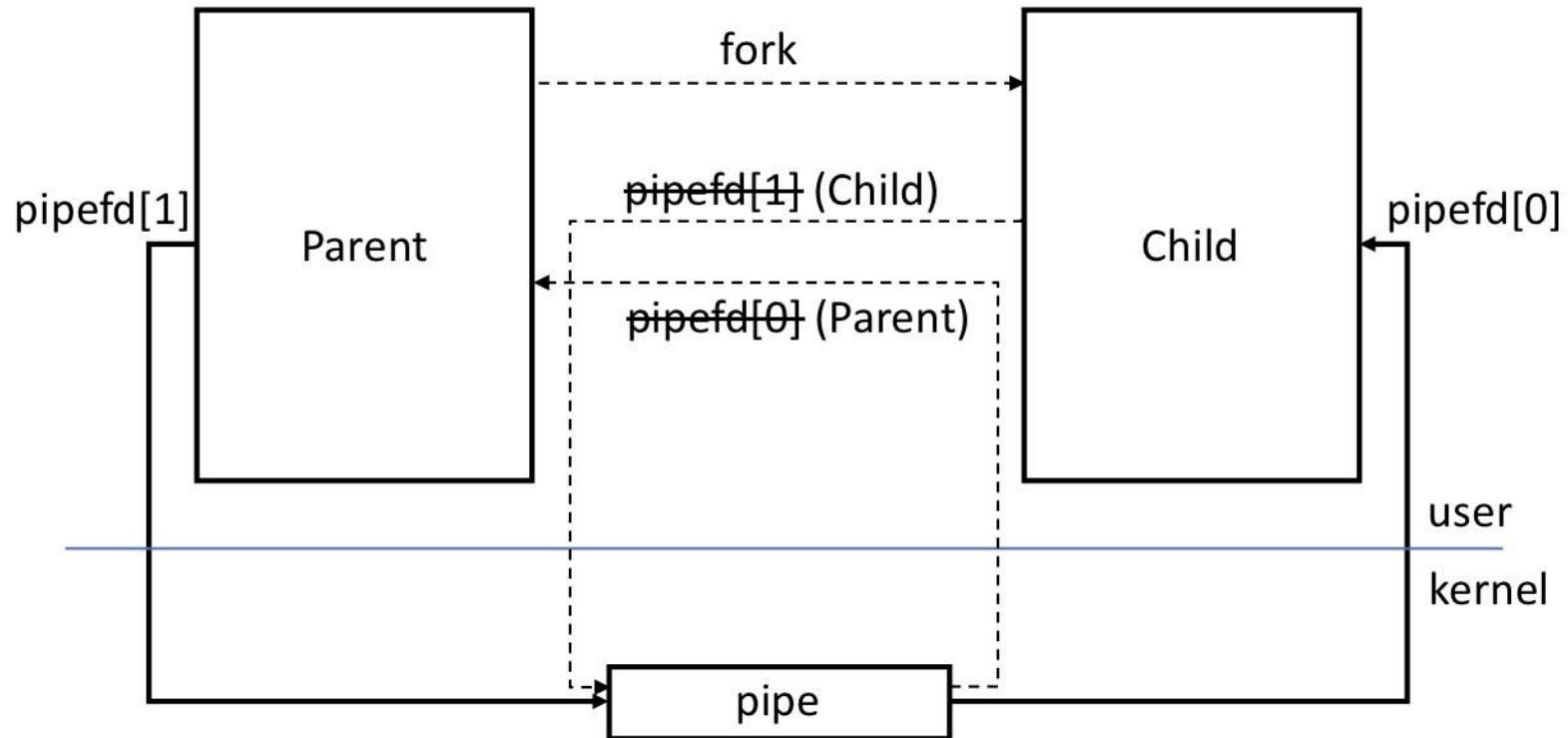


- We really don't need to create a pipe to communicate within the same process, typically we use pipes to communicate between a parent process and a child process.
- In such a case, first a pipe is created by the parent process and then it creates a child process using the fork command.
- Since fork creates a copy of the parent process, the child process will also inherit the all open file descriptors and will have access to the pipe

# parent - child



- To provide a unidirectional data channel for communication between the two process, the parent process closes the read end of the pipe and the child process closes the write end of the pipe as shown in the diagram below.



# Exercise 1

- The following example shows the steps involved in creating a pipe, forking a child process, closing the file descriptors in the parent and child process, and communication between the parent and child process.
- The parent process writes the string passed as the command-line argument to the pipe and the child process reads the string from the pipe, converts the string to uppercase, and prints it to the standard output.
- The read and write functions that operate on files are used to read and write data from the pipe.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <ctype.h>
6 #include <sys/wait.h>
7 #include <sys/stat.h>
8
9 int main(int argc, char **argv) {
10     pid_t pid;
11     int status;
12     int pipefd[2]; /* pipefd[0] for read, pipefd[1] for write */
13     char c;
14
15     if (argc != 2) {
16         printf("Usage: %s <string>\n", argv[0]);
17         exit(-1);
18     }
19
20     if (pipe(pipefd) == -1) { /* Open a pipe */
21         if ((pid = fork()) == -1) { /* I am the child process */
22             close(pipefd[1]); /* close write end */
```

```
23
24     while (read(pipefd[0], &c, 1) > 0) {
25         c = toupper(c);
26         write(1, &c, 1);
27     }
28     write(1, "\n", 1);
29     close(pipefd[0]);
30
31     exit(EXIT_SUCCESS);
32 } else if (pid > 0) { /* I am the parent process */
33     close(pipefd[0]); /* close read end */
34
35     write(pipefd[1], argv[1], strlen(argv[1]));
36     close(pipefd[1]);
37
38     wait(&status); /* wait for child to terminate */
39     if (WIFEXITED(status))
40         printf("Child process exited with status = %d\n", WEXITSTATUS(status));
```

```
41         else
42             printf("Child process did not terminate normally!\n");
43     } else { /* we have an error in fork */
44         perror("fork");
45         exit(EXIT_FAILURE);
46     }
47 } else {
48     perror("pipe");
49     exit(EXIT_FAILURE);
50 }
51
52 exit(EXIT_SUCCESS);
53 }
54 }
```

```
(base) mahmutunan@MacBook-Pro lecture28 % gcc -Wall pipe1.c -o pipe1
```

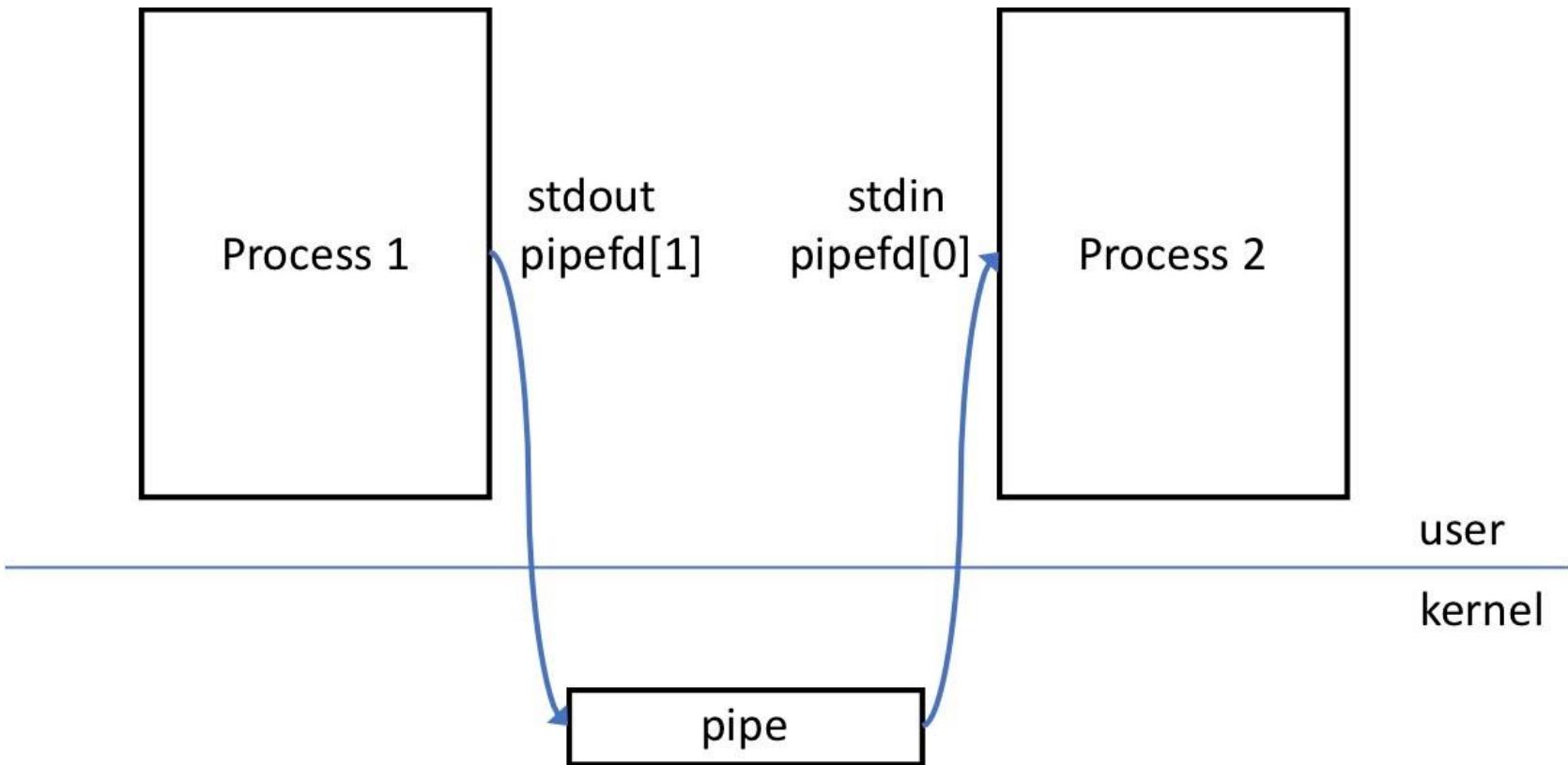
```
[base) mahmutunan@MacBook-Pro lecture28 % ./pipe1 cs332lowercase
CS332LOWERCASE
```

```
Child process exited with status = 0
```

```
(base) mahmutunan@MacBook-Pro lecture28 %
```

- The above example shows how the pipe is used by a parent and a child process to communicate.
- We can further extend this to implement pipes between any two programs such that the output of one program is redirected to the input of another program (e.g., `ps -elf | grep ssh`).
- In order to do this, we have to replace the standard output of the first program with the write end of the pipe and replace the standard input of the second program with the read end of the pipe.
- We have seen in the previous lecture/lab that this can be done using the `dup2` system call.
- We will use the `dup2` to perform this redirection and implement the pipe operation between two processes

- The pipe operation between two processes as shown in the diagram below.



- We have several options to create the two processes, some of the possible options include:
  1. The parent process creates a child process, the child process uses exec to launch the second program, and the parent process will use exec to launch the first program.
  2. The parent process creates two child process, the first child process uses exec to launch the first program, the second child process uses exec to launch the second program, and the parent process waits for the two child processes to terminate.
  3. The parent process create a child process, the child process creates another child process which in turn uses exec to launch the first program, then the child process uses exec to launch the second program, and the parent waits for the child process to terminate.

## Exercise 2

In this example, we will use the first option

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/types.h>
5 #include <sys/wait.h>
6 #include <sys/stat.h>
7
8 int main(int argc, char **argv) {
9     pid_t pid;
10    int pipefd[2]; /* fildes[0] for read, fildes[1] for write */
11
12    if (argc != 3) {
13        printf("Usage: %s <command1> <command2>\n", argv[0]);
14        exit(EXIT_FAILURE);
15    }
16
17    if (pipe(pipefd) == -1) { /* Open a pipe */
18        pid = fork(); /* fork child process to execute command2 */
19        if (pid == 0) { /* this is the child process */
20            /* close write end of the pipe */
21            close(pipefd[1]);
22
23            /* replace stdin with read end of pipe */
24            if (dup2(pipefd[0], 0) == -1) {
25                perror("dup2");
26                exit(EXIT_FAILURE);
27            }
28        }
29    }
30
31    /* wait for command2 to finish */
32    if (waitpid(pid, &status, 0) == -1) {
33        perror("waitpid");
34        exit(EXIT_FAILURE);
35    }
36
37    /* read output from command2 */
38    if (read(pipefd[0], buffer, sizeof(buffer)) == -1) {
39        perror("read");
40        exit(EXIT_FAILURE);
41    }
42
43    /* print output */
44    for (i = 0; i < strlen(buffer); i++) {
45        if (buffer[i] == '\n') {
46            printf("\n");
47        } else {
48            printf("%c", buffer[i]);
49        }
50    }
51
52    /* close read end of pipe */
53    close(pipefd[0]);
54}
```

# Exercise 2

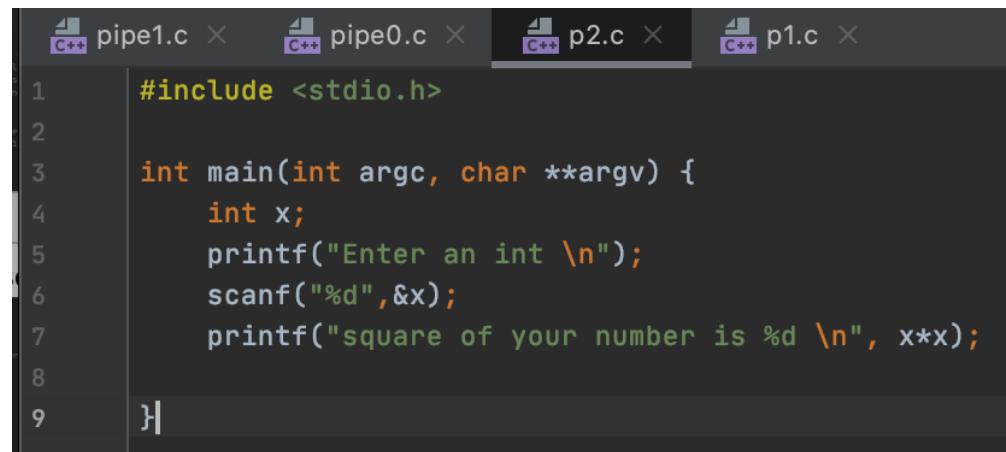
```
28
29     /* execute <command2> */
30     execlp(argv[2], argv[2], (char *)NULL);
31     perror("execlp");
32     exit(EXIT_FAILURE);
33 } else if (pid > 0) { /* this is the parent process */
34     /* close read end of the pipe */
35     close(pipefd[0]);
36
37     /* replace stdout with write end of pipe */
38     if (dup2(pipefd[1], 1) == -1) {
39         perror("dup2");
40         exit(EXIT_FAILURE);
41     }
42
43     /* execute <command1> */
44     execlp(argv[1], argv[1], (char *)NULL);
45     perror("execlp");
46     exit(EXIT_FAILURE);
47 } else if (pid < 0) { /* we have an error */
48     perror("fork"); /* use perror to print the system error message */
49     exit(EXIT_FAILURE);
50 }
51 } else {
52     perror("pipe");
53     exit(EXIT_FAILURE);
54 }
55
56     return 0;
57 }
```

# compile & run



```
#include <stdio.h>

int main(int argc, char **argv) {
    int a = 15, b = 25;
    printf("%d\n", a+b);
}
```



```
#include <stdio.h>

int main(int argc, char **argv) {
    int x;
    printf("Enter an int \n");
    scanf("%d", &x);
    printf("square of your number is %d \n", x*x);
}
```

```
[base] mahmutunan@MacBook-Pro lecture28 % gcc p1.c -o p1
[base] mahmutunan@MacBook-Pro lecture28 % ./p1
40
[base] mahmutunan@MacBook-Pro lecture28 % gcc p2.c -o p2
[base] mahmutunan@MacBook-Pro lecture28 % ./p2
Enter an int
5
square of your number is 25
[base] mahmutunan@MacBook-Pro lecture28 % gcc pipe0.c -o pipe0
[base] mahmutunan@MacBook-Pro lecture28 % ./pipe0 ./p1 ./p2
Enter an int
square of your number is 1600
[base] mahmutunan@MacBook-Pro lecture28 %
```

- We follow the same approach as the example above and create a pipe in the parent process.
- First we create a child process, close the read end of the pipe, replace the standard output stream with the write end of the pipe using dup2 system call, and then use exec to launch the first program.
- Then we create another child process, close the write end of the pipe, replace the standard input stream with the read end of the pipe using dup2 system call, and then use exec to launch the second program.
- The parent then closes both ends of the pipe and uses the waitpid system call to wait for both child process to terminate.
- The name of the two programs to be executed is provided as command-line arguments to this program.

# pipe2.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/types.h>
5 #include <sys/wait.h>
6 #include <sys/stat.h>
7
8
9 int main(int argc, char **argv) {
10     pid_t pid1, pid2;
11     int pipefd[2]; /* fildes[0] for read, fildes[1] for write */
12     int status1, status2;
13
14     if (argc != 3) {
15         printf("Usage: %s <command1> <command2>\n", argv[0]);
16         exit(EXIT_FAILURE);
17     }
18
19     if (pipe(pipefd) == -1) { /* Open a pipe */
20
21         pid1 = fork(); /* fork first process to execute command1 */
22         if (pid1 == 0) { /* this is the child process */
23             /* close read end of the pipe */
24             close(pipefd[0]);
25
26             /* replace stdout with write end of pipe */
27             if (dup2(pipefd[1], 1) != -1) {
28                 perror("dup2");
29                 exit(EXIT_FAILURE);
30             }
31         }
32     }
33 }
```

```
31
32     /* execute <command1> */
33     execlp(argv[1], argv[1], (char *)NULL);
34     printf("If you see this statement then exec failed ;-(\n");
35     perror("execlp");
36     exit(EXIT_FAILURE);
37
38 } else if (pid1 < 0) { /* we have an error */
39     perror("fork"); /* use perror to print the system error message */
40     exit(EXIT_FAILURE);
41 }
42
43 pid2 = fork(); /* fork second process to execute command2 */
44 if (pid2 == 0) { /* this is child process */
45     /* close write end of the pipe */
46     close(pipefd[1]);
47
48     /* replace stdin with read end of pipe */
49     if (dup2(pipefd[0], 0) == -1) {
50         perror("dup2");
51         exit(EXIT_FAILURE);
52     }
53
54     /* execute <command2> */
55     execlp(argv[2], argv[2], (char *)NULL);
56     printf("If you see this statement then exec failed ;-(\n");
57     perror("execlp");
58     exit(EXIT_FAILURE);
```

```
60 } else if (pid2 < 0) { /* we have an error */
61     perror("fork"); /* use perror to print the system error message */
62     exit(EXIT_FAILURE);
63 }
64
65     /* close the pipe in the parent */
66     close(pipefd[0]);
67     close(pipefd[1]);
68
69     /* wait for both child processes to terminate */
70     waitpid(pid1, &status1, 0);
71     waitpid(pid2, &status2, 0);
72 } else {
73     perror("pipe");
74     exit(EXIT_FAILURE);
75 }
76
77     return 0;
78 }
79 }
```

# compile & run

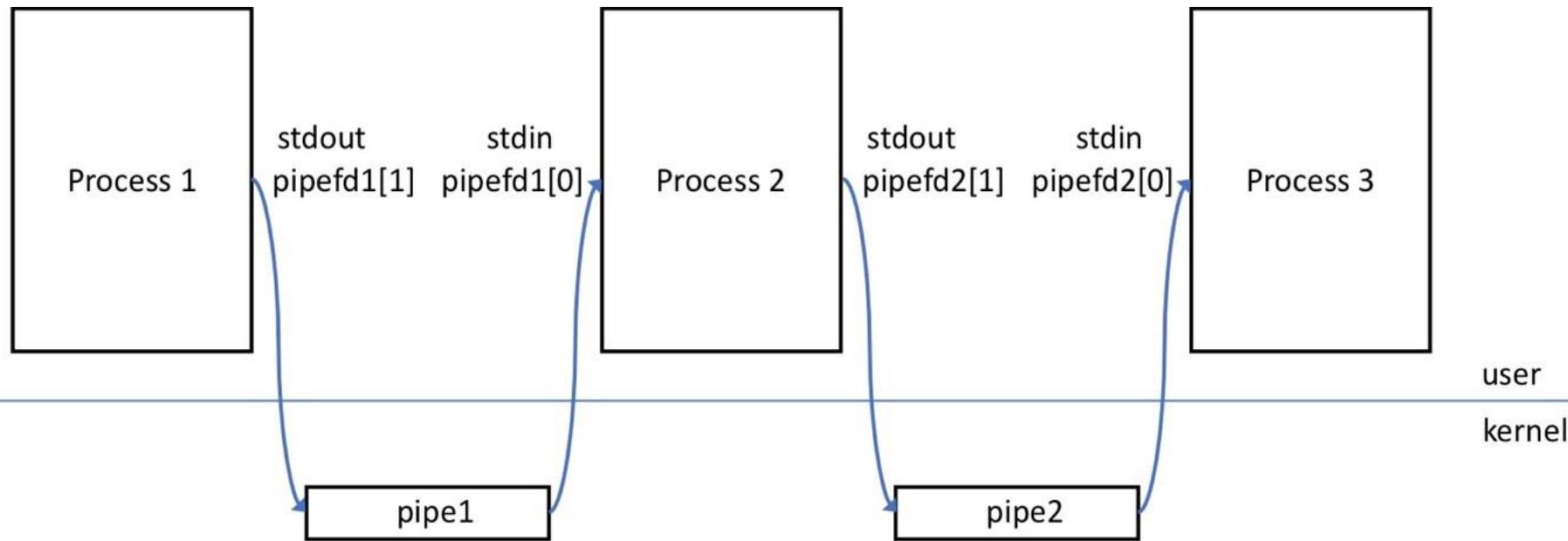
```
[base] mahmutunan@MacBook-Pro lecture29 % ls
p1          pager.c      pipe1      pipe3.c
p1.c        pager2.c    pipe1.c    popen.c
p2          pipe0       pipe2.c
p2.c        pipe0.c     pipe2a.c
(base) mahmutunan@MacBook-Pro lecture29 % gcc -Wall pipe2.c -o pipe2

(base) mahmutunan@MacBook-Pro lecture29 % ./pipe2 ls sort
p1
p1.c
p2
p2.c
pager.c
pager2.c
pipe0
pipe0.c
pipe1
pipe1.c
pipe2
pipe2.c
pipe2a.c
pipe3.c
popen.c
(base) mahmutunan@MacBook-Pro lecture29 % _
```

# extend to three process

- We can extend this to three processes if we like to implement something like: *ls* / *sort* / *wc*.
- We will create three processes and use two pipes
  - one for communication between the first and second process and one for communication between second and third process.
- The code for the first and third children will be similar to the example above while the second child has to replace both standard input and standard output streams instead of just one. T

# The diagram below illustrates this



# pipe3.c

```
1
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5  #include <sys/types.h>
6  #include <sys/wait.h>
7  #include <sys/stat.h>
8
9  int main(int argc, char **argv) {
10     pid_t pid1, pid2, pid3;
11     int pipefd1[2]; /* pipefd1[0] for read, pipefd1[1] for write */
12     int pipefd2[2]; /* pipefd2[0] for read, pipefd2[1] for write */
13     int status1, status2, status3;
14
15     if (argc != 4) {
16         printf("Usage: %s <command1> <command2> <command3>\n", argv[0]);
17         exit(EXIT_FAILURE);
18     }
19
20     if (pipe(pipefd1) != 0) { /* Open pipefd1 */
21         perror("pipe");
22         exit(EXIT_FAILURE);
23     }
24
25     if (pipe(pipefd2) != 0) { /* Open pipefd2 */
26         perror("pipe");
27         exit(EXIT_FAILURE);
28     }
29
30     pid1 = fork(); /* fork first process to execute command1 */
31     if (pid1 == 0) { /* this is the child process */
32         /* close read end of the pipefd1 */
33         close(pipefd1[0]);
34
35         /* close both ends of pipefd2 */
36         close(pipefd2[0]);
37         close(pipefd2[1]);
```

```
39          /* replace stdout with write end of pipefd1 */
40          if (dup2(pipefd1[1], 1) == -1) {
41              perror("dup2");
42              exit(EXIT_FAILURE);
43          }
44
45          /* execute <command1> */
46          execlp(argv[1], argv[1], (char *)NULL);
47          perror("execlp");
48          exit(EXIT_FAILURE);
49
50      } else if (pid1 < 0) { /* we have an error */
51          perror("fork"); /* use perror to print the system error message */
52          exit(EXIT_FAILURE);
53      }
54
55      pid2 = fork(); /* fork second process to execute command2 */
56      if (pid2 == 0) { /* this is child process */
57          /* close write end of the pipefd1 */
58          close(pipefd1[1]);
59
60          /* replace stdin with read end of pipefd2 */
61          if (dup2(pipefd1[0], 0) == -1) {
62              perror("dup2");
63              exit(EXIT_FAILURE);
64          }
65
66          /* close read end of pipefd2 */
67          close(pipefd2[0]);
68
69          /* replace stdout with write end of pipefd2 */
70          if (dup2(pipefd2[1], 1) == -1) {
71              perror("dup2");
72              exit(EXIT_FAILURE);
73          }
74
75          /* execute <command2> */
76          execlp(argv[2], argv[2], (char *)NULL);
77          perror("execlp");
78          exit(EXIT_FAILURE);
```

```
79
80 } else if (pid2 < 0) { /* we have an error */
81     perror("fork"); /* use perror to print the system error message */
82     exit(EXIT_FAILURE);
83 }
84
85 pid3 = fork(); /* fork third process to execute command3 */
86 if (pid3 == 0) { /* this is child process */
87     /* close both ends of the pipefd1 */
88     close(pipefd1[0]);
89     close(pipefd1[1]);
90
91     /* close write end of pipefd2 */
92     close(pipefd2[1]);
93
94     /* replace stdin with read end of pipefd2 */
95     if (dup2(pipefd2[0], 0) == -1) {
96         perror("dup2");
97         exit(EXIT_FAILURE);
98     }
99
100    /* execute <command3> */
101    execlp(file: argv[3], arg0: argv[3], (char *)NULL);
102    perror("execlp");
103    exit(EXIT_FAILURE);
104
105 } else if (pid3 < 0) { /* we have an error */
106     perror("fork"); /* use perror to print the system error message */
107     exit(EXIT_FAILURE);
108 }
109
110 /* close the pipes in the parent */
111 close(pipefd1[0]);
112 close(pipefd1[1]);
113 close(pipefd2[0]);
114 close(pipefd2[1]);
115
116 /* wait for both child processes to terminate */
117 waitpid(pid1, &status1, 0);
118 waitpid(pid2, &status2, 0);
119 waitpid(pid3, &status3, 0);
120
121     return 0;
122 }
```

# compile & run

```
(base) mahmutunan@MacBook-Pro lecture29 % gcc -Wall pipe3.c -o pipe3  
[  
(base) mahmutunan@MacBook-Pro lecture29 % ./pipe3 ps sort wc  
      5      23     161  
(base) mahmutunan@MacBook-Pro lecture29 % ]
```

# Pager.C

```
1
2 ● #include <stdio.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5 #include <string.h>
6 #include <cctype.h>
7 #include <sys/wait.h>
8 #include <sys/stat.h>
9
10 int main(int argc, char **argv) {
11     pid_t pid;
12     int status;
13     int pipefd[2]; /* pipefd[0] for read, pipefd[1] for write */
14     FILE *fp;
15     char line[BUFSIZ];
16     int n;
17
18     if (argc != 2) {
19         printf("Usage: %s <filename>\n", argv[0]);
20         exit(-1);
21     }
22
23     if ((fp = fopen(filename: argv[1], mode: "r")) == NULL) {
24         printf("Error opening file %s for reading\n", argv[1]);
25         exit(-1);
26     }
27
28     if (pipe(pipefd) == 0) { /* Open a pipe */
29         if ((pid = fork()) == 0) { /* I am the child process */
30             close(pipefd[1]); /* close write end */
31             dup2(pipefd[0], STDIN_FILENO); /* replace stdin of child */
32             execlp(file: "/usr/bin/more", arg0: "more", (char *)NULL);
33             perror("exec");
34             exit(EXIT_FAILURE);
35         }
36     }
37 }
```

```
34         exit(EXIT_FAILURE);
35     } else if (pid > 0) { /* I am the parent process */
36         close(pipefd[0]);    /* close read end */
37         /* read lines from the file and write it to pipe */
38         while (fgets(line, BUFSIZ, fp) != NULL) {
39             n = strlen(line);
40             if (write(fd: pipefd[1], line, n) != n) {
41                 printf("Error writing to pipe\n");
42                 exit(-1);
43             }
44         }
45         close(pipefd[1]);    /* close write end */
46
47         wait(&status);      /* wait for child to terminate */
48         if (WIFEXITED(status))
49             printf("Child process exited with status = %d\n", WEXITSTATUS(status));
50         else
51             printf("Child process did not terminate normally!\n");
52     } else { /* we have an error in fork */
53         perror("fork");
54         exit(EXIT_FAILURE);
55     }
56 } else {
57     perror("pipe");
58     exit(EXIT_FAILURE);
59 }
60
61 exit(EXIT_SUCCESS);
62 }
```

# compile & run

```
(base) mahmutunan@MacBook-Pro lecture29 % ./pager smalltale.txt
it was the best of times it was the worst of times
it was the age of wisdom it was the age of foolishness
it was the epoch of belief it was the epoch of incredulity
it was the season of light it was the season of darkness
it was the spring of hope it was the winter of despair
we had everything before us we had nothing before us
we were all going direct to heaven we were all going direct
the other wayin short the period was so far like the present
period that some of its noisiest authorities insisted on its
being received for good or for evil in the superlative degree
of comparison only
```

```
there were a king with a large jaw and a queen with a plain face
on the throne of england there were a king with a large jaw and
a queen with a fair face on the throne of france in both
countries it was clearer than crystal to the lords of the state
```

# popen and pclose functions

- As we have seen in the examples, the common usage of pipes involve creating a pipe, creating a child process with fork, closing the unused ends of the pipe, execing a command in the child process, and waiting for the child process to terminate in the parent process.
- Since this is such a common usage, UNIX systems provide *popen* and *pclose* functions that perform most of these operations in a single operation.
- The C APIs for the *popen* and *pclose* functions are shown below:

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

```
FILE *popen(const char *command, const char *type);  
int pclose(FILE *stream);
```

- The popen function performs the following steps:
  - creates a pipe
  - creates a new process using fork
  - perform the following steps in the child process
    - close unused ends of the pipe (based on the *type* argument)
    - execs a shell to execute the *command* provided as argument to popen (i.e., executes "sh -c command")
  - perform the following steps in the parent process
    - close unused ends of the pipe (based on the *type* argument)
    - wait for the child process to terminate

- The *popen* function returns the FILE handle to the pipe created so that the calling process can read or write to the pipe using standard I/O system calls.
- If the *type* argument is specified as read-only ("r") then the calling process can read from the pipe, this results in reading from the *stdout* of the child process (see Figure 15.9).
- If the *type* argument is specified as write-only ("w") then the calling process can write to the pipe, this results in writing to the *stdin* of the child process created (see Figure 15.10).

- The FILE handle returned by *popen* must be closed using *pclose* to make sure that the I/O stream opened to read or write to the pipe is closed and wait for the child process to terminate.
- The termination status of the shell started by *exec* will be returned when the *pclose* function returns.
-

# pipe2a.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char **argv) {
5      FILE *fp1, *fp2;
6      char line[BUFSIZ];
7
8      if (argc != 3) {
9          printf("Usage: %s <command1> <command2>\n", argv[0]);
10         exit(EXIT_FAILURE);
11     }
12
13     /* create a pipe, fork/exec command argv[1], in "read" mode */
14     /* read mode - parent process reads stdout of child process */
15     if ((fp1 = popen(argv[1], "r")) == NULL) {
16         perror("popen");
17         exit(EXIT_FAILURE);
18     }
19 }
```

```
19
20     /* create a pipe, fork/exec command argv[2], in "write" mode */
21     /* write mode - parent process writes to stdin of child process */
22     if ((fp2 = popen(argv[2], "w")) == NULL) {
23         perror("popen");
24         exit(EXIT_FAILURE);
25     }
26
27     /* read stdout from child process 1 and write to stdin of
28      child process 2 */
29     while (fgets(line, BUFSIZ, fp1) != NULL) {
30         if (fputs(line, fp2) == EOF) {
31             printf("Error writing to pipe\n");
32             exit(EXIT_FAILURE);
33         }
34     }
35
36     /* wait for child process to terminate */
37     if ((pclose(fp1) == -1) || pclose(fp2) == -1) {
38         perror("pclose");
39         exit(EXIT_FAILURE);
40     }
41
42     return 0;
43 }
```

# compile & run

```
(base) mahmutunan@MacBook-Pro lecture29 % ./pipe2a "ls -l" sort
-rw-r--r--@ 1 mahmutunan staff 105 Nov 2 13:37 p1.c
-rw-r--r--@ 1 mahmutunan staff 169 Nov 2 13:36 p2.c
-rw-r--r--@ 1 mahmutunan staff 790 Oct 27 22:41 popen.c
-rw-r--r--@ 1 mahmutunan staff 1694 Oct 27 22:41 pager2.c
-rw-r--r--@ 1 mahmutunan staff 1853 Nov 4 13:07 pipe2a.c
-rw-r--r--@ 1 mahmutunan staff 2073 Oct 27 22:41 pipe1.c
-rw-r--r--@ 1 mahmutunan staff 2121 Oct 27 22:41 pipe0.c
-rw-r--r--@ 1 mahmutunan staff 2284 Nov 4 12:44 pager.c
-rw-r--r--@ 1 mahmutunan staff 2782 Nov 4 11:31 pipe2.c
-rw-r--r--@ 1 mahmutunan staff 3858 Nov 4 11:51 pipe3.c
-rw-r--r--@ 1 mahmutunan staff 5074 Nov 4 12:51 smalltale.txt
-rwxr-xr-x 1 mahmutunan staff 12556 Nov 2 13:37 p1
-rwxr-xr-x 1 mahmutunan staff 12604 Nov 2 13:37 p2
-rwxr-xr-x 1 mahmutunan staff 12952 Nov 4 13:07 pipe2a
-rwxr-xr-x 1 mahmutunan staff 12984 Nov 2 13:37 pipe0
-rwxr-xr-x 1 mahmutunan staff 12996 Nov 2 13:20 pipe1
-rwxr-xr-x 1 mahmutunan staff 13040 Nov 4 11:31 pipe2
-rwxr-xr-x 1 mahmutunan staff 13040 Nov 4 12:41 pipe3
-rwxr-xr-x 1 mahmutunan staff 13076 Nov 4 12:44 pager
total 352
```

- Note that since the command is executed using a shell, we can provide wildcards and other special characters that the shell can expand.
- Also note that in this version of the program the parent process is reading the *stdout* stream of the first child process and then writing to the *stdin* stream of the second child process (we did not do this in the first version).

# pager2.c

Here is an updated version of the pager program that uses popen and pclose

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

int main(int argc, char **argv) {
    FILE *fpin, *fpout;
    char line[BUFSIZ];

    if (argc != 2) {
        printf("Usage: %s <filename>\n", argv[0]);
        exit(-1);
    }

    /* open file for reading */
    if ( (fpin = fopen(argv[1], "r")) == NULL ) {
        printf("Error opening file %s for reading\n", argv[1]);
        exit(-1);
    }

    /* create a pipe, fork/exec process "more", in "write" mode */
    /* write mode - parent process writes, child process reads */
    if ( (fpout = popen("more", "w")) == NULL ) {
        perror("exec");
        exit(EXIT_FAILURE);
    }
}
```

```
/* read lines from the file and write it fpout */
while (fgets(line, BUFSIZ, fpin) != NULL) {
    if (fputs(line, fpout) == EOF) {
        printf("Error writing to pipe\n");
        exit(EXIT_FAILURE);
    }
}

/* close the pipe and wait for child process to terminate */
if (pclose(fpout) == -1) {
    perror("pclose");
    exit(EXIT_FAILURE);
}

exit(EXIT_SUCCESS);
}
```

# compile & run

```
(base) mahmutunan@MacBook-Pro lecture29 % gcc -Wall pager2.c -o pager2  
(base) mahmutunan@MacBook-Pro lecture29 % ./pager2 smalltale.txt  
it was the best of times it was the worst of times  
it was the age of wisdom it was the age of foolishness  
it was the epoch of belief it was the epoch of incredulity  
it was the season of light it was the season of darkness  
it was the spring of hope it was the winter of despair  
we had everything before us we had nothing before us  
we were all going direct to heaven we were all going direct  
the other wayin short the period was so far like the present  
period that some of its minor distinctions have almost entirely vanished
```

# popen.c

- You can also find a simpler version of the program that uses a single `popen` system call to create a pipe in "read" mode, execute the command specified as the command-line argument, reads the pipe and prints it to `stdout`

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

int main(int argc, char **argv) {
    FILE *fp;
    char line[BUFSIZ];

    if (argc != 2) {
        printf("Usage: %s <command>\n", argv[0]);
        exit(EXIT_FAILURE);
    }
}
```

```
    if ((fp = popen(argv[1], "r")) == NULL) {
        perror("popen");
        exit(EXIT_FAILURE);
    }

    while (fgets(line, BUFSIZ, fp) != NULL) {
        fputs(line, stdout);
    }

    if (pclose(fp) == -1) {
        perror("pclose");
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

# compile & run

```
(base) mahmutunan@MacBook-Pro lecture29 % gcc -Wall popen.c -o popen  
[  
[base) mahmutunan@MacBook-Pro lecture29 % ./popen ps  
 PID TTY          TIME CMD  
1600 ttys000    0:00.46 -zsh  
26658 ttys000    0:00.00 ./popen ps  
(base) mahmutunan@MacBook-Pro lecture29 % _
```

# THREAD

# Threads

- In an OS that supports threads, scheduling and dispatching is done on a thread basis
- Most of the state information dealing with execution is maintained in thread-level data structures
  - Suspending a process involves suspending all threads of the process
  - Termination of a process terminates all threads within the process

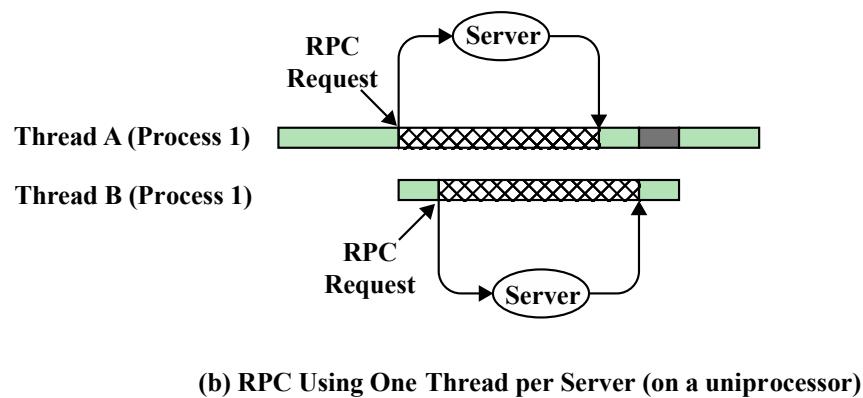
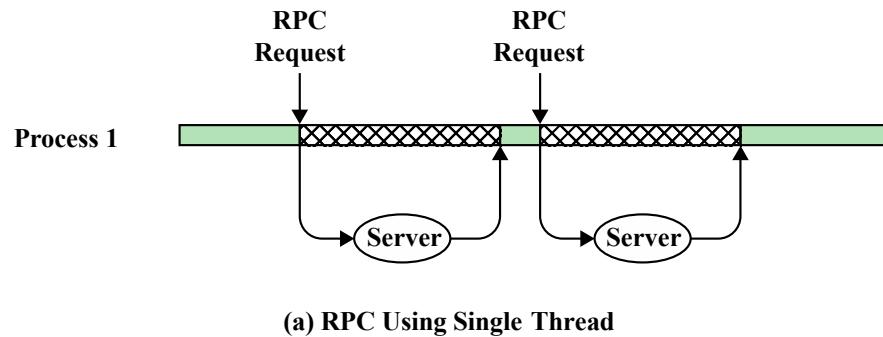
# Thread Execution States

The key states for a thread are:

- Running
- Ready
- Blocked

Thread operations associated with a change in thread state are:

- Spawn
- Block
- Unblock
- Finish

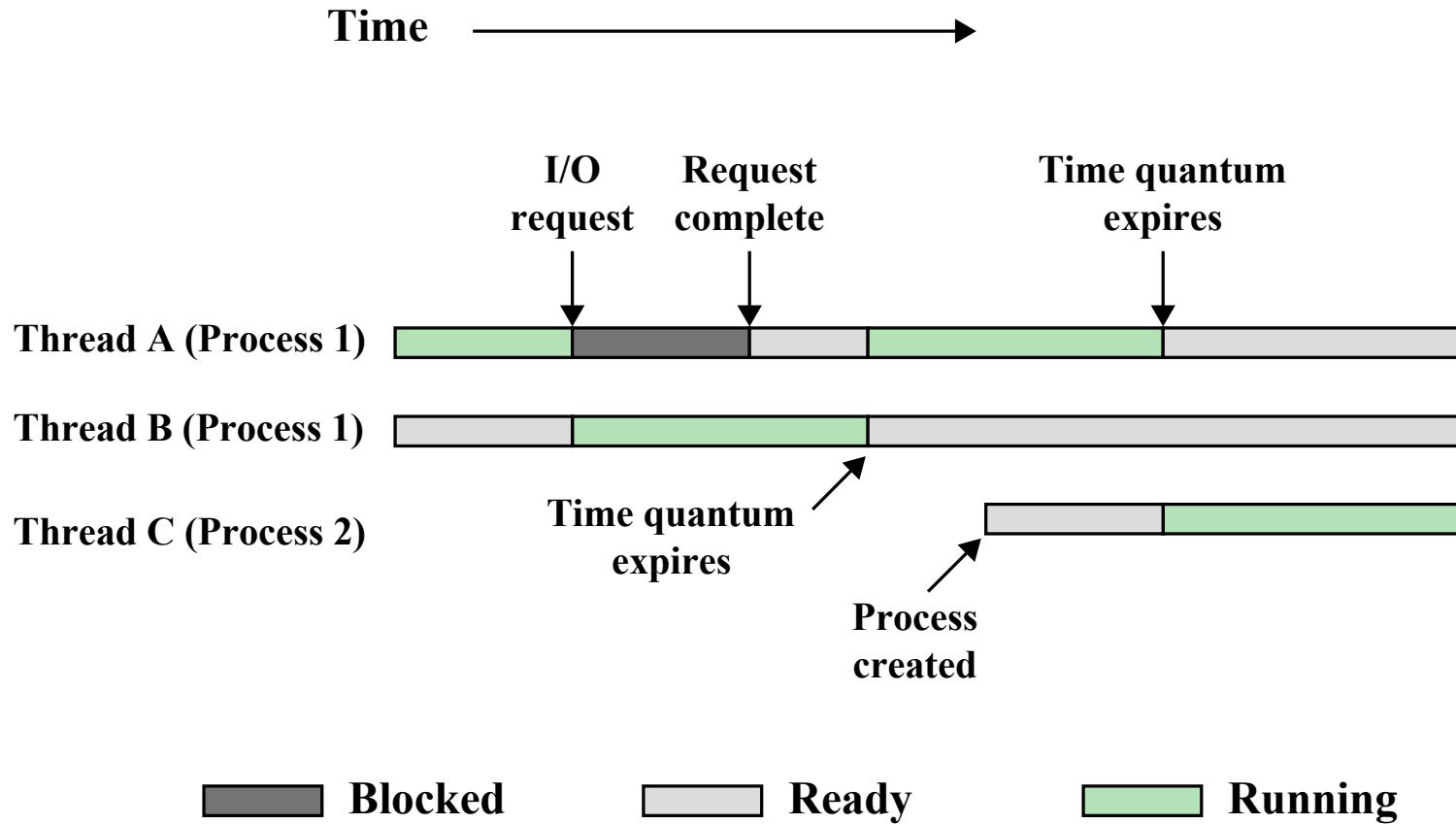


Blocked, waiting for response to RPC

Blocked, waiting for processor, which is in use by Thread B

Running

**Figure 4.3 Remote Procedure Call (RPC) Using Threads**

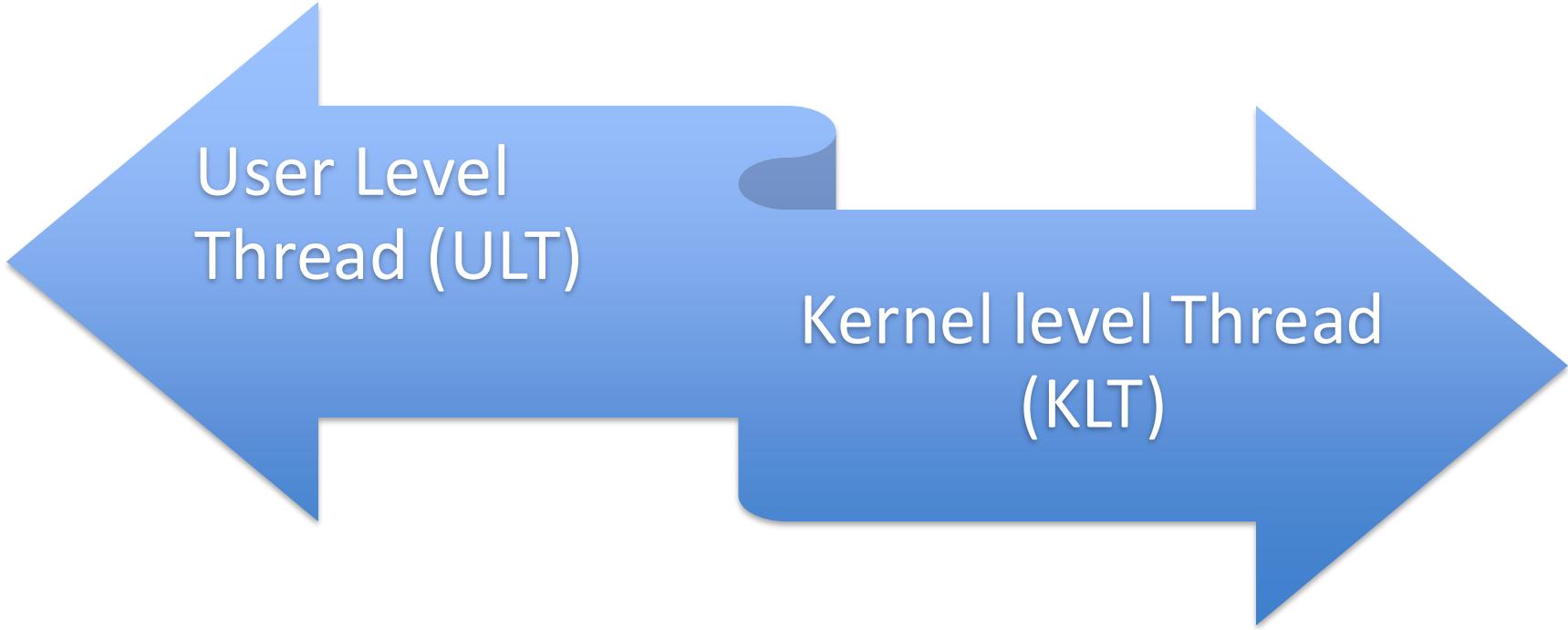


**Figure 4.4 Multithreading Example on a Uniprocessor**

# Thread Synchronization

- It is necessary to synchronize the activities of the various threads
  - All threads of a process share the same address space and other resources
  - Any alteration of a resource by one thread affects the other threads in the same process

# Types of Threads

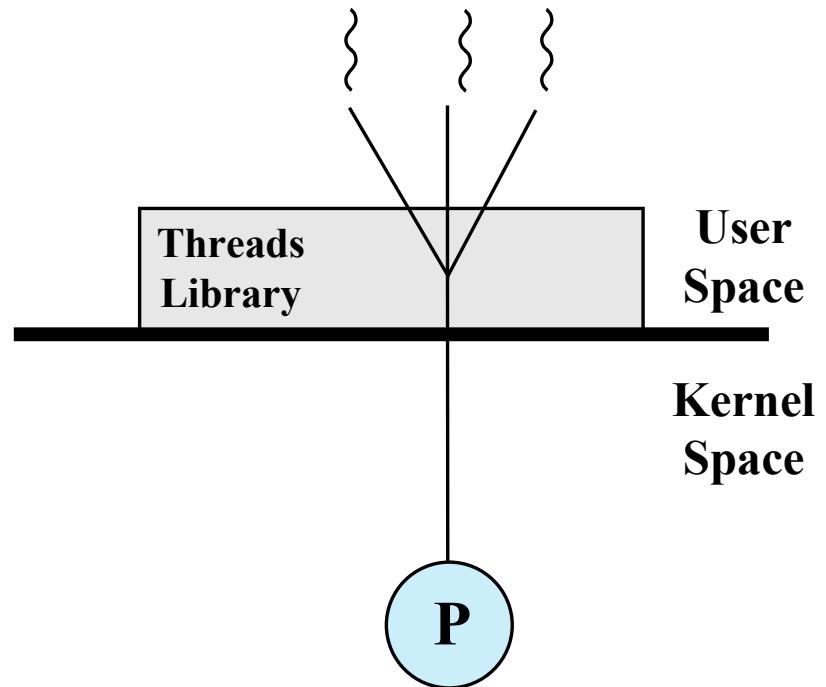


User Level  
Thread (ULT)

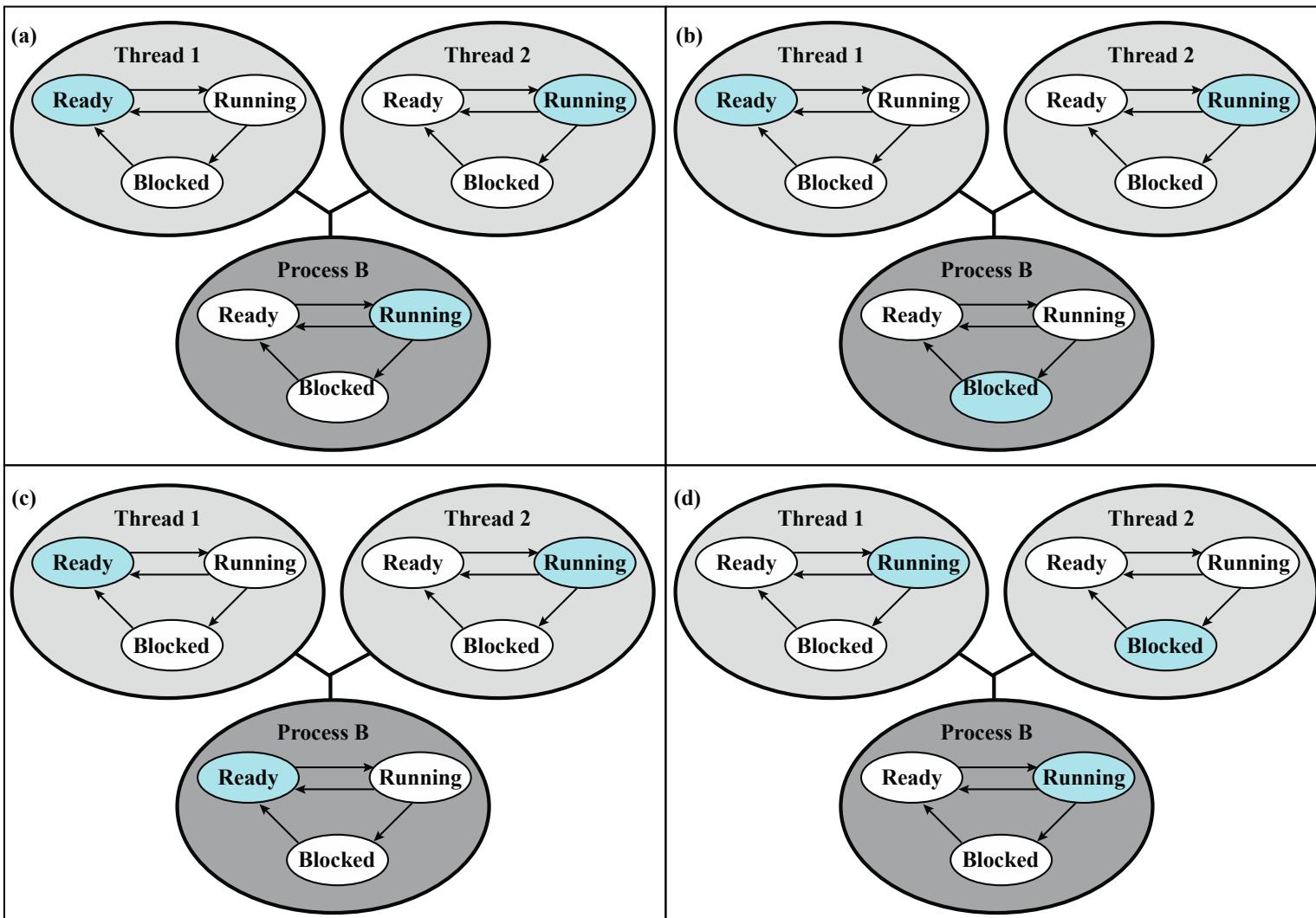
Kernel level Thread  
(KLT)

# User-Level Threads (ULTs)

- All thread management is done by the application
- The kernel is not aware of the existence of threads

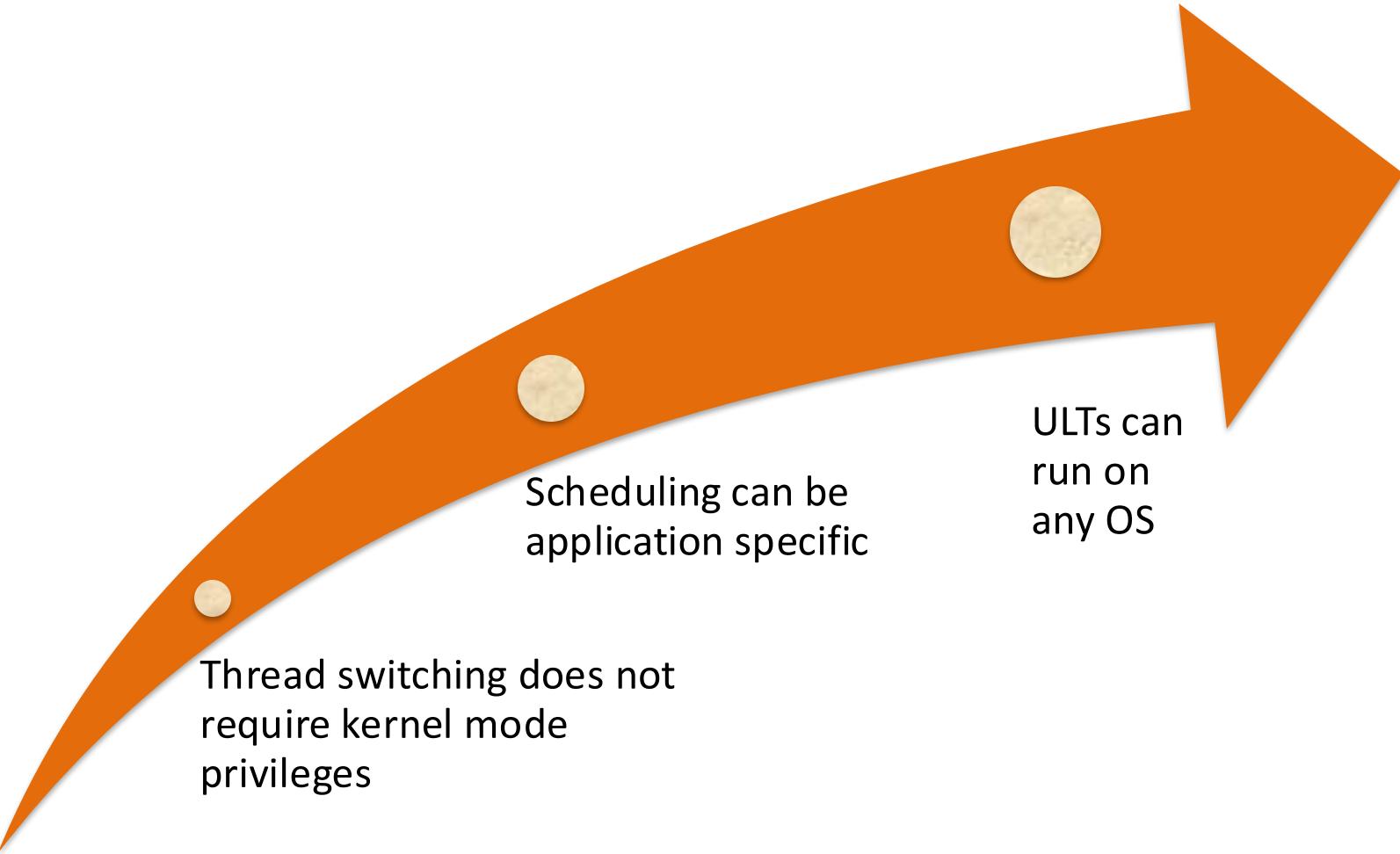


(a) Pure user-level



**Figure 4.6 Examples of the Relationships Between User-Level Thread States and Process States**

# Advantages of ULTs



- Thread switching does not require kernel mode privileges

- Scheduling can be application specific

- ULTs can run on any OS

# Disadvantages of ULTs

- In a typical OS many system calls are blocking
  - As a result, when a ULT executes a system call, not only is that thread blocked, but all of the threads within the process are blocked as well
- In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing
  - A kernel assigns one process to only one processor at a time, therefore, only a single thread within a process can execute at a time

# Overcoming ULT Disadvantages

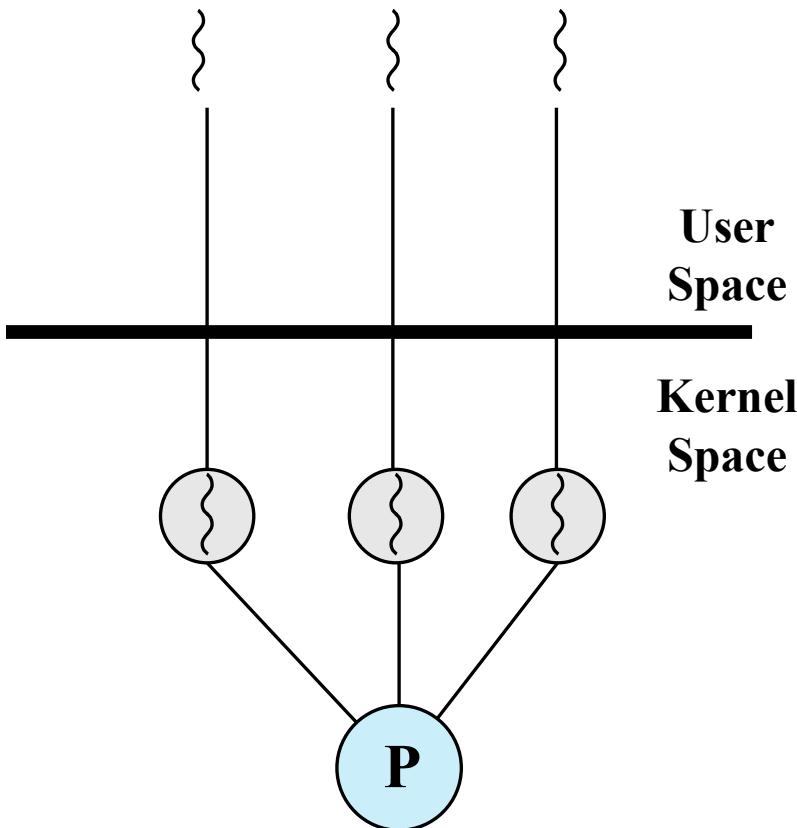
## Jacketing

- Purpose is to convert a blocking system call into a non-blocking system call

Writing an application as multiple processes rather than multiple threads

- However, this approach eliminates the main advantage of threads

# Kernel-Level Threads (KLTs)



(b) Pure kernel-level

- Thread management is done by the kernel
  - There is no thread management code in the application level, simply an application programming interface (API) to the kernel thread facility
  - Windows is an example of this approach

# Advantages of KLTs

- The kernel can simultaneously schedule multiple threads from the same process on multiple processors
- If one thread in a process is blocked, the kernel can schedule another thread of the same process
- Kernel routines themselves can be multithreaded

# Disadvantage of KLTs

- The transfer of control from one thread to another within the same process requires a mode switch to the kernel

Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

**Table 4.1**  
**Thread and Process Operation Latencies ( $\mu$ s)**

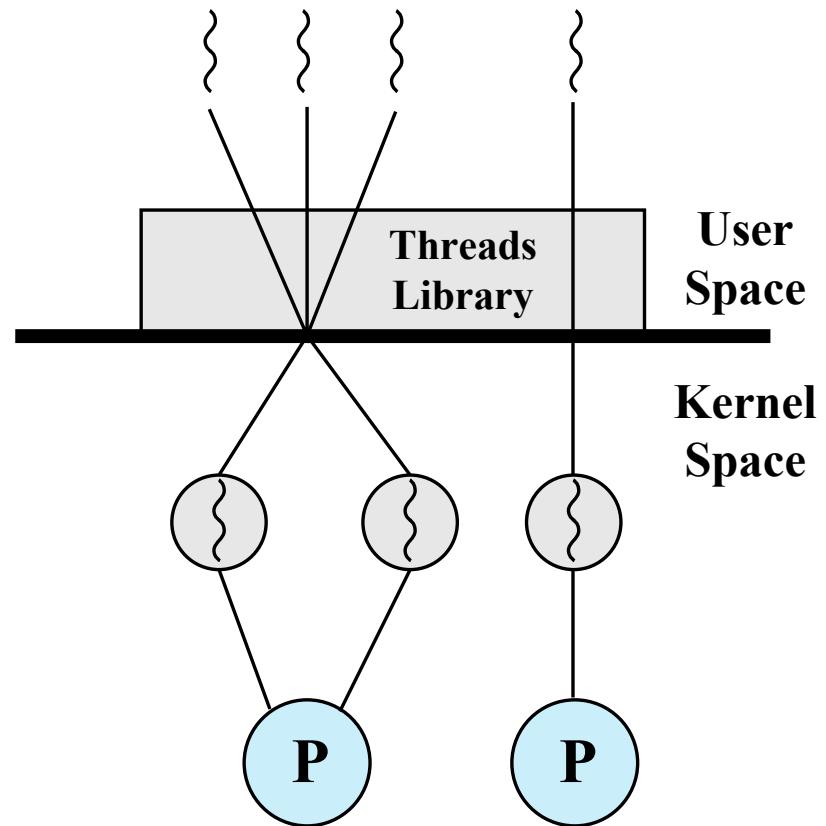
S.N.	<b>User-Level Threads</b>	<b>Kernel-Level Thread</b>
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.
4	Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

# Multithreading Models

- Many to many relationship.
- Many to one relationship.
- One to one relationship.

# Combined Approaches

- Thread creation is done completely in the user space, as is the bulk of the scheduling and synchronization of threads within an application
- Solaris is a good example



(c) Combined

Threads:Processes	Description	Example Systems
<b>1:1</b>	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
<b>M:1</b>	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH
<b>1:M</b>	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra (Clouds), Emerald
<b>M:N</b>	Combines attributes of M:1 and 1:M cases.	TRIX

**Table 4.2**  
**Relationship between Threads and Processes**

# Create threads using POSIX threads library

- In the previous lectures/labs we focused on how to create processes, in this lab we will focus on creating threads and mechanisms for establishing synchronization among threads.
- First, let us understand the difference between a process and a thread.
  - A process could be considered to have two characteristics:
    - (a) resource ownership
    - (b) scheduling or execution.
- The unit of scheduling and dispatching is usually referred to as a **thread** or **lightweight process** and the ability of to support multiple, concurrent paths of execution within a single process is often referred to as *multithreading*.

- Threads offer several benefits compared to a process:
  - Threads takes less time to create a new thread than a process
  - Threads take less time to terminate a thread than a process
  - Switching between two threads (context switching) takes less time than switching between processes
  - All of the threads in a process share the state and resources of that process (since threads reside in the same address space and have access to the same data)
  - Threads enhance efficiency in communication between programs (since threads share memory and files within the same process and can communicate without invoking the kernel)

- As a result of these advantages, if we have to implement a set of functions that are closely related, implementing this functionality using multiple threads is far more efficient than using multiple processes.
- We will use the **POSIX threads library**, usually referred to as Pthreads library, that provides C APIs to create and manage threads. We have to include the file *pthread.h* and link with *-lpthread* to compile and link.
- We can create new threads using the *pthread\_create()* function which has the following function definition:

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const
pthread_attr_t *attr,
void *(*start_routine) (void *), void *arg);
```

- The new thread that will be created by the *pthread\_create* function will invoke the function *start\_routine*.
- Note that the function *start\_routine* takes one argument of type *void \** and has the return type as *void \**.
- In other words, the function *start\_routine* has the following function definition:

```
void *start_routine(void *arg)
```

- When the *pthread\_create* call returns successfully, it returns the thread ID associated with the new thread created in the variable *thread*.
- This can be used by the main thread in subsequent *pthread* function calls such as *pthread\_join*.
- The second argument, *attr*, provides a reference to the *pthread\_attr\_t* structure that describes the various attributes of the new thread to be created.
- It can be initialized using *pthread\_attr\_init* call or set to NULL if default attributes must be used.
- You can find out more about the different thread attributes that can be specified by looking at the man page for *pthread\_attr\_init*.

- The new thread created will terminate when the function *start\_routine* returns or when a call to *pthread\_exit* is made inside the *start\_routine*.
- We can use the *pthread\_join* function to wait for a thread to complete using the thread ID that was returned when *pthread\_create* call was invoked.
- If a thread has already completed,
  - *pthread\_join* will return immediately, otherwise, it will wait for the corresponding thread to complete.

# exercise 1

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <pthread.h>
5
6
7 void *someFuncToCreateThread(void *someValue)
8 {
9     sleep(2);
10    printf("I am inside the thread \n");
11    return NULL;
12 }
13
14 int main()
15 {
16     pthread_t thread_id;
17     printf("I am inside the main function\n");
18     pthread_create(&thread_id, NULL, someFuncToCreateThread, NULL);
19     pthread_join(thread_id, NULL);
20     printf("Back to the main function\n");
21     exit(0);
22 }
23
```

# compile & run

To compile a multithreaded program, we will be using gcc and we need to link it with the pthreads library.

```
(base) mahmutunan@MacBook-Pro lecture31 % gcc exercise1.c -o exercise1 -lpthread
(base) mahmutunan@MacBook-Pro lecture31 % ./exercise1
I am inside the main function
I am inside the thread
Back to the main function
(base) mahmutunan@MacBook-Pro lecture31 %
```

# exercise 2

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

void *function1(void *someValue)
{
    while(1==1) {
        sleep(1);
        printf("function 1 \n");
    }
}

void function2()
{
    while(1==1) {
        sleep(2);
        printf("function 2\n");
    }
}

int main()
{
    pthread_t thread_id;
    printf("I am inside the main function\n");
    pthread_create(&thread_id, NULL, function1, NULL);
    function2();
    exit(0);
}
```

# compile & run

```
[base] mahmutunan@MacBook-Pro lecture31 % gcc exercise2.c -o exercise2 -lpthread
[base] mahmutunan@MacBook-Pro lecture31 % ./exercise2
I am inside the main function
function 1
function 2
function 1
function 1
function 1
```

# exercise 3

```
int globalVar = 50; //define a global variable

void *someFuncToCreateThread(void *someValue)
{
    int *threadId = (int *)someValue; // Store the value argument passed to this thread

    //define a static and a local variable
    static int staticVar = 75;
    int localVar = 10;

    // let's change the variables
    globalVar +=100;
    staticVar +=100;
    localVar +=100;
    printf("id =%d,global = %d,  local = %d, static =%d, \n", *threadId, globalVar,localVar ,staticVar);

    return NULL;
}

int main()
{
    int i;
    pthread_t thread_id;
    for (i = 0; i < 4; i++)
        pthread_create(&thread_id, NULL, someFuncToCreateThread, (void *)&thread_id);
    pthread_exit(NULL);
}
```

# compile & run

```
(base) mahmutunan@MacBook-Pro lecture31 % gcc exercise3.c -o exercise3 -lpthread  
(base) mahmutunan@MacBook-Pro lecture31 % ./exercise3  
id =151261184,global = 150, local = 110, static =175,  
id =151261184,global = 150, local = 110, static =175,  
id =151261184,global = 250, local = 110, static =275,  
id =151261184,global = 250, local = 110, static =275,  
(base) mahmutunan@MacBook-Pro lecture31 %
```

Remember, global and static variables are stored in data segment.

All threads share data segment, so they are shared by all threads.

# pthread1.c

```
1  ● #include <stdio.h>
2
3  #include <stdlib.h>
4
5  #include <pthread.h>
6
7
8  int nthreads;
9
10 void *compute(void *arg) {
11     long tid = (long)arg;
12
13     printf("Hello, I am thread %ld of %d\n", tid, nthreads);
14
15     return (NULL);
16 }
17
18 int main(int argc, char **argv) {
19     long i;
20     pthread_t *tid;
21
22     if (argc != 2) {
23         printf("Usage: %s <# of threads>\n", argv[0]);
24         exit(-1);
25     }
26
27     nthreads = atoi(argv[1]); // no. of threads
```

```
25
26     // allocate vector and initialize
27     tid = (pthread_t *)malloc(sizeof(pthread_t)*nthreads);
28
29     // create threads
30     for ( i = 0; i < nthreads; i++)
31         pthread_create(&tid[i], NULL, compute, (void *)i);
32
33     // wait for them to complete
34     for ( i = 0; i < nthreads; i++)
35         pthread_join(tid[i], NULL);
36
37     printf("Exiting main program\n");
38
39     return 0;
40 }
41 }
```

```
[base] mahmutunan@MacBook-Pro lecture31 % gcc pthread1.c -o exercise4 -lpthread
[base] mahmutunan@MacBook-Pro lecture31 % ./exercise4 4
Hello, I am thread 0 of 4
Hello, I am thread 1 of 4
Hello, I am thread 2 of 4
Hello, I am thread 3 of 4
Exiting main program
```

# pthread2.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4
5  int nthreads;
6
7  void *compute(void *arg) {
8      long tid = (long)arg;
9      pthread_t pthread_id = pthread_self();
10
11     printf("Hello, I am thread %ld of %d, pthread_self() = %lu (0x%lx)\n",
12           tid, nthreads, (unsigned long)pthread_id, (unsigned long)pthread_id);
13
14     return (NULL);
15 }
16
17 int main(int argc, char **argv) {
18     long i;
19     pthread_t *tid;
20     pthread_t pthread_id = pthread_self();
```

```
21
22     if (argc != 2) {
23         printf("Usage: %s <# of threads>\n", argv[0]);
24         exit(-1);
25     }
26
27     nthreads = atoi(argv[1]); // no. of threads
28
29     // allocate vector and initialize
30     tid = (pthread_t *)malloc(sizeof(pthread_t)*nthreads);
31
32     // create threads
33     for ( i = 0; i < nthreads; i++)
34         pthread_create(&tid[i], NULL, compute, (void *)i);
35
36     for ( i = 0; i < nthreads; i++)
37         printf("tid[%ld] = %lu (0x%lx)\n", i, tid[i], tid[i]);
38
39     printf("Hello, I am main thread. pthread_self() = %lu (0x%lx)\n",
40           (unsigned long)pthread_id, (unsigned long)pthread_id);
41
42     // wait for them to complete
43     for ( i = 0; i < nthreads; i++)
44         pthread_join(tid[i], NULL);
45
46     printf("Exiting main program\n");
47
48     return 0;
49 }
```

```
[base] mahmutunan@MacBook-Pro lecture31 % ./exercise5 4
tid[0] = 123145541038080 (0x70000e3ab000)
tid[1] = 123145541574656 (0x70000e42e000)
tid[2] = 123145542111232 (0x70000e4b1000)
tid[3] = 123145542647808 (0x70000e534000)
Hello, I am main thread. pthread_self() = 4365594048 (0x10435adc0)
Hello, I am thread 1 of 4, pthread_self() = 123145541574656 (0x70000e42e000)
Hello, I am thread 2 of 4, pthread_self() = 123145542111232 (0x70000e4b1000)
Hello, I am thread 0 of 4, pthread_self() = 123145541038080 (0x70000e3ab000)
Hello, I am thread 3 of 4, pthread_self() = 123145542647808 (0x70000e534000)
Exiting main program
```

# pthread3.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4
5  typedef struct foo {
6      pthread_t ptid; /* thread id returned by pthread_create */
7      int tid;         /* user managed thread id (0 through nthreads-1) */
8      int nthreads;   /* total no. of threads created */
9  } FOO;
10
11 void *compute(void *args) {
12     FOO *info = (FOO *)args;
13     printf("Hello, I am thread %d of %d\n", info->tid, info->nthreads);
14
15     return (NULL);
16 }
17
18 int main(int argc, char **argv) {
19     int i, nthreads;
20     FOO *info;
21
22     if (argc != 2) {
23         printf("Usage: %s <# of threads>\n", argv[0]);
24         exit(-1);
25     }
```

```
27     nthreads = atoi(argv[1]); // no. of threads
28
29     // allocate structure
30     info = (FOO *)malloc(sizeof(FOO)*nthreads);
31
32     // create threads
33     for ( i = 0; i < nthreads; i++) {
34         info[i].tid = i;
35         info[i].nthreads = nthreads;
36         pthread_create(&info[i].ptid, NULL, compute, (void *)&info[i]);
37     }
38
39     // wait for them to complete
40     for ( i = 0; i < nthreads; i++)
41         pthread_join(info[i].ptid, NULL);
42
43     free(info);
44     printf("Exiting main program\n");
45
46     return 0;
47 }
```

```
(base) mahmutunan@MacBook-Pro lecture31 % gcc pthread3.c -o exercise6 -lpthread
(base) mahmutunan@MacBook-Pro lecture31 % ./exercise6 4
Hello, I am thread 1 of 4
Hello, I am thread 0 of 4
Hello, I am thread 2 of 4
Hello, I am thread 3 of 4
Exiting main program
```

# Thread Synchronization using Mutexes

- We can use the mutexes provided by the Pthreads library to control access to critical sections of the program and provide synchronization across the threads.
- We will use the example of computing the sum of the elements in a vector to illustrate the use of mutexes.
- We have a vector of  $N$  elements, we would like to assign each thread to compute the partial sum of  $N/P$  elements (where  $P$  is the number of threads), then we will update the shared global variable *sum* with the partial sums using mutex locks.

- The API for the mutex lock and unlock functions are shown below:

```
#include <pthread.h>
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

```
int pthread_mutex_unlock(pthread_mutex_t  
*mutex);
```

# `pthread_sum.c`

- The *mutex* variable is of type `pthread_mutex_t` and can be initially statically by assigning the value `PTHREAD_MUTEX_INITIALIZER`.
- Note that the mutex variable must be declared in global scope since it will be shared among multiple threads.
- A mutex can also be initialized dynamically using the function `pthread_mutex_init`.
- The `pthread_mutex_destroy` function can be used to destroy the mutex that was initialized using `pthread_mutex_init`.

# pthread\_sum.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <unistd.h>
5
6 pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
7
8 double *a=NULL, sum=0.0;
9 int N, size;
10
11 void *compute(void *arg) {
12     int myStart, myEnd, myN, i;
13     long tid = (long)arg;
14
15     // determine start and end of computation for the current thread
16     myN = N/size;
17     myStart = tid*myN;
18     myEnd = myStart + myN;
19     if (tid == (size-1)) myEnd = N;
20
21     // compute partial sum
22     double mysum = 0.0;
23     for (i=myStart; i<myEnd; i++)
24         mysum += a[i];
25
26     // grab the lock, update global sum, and release lock
27     pthread_mutex_lock(&mutex);
28     sum += mysum;
29     pthread_mutex_unlock(&mutex);
30
31     return (NULL);
32 }
```

# pthread\_sum.c

```
34 | int main(int argc, char **argv) {
35 |     long i;
36 |     pthread_t *tid;
37 |
38 |     if (argc != 3) {
39 |         printf("Usage: %s <# of elements> <# of threads>\n", argv[0]);
40 |         exit(-1);
41 |     }
42 |
43 |     N = atoi(argv[1]); // no. of elements
44 |     size = atoi(argv[2]); // no. of threads
45 |
46 |     // allocate vector and initialize
47 |     tid = (pthread_t *)malloc(sizeof(pthread_t)*size);
48 |     a = (double *)malloc(sizeof(double)*N);
49 |     for (i=0; i<N; i++)
50 |         a[i] = (double)(i + 1);
51 |
52 |     // create threads
53 |     for ( i = 0; i < size; i++)
54 |         pthread_create(&tid[i], NULL, compute, (void *)i);
55 |
56 |     // wait for them to complete
57 |     for ( i = 0; i < size; i++)
58 |         pthread_join(tid[i], NULL);
59 |
60 |     printf("The total is %g, it should be equal to %g\n",
61 |           sum, ((double)N*(N+1))/2);
62 |
63 |     return 0;
64 | }
```

# pthread\_sum.c

- compile & run

```
(base) mahmutunan@MacBook-Pro lecture32 % gcc pthread_sum.c -o exercise7 -lpthread
(base) mahmutunan@MacBook-Pro lecture32 % ./exercise7 200 4
The total is 20100, it should be equal to 20100
(base) mahmutunan@MacBook-Pro lecture32 % ./exercise7 1000 4
The total is 500500, it should be equal to 500500
(base) mahmutunan@MacBook-Pro lecture32 %
```

# pthread\_sum2.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <unistd.h>
5
6  double *a=NULL, *partialsum;
7  int    N, nthreads;
8
9  void *compute(void *arg) {
10     int myStart, myEnd, myN, i;
11     long tid = (long)arg;
12
13     // determine start and end of computation for the current thread
14     myN = N/nthreads;
15     myStart = tid*myN;
16     myEnd = myStart + myN;
17     if (tid == (nthreads-1)) myEnd = N;
18
19     // compute partial sum
20     double mysum = 0.0;
21     for (i=myStart; i<myEnd; i++)
22         mysum += a[i];
23
24     partialsum[tid] = mysum;
25     return (NULL);
26 }
27 }
```

# pthread\_sum2.c

```
27
28 int main(int argc, char **argv) {
29     long i;
30     pthread_t *tid;
31     double sum = 0.0;
32
33     if (argc != 3) {
34         printf("Usage: %s <# of elements> <# of threads>\n", argv[0]);
35         exit(-1);
36     }
37
38     N = atoi(argv[1]); // no. of elements
39     nthreads = atoi(argv[2]); // no. of threads
40
41     // allocate vector and initialize
42     tid = (pthread_t *)malloc(sizeof(pthread_t)*nthreads);
43     a = (double *)malloc(sizeof(double)*N);
44     partialsum = (double *)malloc(sizeof(double)*nthreads);
45     for (i=0; i<N; i++)
46         a[i] = (double)(i + 1);
47
48     // create threads
49     for ( i = 0; i < nthreads; i++)
50         pthread_create(&tid[i], NULL, compute, (void *)i);
```

# pthread\_sum2.c

```
51
52     // wait for them to complete
53     for ( i = 0; i < nthreads; i++)
54         pthread_join(tid[i], NULL);
55
56     for ( i = 0; i < nthreads; i++)
57         sum += partialsum[i];
58
59     printf("The total is %g, it should be equal to %g\n",
60            sum, ((double)N*(N+1))/2);
61
62     free(tid);
63     free(a);
64     free(partialsum);
65
66     return 0;
67 }
68 }
```

```
[base] mahmutunan@MacBook-Pro lecture32 % gcc pthread_sum2.c -o exercise8 -lpthread
[base] mahmutunan@MacBook-Pro lecture32 % ./exercise8 1000 4
The total is 500500, it should be equal to 500500
[base] mahmutunan@MacBook-Pro lecture32 % ./exercise8 200 4
The total is 20100, it should be equal to 20100
```

# Extra exercises

```
pthread_t tid[2];
int counter;

void* trythis(void* arg)
{
    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);

    for (i = 0; i < (0xFFFFFFFF); i++)
        ;
    printf("\n Job %d has finished\n", counter);

    return NULL;
}

int main(void)
{
    int i = 0;
    int error;

    while (i < 2) {
        error = pthread_create(&(tid[i]), NULL, &trythis, NULL);
        if (error != 0)
            printf("\nThread can't be created : [%s]", strerror(error));
        i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);

    return 0;
}
```

```
[base] mahmutunan@MacBook-Pro lecture32 % gcc exercise2.c -o exercise2 -lpthread  
[base] mahmutunan@MacBook-Pro lecture32 % ./exercise2  
  
Job 1 has started  
  
Job 2 has started  
  
Job 2 has finished  
Job 2 has finished
```

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <string.h>
4
5
6     pthread_t tid[2];
7     int counter;
8     pthread_mutex_t lock;
9
10    void* trythis(void* arg)
11    {
12        pthread_mutex_lock(&lock);
13
14        unsigned long i = 0;
15        counter += 1;
16        printf("\n Job %d has started\n", counter);
17
18        for (i = 0; i < (0xFFFFFFFF); i++)
19            ;
20
21        printf("\n Job %d has finished\n", counter);
22
23        pthread_mutex_unlock(&lock);
24
25        return NULL;
26    }
```

```
28 int main(void)
29 {
30     int i = 0;
31     int error;
32
33     if (pthread_mutex_init(&lock, NULL) != 0) {
34         printf("\n mutex init has failed\n");
35         return 1;
36     }
37
38     while (i < 2) {
39         error = pthread_create(&(tid[i]),
40                               NULL,
41                               &trythis, NULL);
42         if (error != 0)
43             printf("\nThread can't be created :[%s]",
44                   strerror(error));
45         i++;
46     }
47
48     pthread_join(tid[0], NULL);
49     pthread_join(tid[1], NULL);
50     pthread_mutex_destroy(&lock);
51
52     return 0;
53 }
```

```
(base) mahmutunan@MacBook-Pro lecture32 % gcc exercise3.c -o exercise3 -lpthread  
(base) mahmutunan@MacBook-Pro lecture32 % ./exercise3  
  
Job 1 has started  
  
Job 1 has finished  
  
Job 2 has started  
  
Job 2 has finished
```

# Thread Synchronization using Mutexes

- We can use the mutexes provided by the Pthreads library to control access to critical sections of the program and provide synchronization across the threads.
- We will use the example of computing the sum of the elements in a vector to illustrate the use of mutexes.
- We have a vector of N elements, we would like to assign each thread to compute the partial sum of  $N/P$  elements (where P is the number of threads), then we will update the shared global variable *sum* with the partial sums using mutex locks.

Thread 1	Thread 2	Balance
<b>Read balance: \$1000</b>		\$1000
	<b>Read balance: \$1000</b>	\$1000
	<b>Deposit \$200</b>	\$1000
<b>Deposit \$200</b>		\$1000
<b>Update balance \$1000+\$200</b>		\$1200
	<b>Update balance \$1000+\$200</b>	\$1200

- The API for the mutex lock and unlock functions are shown below:

```
#include <pthread.h>
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

```
int pthread_mutex_unlock(pthread_mutex_t  
*mutex);
```

# `pthread_sum.c`

- The *mutex* variable is of type `pthread_mutex_t` and can be initially statically by assigning the value `PTHREAD_MUTEX_INITIALIZER`.
- Note that the mutex variable must be declared in global scope since it will be shared among multiple threads.
- A mutex can also be initialized dynamically using the function `pthread_mutex_init`.
- The `pthread_mutex_destroy` function can be used to destroy the mutex that was initialized using `pthread_mutex_init`.

# pthread\_sum.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <unistd.h>
5
6 pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
7
8 double *a=NULL, sum=0.0;
9 int N, size;
10
11 void *compute(void *arg) {
12     int myStart, myEnd, myN, i;
13     long tid = (long)arg;
14
15     // determine start and end of computation for the current thread
16     myN = N/size;
17     myStart = tid*myN;
18     myEnd = myStart + myN;
19     if (tid == (size-1)) myEnd = N;
20
21     // compute partial sum
22     double mysum = 0.0;
23     for (i=myStart; i<myEnd; i++)
24         mysum += a[i];
25
26     // grab the lock, update global sum, and release lock
27     pthread_mutex_lock(&mutex);
28     sum += mysum;
29     pthread_mutex_unlock(&mutex);
30
31     return (NULL);
32 }
```

# pthread\_sum.c

```
34 | int main(int argc, char **argv) {
35 |     long i;
36 |     pthread_t *tid;
37 |
38 |     if (argc != 3) {
39 |         printf("Usage: %s <# of elements> <# of threads>\n", argv[0]);
40 |         exit(-1);
41 |     }
42 |
43 |     N = atoi(argv[1]); // no. of elements
44 |     size = atoi(argv[2]); // no. of threads
45 |
46 |     // allocate vector and initialize
47 |     tid = (pthread_t *)malloc(sizeof(pthread_t)*size);
48 |     a = (double *)malloc(sizeof(double)*N);
49 |     for (i=0; i<N; i++)
50 |         a[i] = (double)(i + 1);
51 |
52 |     // create threads
53 |     for ( i = 0; i < size; i++)
54 |         pthread_create(&tid[i], NULL, compute, (void *)i);
55 |
56 |     // wait for them to complete
57 |     for ( i = 0; i < size; i++)
58 |         pthread_join(tid[i], NULL);
59 |
60 |     printf("The total is %g, it should be equal to %g\n",
61 |           sum, ((double)N*(N+1))/2);
62 |
63 |     return 0;
64 | }
```

# pthread\_sum.c

- compile & run

```
(base) mahmutunan@MacBook-Pro lecture32 % gcc pthread_sum.c -o exercise7 -lpthread
(base) mahmutunan@MacBook-Pro lecture32 % ./exercise7 200 4
The total is 20100, it should be equal to 20100
(base) mahmutunan@MacBook-Pro lecture32 % ./exercise7 1000 4
The total is 500500, it should be equal to 500500
(base) mahmutunan@MacBook-Pro lecture32 %
```

# pthread\_sum2.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <unistd.h>
5
6 double *a=NULL, *partialsum;
7 int N, nthreads;
8
9 void *compute(void *arg) {
10     int myStart, myEnd, myN, i;
11     long tid = (long)arg;
12
13     // determine start and end of computation for the current thread
14     myN = N/nthreads;
15     myStart = tid*myN;
16     myEnd = myStart + myN;
17     if (tid == (nthreads-1)) myEnd = N;
18
19     // compute partial sum
20     double mysum = 0.0;
21     for (i=myStart; i<myEnd; i++)
22         mysum += a[i];
23
24     partialsum[tid] = mysum;
25     return (NULL);
26 }
27 }
```

# pthread\_sum2.c

```
27
28 int main(int argc, char **argv) {
29     long i;
30     pthread_t *tid;
31     double sum = 0.0;
32
33     if (argc != 3) {
34         printf("Usage: %s <# of elements> <# of threads>\n", argv[0]);
35         exit(-1);
36     }
37
38     N = atoi(argv[1]); // no. of elements
39     nthreads = atoi(argv[2]); // no. of threads
40
41     // allocate vector and initialize
42     tid = (pthread_t *)malloc(sizeof(pthread_t)*nthreads);
43     a = (double *)malloc(sizeof(double)*N);
44     partialsum = (double *)malloc(sizeof(double)*nthreads);
45     for (i=0; i<N; i++)
46         a[i] = (double)(i + 1);
47
48     // create threads
49     for ( i = 0; i < nthreads; i++)
50         pthread_create(&tid[i], NULL, compute, (void *)i);
```

# pthread\_sum2.c

```
51
52     // wait for them to complete
53     for ( i = 0; i < nthreads; i++)
54         pthread_join(tid[i], NULL);
55
56     for ( i = 0; i < nthreads; i++)
57         sum += partialsum[i];
58
59     printf("The total is %g, it should be equal to %g\n",
60            sum, ((double)N*(N+1))/2);
61
62     free(tid);
63     free(a);
64     free(partialsum);
65
66     return 0;
67 }
68 }
```

```
[base] mahmutunan@MacBook-Pro lecture32 % gcc pthread_sum2.c -o exercise8 -lpthread
[base] mahmutunan@MacBook-Pro lecture32 % ./exercise8 1000 4
The total is 500500, it should be equal to 500500
[base] mahmutunan@MacBook-Pro lecture32 % ./exercise8 200 4
The total is 20100, it should be equal to 20100
```

# Extra exercises

```
pthread_t tid[2];
int counter;

void* trythis(void* arg)
{
    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);

    for (i = 0; i < (0xFFFFFFFF); i++)
        ;
    printf("\n Job %d has finished\n", counter);

    return NULL;
}

int main(void)
{
    int i = 0;
    int error;

    while (i < 2) {
        error = pthread_create(&(tid[i]), NULL, &trythis, NULL);
        if (error != 0)
            printf("\nThread can't be created : [%s]", strerror(error));
        i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);

    return 0;
}
```

```
[base] mahmutunan@MacBook-Pro lecture32 % gcc exercise2.c -o exercise2 -lpthread  
[base] mahmutunan@MacBook-Pro lecture32 % ./exercise2  
  
Job 1 has started  
  
Job 2 has started  
  
Job 2 has finished  
Job 2 has finished
```

```
1 #include <pthread.h>
2
3 #include <stdio.h>
4
5
6     pthread_t tid[2];
7
8     int counter;
9
10    pthread_mutex_t lock;
11
12 void* trythis(void* arg)
13 {
14
15     pthread_mutex_lock(&lock);
16
17     unsigned long i = 0;
18     counter += 1;
19     printf("\n Job %d has started\n", counter);
20
21     for (i = 0; i < (0xFFFFFFFF); i++)
22         ;
23
24     printf("\n Job %d has finished\n", counter);
25
26     pthread_mutex_unlock(&lock);
27
28     return NULL;
29 }
```

```
28 int main(void)
29 {
30     int i = 0;
31     int error;
32
33     if (pthread_mutex_init(&lock, NULL) != 0) {
34         printf("\n mutex init has failed\n");
35         return 1;
36     }
37
38     while (i < 2) {
39         error = pthread_create(&(tid[i]),
40                               NULL,
41                               &trythis, NULL);
42         if (error != 0)
43             printf("\nThread can't be created :[%s]",
44                   strerror(error));
45         i++;
46     }
47
48     pthread_join(tid[0], NULL);
49     pthread_join(tid[1], NULL);
50     pthread_mutex_destroy(&lock);
51
52     return 0;
53 }
```

```
(base) mahmutunan@MacBook-Pro lecture32 % gcc exercise3.c -o exercise3 -lpthread  
(base) mahmutunan@MacBook-Pro lecture32 % ./exercise3  
  
Job 1 has started  
  
Job 1 has finished  
  
Job 2 has started  
  
Job 2 has finished
```

# Extra Exercise

- [https://computing.llnl.gov/tutorials/pthreads/  
samples/dotprod\\_mutex.c](https://computing.llnl.gov/tutorials/pthreads/samples/dotprod_mutex.c)

# Semaphore

- The fundamental principle is this: Two or more processes can cooperate by means of simple signals, such that a process can be forced to stop at a specified place until it has received a specific signal.
- Any complex coordination requirement can be satisfied by the appropriate structure of signals.
- For signaling, special variables called semaphores are used.
- To transmit a signal via semaphore  $s$  , a process executes the primitive `semSignal(s)` .
- To receive a signal via semaphore  $s$  , a process executes the primitive `semWait(s)` ; if the corresponding signal has not yet been transmitted, the process is suspended until the transmission takes place.

# Semaphores

- While mutexes are one solution to ensure synchronization, semaphores provide an alternative and more generalized approach to establish synchronization among multiple threads.
- While mutexes provide a locking mechanism (both lock and unlock are performed by a single thread - *cooperative locks*), semaphores provide a signaling mechanism (two different threads cooperate with the wait/signal calls) to synchronize access to shared resources.
- Note that it is possible to implement a mutex using a binary semaphore.

# Semaphore

A variable that has an integer value upon which only three operations are defined:

- There is no way to inspect or manipulate semaphores other than these three operations

- 1) A semaphore may be initialized to a nonnegative integer value
- 2) The semWait operation decrements the semaphore value
- 3) The semSignal operation increments the semaphore value

- The Pthread library provides the wait and signal calls that are described in the text book. However, since Linux uses the word signal for software interrupts, the wait and signal calls are called *sem\_wait* and *sem\_post*.
- The C APIs for *sem\_wait* and *sem\_post* are:

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

- We have to include the header file *semaphore.h* to compile and link with the pthread library using -*Ipthread*.
- The *sem\_wait* call decrements the semaphore variable *sem*.
- The *sem\_wait* call returns immediately if the value of *sem* is greater than zero after decrementing the semaphore.
- If the current value of the semaphore is zero, then *sem\_wait* call will block until the semaphore values goes above zero or it is interrupted by a signal handler.
- Similarly, the *sem\_post* call increments the semaphore variable *sem* and if the value of *sem* goes above zero, it will then wake up the corresponding thread or process that was blocked in a *sem\_wait* call.

- The *sem\_init* function must be used to initialize a semaphore and the *sem\_destroy* method to destroy a semaphore.
- The C API for the *sem\_init* and *sem\_destroy* are given below:

```
#include <semaphore.h>
```

```
int sem_init(sem_t *sem, int pshared, unsigned  
int value);
```

```
int sem_destroy(sem_t *sem);
```

- The *sem\_init* call will assign a semaphore *sem* with the initial value provided and the second parameter *pshared* specifies whether the semaphore is shared between threads in with a process or it is shared between processes.

- If the value of *pshared* is zero, then the semaphore is shared between threads and the semaphore must be declared in a shared address space so that all threads can access it.
- If the value of *pshared* is nonzero, then the semaphore will be shared between processes and the semaphore must be declared in a shared memory region

# Producer/Consumer Problem

General Statement:

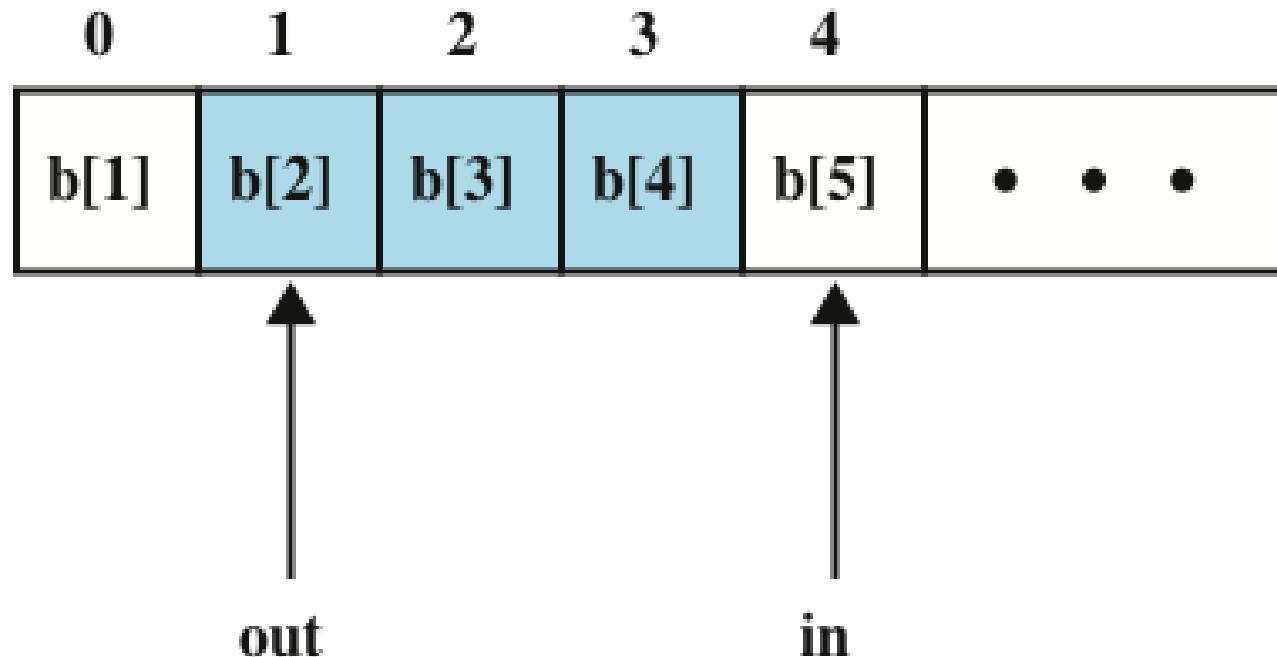
One or more producers are generating data and placing these in a buffer

A single consumer is taking items out of the buffer one at a time

Only one producer or consumer may access the buffer at any one time

The Problem:

Ensure that the producer won't try to add data into the buffer if its full, and that the consumer won't try to remove data from an empty buffer



Note: shaded area indicates portion of buffer that is occupied

**Figure 5.11 Infinite Buffer for the Producer/Consumer Problem**

- We will implement the bounded-buffer producer-consumer problem using semaphores here. In this exercise we will consider the case of a single producer and single consumer and use threads to create a producer and a consumer.
- We use the pseudo code from the textbook (Figure 5.16 on page 228) and replace semWait and semSignal with *sem\_wait* and *sem\_post*.
- This code is based on the examples provided in the classic book - UNIX Networking Programming, Volume 2 by W. Richard Stevens

# prodcons1.c

```
1  /* Solution to the single Producer/Consumer problem using semaphores.
2   This example uses a circular buffer to put and get the data
3   (a bounded-buffer).
4   Source: UNIX Network Programming, Volume 2 by W. Richard Stevens
5
6   To compile: gcc -O -Wall -o <filename> <filename>.c -lpthread
7   To run: ./<filename> <#items>
8
9   To enable printing add -DDEBUG to compile:
10  gcc -O -Wall -DDEBUG -o <filename> <filename>.c -lpthread
11 */
12
13 /* include globals */
14 #include <stdio.h>
15 #include <stdlib.h>
16 #include <pthread.h>      /* for POSIX threads */
17 #include <semaphore.h>    /* for POSIX semaphores */
```

```
18
19 #define NBUFF          10
20
21 int      nitems;           /* read-only */
22
23 struct {                  /* data shared by producer and consumer */
24     int      buff[NBUFF];
25     sem_t    mutex, nempty, nstored; /* semaphores, not pointers */
26 } shared;
27
28 void      *producer(void *), *consumer(void *);
29
30 /* end globals */
31
32 /* main program */
33 int main(int argc, char **argv)
34 {
35     pthread_t      tid_producer, tid_consumer;
36
37     if (argc != 2) {
38         printf("Usage: %s <#items>\n", argv[0]);
39         exit(-1);
40     }
41 }
```

```
41
42     nitems = atoi(argv[1]);
43
44     /* initialize three semaphores */
45     sem_init(&shared.mutex, 0, 1);
46     sem_init(&shared.nempty, 0, NBUFF);
47     sem_init(&shared.nstored, 0, 0);
48
49     /* create one producer thread and one consumer thread */
50     pthread_create(&tid_producer, NULL, producer, NULL);
51     pthread_create(&tid_consumer, NULL, consumer, NULL);
52
53     /* wait for producer and consumer threads */
54     pthread_join(tid_producer, NULL);
55     pthread_join(tid_consumer, NULL);
56
57     /* remove the semaphores */
58     sem_destroy(&shared.mutex);
59     sem_destroy(&shared.nempty);
60     sem_destroy(&shared.nstored);
61
62     return 0;
63 }
64 /* end main */
```

```
66  /* producer function */
67  void *producer(void *arg)
68  {
69      int i;
70
71      for (i = 0; i < nitems; i++) {
72          sem_wait(&shared.nempty);           /* wait for at least 1 empty slot */
73          sem_wait(&shared.mutex);
74
75          shared.buf[i % NBUFF] = i;        /* store i into circular buffer */
76 #ifdef DEBUG
77          printf("wrote %d to buffer at location %d\n", i, i % NBUFF);
78 #endif
79
80          sem_post(&shared.mutex);
81          sem_post(&shared.nstored);        /* 1 more stored item */
82      }
83      return (NULL);
84  }
85  /* end producer */
```

```
87  /* consumer function */
88  void *consumer(void *arg)
89  {
90      int i;
91
92      for (i = 0; i < nitems; i++) {
93          sem_wait(&shared.nstored);           /* wait for at least 1 stored item */
94          sem_wait(&shared.mutex);
95
96          if (shared.buf[i % NBUFF] != i)
97              printf("error: buf[%d] = %d\n", i, shared.buf[i % NBUFF]);
98 #ifdef DEBUG
99             printf("read %d from buffer at location %d\n",
100                shared.buf[i % NBUFF], i % NBUFF);
101 #endif
102
103         sem_post(&shared.mutex);
104         sem_post(&shared.nempty);           /* 1 more empty slot */
105     }
106     return (NULL);
107 }
108 /* end consumer */
```

- **gcc -O -Wall -DDEBUG prodcons1.c -lpthread**

```
$ ./a.out 20
```

```
wrote 0 to buffer at location 0
wrote 1 to buffer at location 1
wrote 2 to buffer at location 2
wrote 3 to buffer at location 3
wrote 4 to buffer at location 4
wrote 5 to buffer at location 5
wrote 6 to buffer at location 6
wrote 7 to buffer at location 7
wrote 8 to buffer at location 8
read 0 from buffer at location 0
read 1 from buffer at location 1
read 2 from buffer at location 2
read 3 from buffer at location 3
read 4 from buffer at location 4
read 5 from buffer at location 5
read 6 from buffer at location 6
read 7 from buffer at location 7
read 8 from buffer at location 8
wrote 9 to buffer at location 9
wrote 10 to buffer at location 0
wrote 11 to buffer at location 1
wrote 12 to buffer at location 2
wrote 13 to buffer at location 3
wrote 14 to buffer at location 4
wrote 15 to buffer at location 5
wrote 16 to buffer at location 6
wrote 17 to buffer at location 7
wrote 18 to buffer at location 8
read 9 from buffer at location 9
read 10 from buffer at location 0
read 11 from buffer at location 1
read 12 from buffer at location 2
read 13 from buffer at location 3
read 14 from buffer at location 4
read 15 from buffer at location 5
read 16 from buffer at location 6
read 17 from buffer at location 7
read 18 from buffer at location 8
wrote 19 to buffer at location 9
read 19 from buffer at location 9
```

- The source code is self-explanatory, we will focus on key sections of the code here.
- First, we define global variables such as the number of items (*nitems*) the producer will produce and the consumer will consume.
- Then we create a shared region, called *shared*, that will be shared between the producer and consumer threads.
- It contains the buffer shared by the producer and consumer and the three semaphores: one for the mutex lock (*mutex*), one for number of empty slots (*nempty*), and one for the number of slots filled (*nstored*) (these correspond to semaphores *s*, *e*, and *n*, respectively, from the textbook).

```

int      nitems;          /* read-only */
struct {
/* data shared by producer and consumer */
    int      buff[NBUFF];
    sem_t    mutex, nempty, nstored;
    /* semaphores */
} shared;

```

- In the main function, we read the number of items to be produced/consumed as a command-line argument and initialize the three semaphores using *sem\_init* as per the pseudocode from the textbook.

```

nitems = atoi(argv[1]);

/* initialize three semaphores */
sem_init(&shared.mutex, 0, 1);
sem_init(&shared.nempty, 0, NBUFF);
sem_init(&shared.nstored, 0, 0);

```

# Semaphores

- Generalization of the semWait and semSignal primitives
  - No other process may access the semaphore until all operations have completed

Consists of:

- Current value of the semaphore
- Process ID of the last process to operate on the semaphore
- Number of processes waiting for the semaphore value to be greater than its current value
- Number of processes waiting for the semaphore value to be zero

# Semaphores

Solaris provides classic counting semaphores with the following primitives:

- `sema_p()` Decrement the semaphore, potentially blocking the thread
- `sema_v()` Increments the semaphore, potentially unblocking a waiting thread
- `sema_try()` Decrement the semaphore if blocking is not required

# Semaphores

- User level:
  - Linux provides a semaphore interface corresponding to that in UNIX SVR4
- Internally:
  - Implemented as functions within the kernel and are more efficient than user-visible semaphores
- Three types of kernel semaphores:
  - Binary semaphores
  - Counting semaphores
  - Reader-writer semaphores

**Table 6.5**

**Linux**

**Semaphores**

<b>Traditional Semaphores</b>	
void sema_init(struct semaphore *sem, int count)	Initializes the dynamically created semaphore to the given count
void init_MUTEX(struct semaphore *sem)	Initializes the dynamically created semaphore with a count of 1 (initially unlocked)
void init_MUTEX_LOCKED(struct semaphore *sem)	Initializes the dynamically created semaphore with a count of 0 (initially locked)
void down(struct semaphore *sem)	Attempts to acquire the given semaphore, entering uninterruptible sleep if semaphore is unavailable
int down_interruptible(struct semaphore *sem)	Attempts to acquire the given semaphore, entering interruptible sleep if semaphore is unavailable; returns -EINTR value if a signal other than the result of an up operation is received
int down_trylock(struct semaphore *sem)	Attempts to acquire the given semaphore, and returns a nonzero value if semaphore is unavailable
void up(struct semaphore *sem)	Releases the given semaphore
<b>Reader-Writer Semaphores</b>	
void init_rwsem(struct rw_semaphore, *rwsem)	Initializes the dynamically created semaphore with a count of 1
void down_read(struct rw_semaphore, *rwsem)	Down operation for readers
void up_read(struct rw_semaphore, *rwsem)	Up operation for readers
void down_write(struct rw_semaphore, *rwsem)	Down operation for writers
void up_write(struct rw_semaphore, *rwsem)	Up operation for writers

(Table can be found on page 293 in textbook)

# sem\_init

- **#include <semaphore.h>**
- **int sem\_init(sem\_t \*sem, int pshared, unsigned int value);**
- Link with *-pthread*.
- **sem\_init()** initializes the unnamed semaphore at the address pointed to by *sem*. The *value* argument specifies the initial value for the semaphore. The *pshared* argument indicates whether this semaphore is to be shared between the threads of a process, or between processes. If *pshared* has the value 0, then the semaphore is shared between the threads of a process, and should be located at some address that is visible to all threads (e.g., a global variable, or a variable allocated dynamically on the heap).

Thread 1	Thread 2	data
sem_wait (&mutex);	---	0
---	sem_wait (&mutex);	0
a = data;	/* blocked */	0
a = a+1;	/* blocked */	0
data = a;	/* blocked */	1
sem_post (&mutex);	/* blocked */	1
/* blocked */	b = data;	1
/* blocked */	b = b - 1;	1
/* blocked */	data = b;	2
/* blocked */	sem_post (&mutex);	2

- We will implement the bounded-buffer producer-consumer problem using semaphores here. In this exercise we will consider the case of a single producer and single consumer and use threads to create a producer and a consumer.
- We use the pseudo code from the textbook (Figure 5.16 on page 228) and replace semWait and semSignal with *sem\_wait* and *sem\_post*.
- This code is based on the examples provided in the classic book - UNIX Networking Programming, Volume 2 by W. Richard Stevens

# prodcons1.c

```
1  /* Solution to the single Producer/Consumer problem using semaphores.
2   This example uses a circular buffer to put and get the data
3   (a bounded-buffer).
4   Source: UNIX Network Programming, Volume 2 by W. Richard Stevens
5
6   To compile: gcc -O -Wall -o <filename> <filename>.c -lpthread
7   To run: ./<filename> <#items>
8
9   To enable printing add -DDEBUG to compile:
10  gcc -O -Wall -DDEBUG -o <filename> <filename>.c -lpthread
11 */
12
13 /* include globals */
14 #include <stdio.h>
15 #include <stdlib.h>
16 #include <pthread.h>      /* for POSIX threads */
17 #include <semaphore.h>    /* for POSIX semaphores */
```

```
18
19 #define NBUFF          10
20
21 int      nitems;           /* read-only */
22
23 struct {                  /* data shared by producer and consumer */
24     int      buff[NBUFF];
25     sem_t    mutex, nempty, nstored; /* semaphores, not pointers */
26 } shared;
27
28 void      *producer(void *), *consumer(void *);
29
30 /* end globals */
31
32 /* main program */
33 int main(int argc, char **argv)
34 {
35     pthread_t      tid_producer, tid_consumer;
36
37     if (argc != 2) {
38         printf("Usage: %s <#items>\n", argv[0]);
39         exit(-1);
40     }
41 }
```

```
41
42     nitems = atoi(argv[1]);
43
44     /* initialize three semaphores */
45     sem_init(&shared.mutex, 0, 1);
46     sem_init(&shared.nempty, 0, NBUFF);
47     sem_init(&shared.nstored, 0, 0);
48
49     /* create one producer thread and one consumer thread */
50     pthread_create(&tid_producer, NULL, producer, NULL);
51     pthread_create(&tid_consumer, NULL, consumer, NULL);
52
53     /* wait for producer and consumer threads */
54     pthread_join(tid_producer, NULL);
55     pthread_join(tid_consumer, NULL);
56
57     /* remove the semaphores */
58     sem_destroy(&shared.mutex);
59     sem_destroy(&shared.nempty);
60     sem_destroy(&shared.nstored);
61
62     return 0;
63 }
64 /* end main */
```

```
66  /* producer function */
67  void *producer(void *arg)
68  {
69      int i;
70
71      for (i = 0; i < nitems; i++) {
72          sem_wait(&shared.nempty);           /* wait for at least 1 empty slot */
73          sem_wait(&shared.mutex);
74
75          shared.buf[i % NBUFF] = i;        /* store i into circular buffer */
76 #ifdef DEBUG
77          printf("wrote %d to buffer at location %d\n", i, i % NBUFF);
78 #endif
79
80          sem_post(&shared.mutex);
81          sem_post(&shared.nstored);        /* 1 more stored item */
82      }
83      return (NULL);
84  }
85  /* end producer */
```

```
87  /* consumer function */
88  void *consumer(void *arg)
89  {
90      int i;
91
92      for (i = 0; i < nitems; i++) {
93          sem_wait(&shared.nstored);           /* wait for at least 1 stored item */
94          sem_wait(&shared.mutex);
95
96          if (shared.buf[i % NBUFF] != i)
97              printf("error: buf[%d] = %d\n", i, shared.buf[i % NBUFF]);
98 #ifdef DEBUG
99             printf("read %d from buffer at location %d\n",
100                shared.buf[i % NBUFF], i % NBUFF);
101 #endif
102
103         sem_post(&shared.mutex);
104         sem_post(&shared.nempty);           /* 1 more empty slot */
105     }
106     return (NULL);
107 }
108 /* end consumer */
```

- The source code is self-explanatory, we will focus on key sections of the code here.
- First, we define global variables such as the number of items (*nitems*) the producer will produce and the consumer will consume.
- Then we create a shared region, called *shared*, that will be shared between the producer and consumer threads.
- It contains the buffer shared by the producer and consumer and the three semaphores: one for the mutex lock (*mutex*), one for number of empty slots (*nempty*), and one for the number of slots filled (*nstored*) (these correspond to semaphores *s*, *e*, and *n*, respectively, from the textbook).

```

int      nitems;          /* read-only */
struct {
/* data shared by producer and consumer */
    int      buff[NBUFF];
    sem_t    mutex, nempty, nstored;
    /* semaphores */
} shared;

```

- In the main function, we read the number of items to be produced/consumed as a command-line argument and initialize the three semaphores using *sem\_init* as per the pseudocode from the textbook.

```

nitems = atoi(argv[1]);

/* initialize three semaphores */
sem_init(&shared.mutex, 0, 1);
sem_init(&shared.nempty, 0, NBUFF);
sem_init(&shared.nstored, 0, 0);

```

- We create two separate threads, one for the producer and one for the consumer, and wait for the two threads to complete.

```
/* create one producer thread and one
consumer thread */
pthread_create(&tid_producer, NULL,
producer, NULL);
pthread_create(&tid_consumer, NULL,
consumer, NULL);

/* wait for producer and consumer threads */
pthread_join(tid_producer, NULL);
pthread_join(tid_consumer, NULL);
```

- Now let us look at the producer thread.
- It executes a loop equal to the number of items specified (*nitems*) and during each iteration of the loop, waits on the semaphore *nempty*.
- Initially *nempty* is set to *NBUFF*, so *sem\_wait* returns immediately and waits on the semaphore *mutex*.  
Since *mutex* is initially set to 1, the producer thread enters the critical section and assigns the value *i* to the buffer location  $i \% NBUFF$  and then release the *mutex* semaphore (calls *sem\_post* on the semaphore *mutex*).
- Now that there is at least one element in the buffer, it also posts *sem\_post* on the semaphore *nstored* to indicate to the consumer that there is an element in the buffer and continues with the loop.
- The producer thread terminates when the loop completes (i.e., after *nitems* iterations).
-

- /\* producer function \*/

```

void *producer(void *arg) {
    int i;

    for (i = 0; i < nitems; i++) {
        sem_wait(&shared.nempty);           /* wait for at least
1 empty slot */
        sem_wait(&shared.mutex);

        shared.buf[i % NBUFF] = i;          /* store i into
circular buffer */
#define DEBUG
        printf("wrote %d to buffer at location %d\n", i, i %
NBUFF);
#endif

        sem_post(&shared.mutex);
        sem_post(&shared.nstored);          /* 1 more stored
item */
    }

    return (NULL);
}
/* end producer */

```

- Meanwhile, the consumer thread will enter the loop and wait on the semaphore *nstored*.
- Since initially *nstored* is set to 0, this call will block and the consumer will wait until the producer posts on the semaphore *nstored*.
- When the producer posts on the semaphore *nstored*, the consumer will return from *sem\_wait* on *nstored* and invoke the *sem\_wait* on the semaphore *mutex*.
- If the producer is not in the critical section, the consumer will obtain the *mutex* semaphore, consume the buffer (we simply check if the value in the buffer match the corresponding (*loop index mod NBUFF*) and print an error message in case they don't match), and release the mutex by calling *sem\_post* on the *mutex* semaphore.
- Then the consumer thread will post the *sem\_post* on the semaphore *nempty* to indicate to the producer that now there is an empty slot. The consumer thread terminates when the loop completes (i.e., after *nitems* iterations).

- /\* consumer function \*/

```

void *consumer(void *arg) {
    int i;

    for (i = 0; i < nitems; i++) {
        sem_wait(&shared.nstored);           /* wait for at
least 1_stored item */
        sem_wait(&shared.mutex);

        if (shared.buf[i % NBUFF] != i)
            printf("error: buf[%d] = %d\n", i, shared.buf[i
% NBUFF]);
#ifdef DEBUG
        printf("read %d from buffer at location %d\n",
               shared.buf[i % NBUFF], i % NBUFF);
#endif

        sem_post(&shared.mutex);
        sem_post(&shared.nempty);           /* 1 more empty
slot */
    }

    return (NULL);
}
/* end consumer */

```

- You can compile the program with the DEBUG variable defined using -DDEBUG during compilation and see how the two threads progress. Here is a sample output when we execute the program with 20 items. You will notice that the output would be different every time you execute the program even with the same number of elements.

- **gcc -O -Wall -DDEBUG prodcons1.c -lpthread**

```
$ ./a.out 20
```

```
wrote 0 to buffer at location 0
wrote 1 to buffer at location 1
wrote 2 to buffer at location 2
wrote 3 to buffer at location 3
wrote 4 to buffer at location 4
wrote 5 to buffer at location 5
wrote 6 to buffer at location 6
wrote 7 to buffer at location 7
wrote 8 to buffer at location 8
read 0 from buffer at location 0
read 1 from buffer at location 1
read 2 from buffer at location 2
read 3 from buffer at location 3
read 4 from buffer at location 4
read 5 from buffer at location 5
read 6 from buffer at location 6
read 7 from buffer at location 7
read 8 from buffer at location 8
wrote 9 to buffer at location 9
wrote 10 to buffer at location 0
wrote 11 to buffer at location 1
wrote 12 to buffer at location 2
wrote 13 to buffer at location 3
wrote 14 to buffer at location 4
wrote 15 to buffer at location 5
wrote 16 to buffer at location 6
wrote 17 to buffer at location 7
wrote 18 to buffer at location 8
read 9 from buffer at location 9
read 10 from buffer at location 0
read 11 from buffer at location 1
read 12 from buffer at location 2
read 13 from buffer at location 3
read 14 from buffer at location 4
read 15 from buffer at location 5
read 16 from buffer at location 6
read 17 from buffer at location 7
read 18 from buffer at location 8
wrote 19 to buffer at location 9
read 19 from buffer at location 9
```

# prodcons2.c

- Solution to multiple producer and single consumer problem:

```
1  /* Solution to the Multiple Producer/Single Consumer problem using
2   semaphores. This example uses a circular buffer to put and get the
3   data (a bounded-buffer).
4   Source: UNIX Network Programming, Volume 2 by W. Richard Stevens
5
6   To compile: gcc -O -Wall -o <filename> <filename>.c -lpthread
7 */
8
9  /* include globals */
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <pthread.h>      /* for POSIX threads */
13 #include <semaphore.h>    /* for POSIX semaphores */
14
15 #define min(a, b) (((a) < (b)) ? (a) : (b))
16
17 #define NBUFF          10
18 #define MAXNTHREADS    100
```

```
19
20     int      nitems, nproducers;                                /* read-only */
21
22     struct {                                              /* data shared by producers and consumers */
23         int      buff[NBUFF];
24         int      nput;                               /* item number: 0, 1, 2, ... */
25         int      nputval;                            /* value to store in buff[] */
26         sem_t    mutex, nempty, nstored;           /* semaphores, not pointers */
27     } shared;
28
29     void      *producer(void *), *consumer(void *);
30
31 /* end globals */
32
33 /* main program */
34 int main(int argc, char **argv)
35 {
36     int          i, prodcnt[MAXNTHREADS];
37     pthread_t    tid_producer[MAXNTHREADS], tid_consumer;
38
39     if (argc != 3) {
40         printf("Usage: %s <#items> <#producers>\n", argv[0]);
41         exit(-1);
42     }
43 }
```

```
44     nitems = atoi(argv[1]);
45     nproducers = min(atoi(argv[2]), MAXNTHREADS);
46
47     /* initialize three semaphores */
48     sem_init(&shared.mutex, 0, 1);
49     sem_init(&shared.nempty, 0, NBUFF);
50     sem_init(&shared.nstored, 0, 0);
51
52     /* create all producers and one consumer */
53     for (i = 0; i < nproducers; i++) {
54         prodcount[i] = 0;
55         pthread_create(&tid_producer[i], NULL, producer, &prodcount[i]);
56     }
57     pthread_create(&tid_consumer, NULL, consumer, NULL);
58
59     /* wait for all producers and the consumer */
60     for (i = 0; i < nproducers; i++) {
61         pthread_join(tid_producer[i], NULL);
62         printf("producer count[%d] = %d\n", i, prodcount[i]);
63     }
64     pthread_join(tid_consumer, NULL);
65
66     sem_destroy(&shared.mutex);
67     sem_destroy(&shared.nempty);
68     sem_destroy(&shared.nstored);
69
70     return 0;
71 }
72 /* end main */
```

```
74     /* producer function */
75     void *producer(void *arg)
76     {
77         for ( ; ; ) {
78             sem_wait(&shared.nempty);           /* wait for at least 1 empty slot */
79             sem_wait(&shared.mutex);
80
81             if (shared.nput >= nitems) {
82                 sem_post(&shared.nempty);
83                 sem_post(&shared.mutex);|
84                 return(NULL);                  /* all done */
85             }
86
87             shared.buff[shared.nput % NBUFF] = shared.nputval;
88 #ifdef DEBUG
89             printf("wrote %d to buffer at location %d\n",
90                   shared.nputval, shared.nput % NBUFF);
91 #endif
92             shared.nput++;
93             shared.nputval++;
94
95             sem_post(&shared.mutex);
96             sem_post(&shared.nstored);        /* 1 more stored item */
97             *((int *) arg) += 1;
98         }
99     }
100    /* end producer */
```

```
101
102     /* consumer function */
103     void *consumer(void *arg)
104 {
105     int i;
106
107     for (i = 0; i < nitems; i++) {
108         sem_wait(&shared.nstored);           /* wait for at least 1 stored item */
109         sem_wait(&shared.mutex);
110
111         if (shared.buf[i % NBUFF] != i)
112             printf("error: buf[%d] = %d\n", i, shared.buf[i % NBUFF]);
113 #ifdef DEBUG
114     printf("read %d from buffer at location %d\n",
115           shared.buf[i % NBUFF], i % NBUFF);
116 #endif
117
118         sem_post(&shared.mutex);
119         sem_post(&shared.nempty);          /* 1 more empty slot */
120     }
121
122     return (NULL);
123 }
124 /* end consumer */
```

# prodcons3.c

- Solution to multiple producer and multiple consumer problem:

```
1  /* Solution to the Multiple Producer/Multiple Consumer problem using
2   semaphores. This example uses a circular buffer to put and get the
3   data (a bounded-buffer).
4   Source: UNIX Network Programming, Volume 2 by W. Richard Stevens
5
6   To compile: gcc -O -Wall -o <filename> <filename>.c -lpthread
7 */
8
9  /* include globals */
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <pthread.h>      /* for POSIX threads */
13 #include <semaphore.h>    /* for POSIX semaphores */
14
15 #define min(a, b) (((a) < (b)) ? (a) : (b))
```

```

16
17 #define NBUFF          10
18 #define MAXNTHREADS    100
19
20 int      nitems, nproducers, nconsumers;           /* read-only */
21
22 struct {
23     int      buff[NBUFF];
24     int      nput;           /* item number: 0, 1, 2, ... */
25     int      nputval;        /* value to store in buff[] */
26     int      nget;           /* item number: 0, 1, 2, ... */
27     int      ngetval;        /* value fetched from buff[] */
28     sem_t    mutex, nempty, nstored; /* semaphores, not pointers */
29 } shared;
30
31 void      *producer(void *), *consumer(void *);
32
33 /* end globals */
34
35 /* main program */
36 int main(int argc, char **argv)
37 {
38     int      i, prodcount[MAXNTHREADS], conscount[MAXNTHREADS];
39     pthread_t tid_producer[MAXNTHREADS], tid_consumer[MAXNTHREADS];
40
41     if (argc != 4) {
42         printf("Usage: %s <#items> <#producers> <#consumers>\n", argv[0]);
43         exit(-1);
44     }

```

```
45     nitems = atoi(argv[1]);
46     nproducers = min(atoi(argv[2]), MAXNTHREADS);
47     nconsumers = min(atoi(argv[3]), MAXNTHREADS);
48
49     /* initialize three semaphores */
50     sem_init(&shared.mutex, 0, 1);
51     sem_init(&shared.nempty, 0, NBUFF);
52     sem_init(&shared.nstored, 0, 0);
53
54
55     /* create all producers and all consumers */
56     for (i = 0; i < nproducers; i++) {
57         prodcount[i] = 0;
58         pthread_create(&tid_producer[i], NULL, producer, &prodcount[i]);
59     }
60
61     for (i = 0; i < nconsumers; i++) {
62         conscount[i] = 0;
63         pthread_create(&tid_consumer[i], NULL, consumer, &conscount[i]);
64     }
65
66     /* wait for all producers and all consumers */
67     for (i = 0; i < nproducers; i++) {
68         pthread_join(tid_producer[i], NULL);
69         printf("producer count[%d] = %d\n", i, prodcount[i]);
70     }
71
72     for (i = 0; i < nconsumers; i++) {
73         pthread_join(tid_consumer[i], NULL);
74         printf("consumer count[%d] = %d\n", i, conscount[i]);
75     }
```

```
74     sem_destroy(&shared.mutex);
75     sem_destroy(&shared.nempty);
76     sem_destroy(&shared.nstored);
77
78     return 0;
79 }
80 /* end main */
81
82
83 /* producer function */
84 void *producer(void *arg)
85 {
86     for ( ; ; ) {
87         sem_wait(&shared.nempty);           /* wait for at least 1 empty slot */
88         sem_wait(&shared.mutex);
89
90         if (shared.nput >= nitems) {
91             sem_post(&shared.nstored);    /* let consumers terminate */
92             sem_post(&shared.nempty);
93             sem_post(&shared.mutex);
94             return(NULL);              /* all done */
95         }
96
97         shared.buff[shared.nput % NBUFF] = shared.nputval;
98         shared.nput++;
99         shared.nputval++;
100
101         sem_post(&shared.mutex);
102         sem_post(&shared.nstored);      /* 1 more stored item */
103         *((int *) arg) += 1;
104     }
105 }
106 /* end producer */
```

```
107
108     /* consumer function */
109     void *consumer(void *arg)
110 {
111     int i;
112
113     for ( ; ; ) {
114         sem_wait(&shared.nstored);           /* wait for at least 1 stored item */
115         sem_wait(&shared.mutex);
116
117         if (shared.nget >= nitems) {
118             sem_post(&shared.nstored);
119             sem_post(&shared.mutex);
120             return(NULL);                  /* all done */
121         }
122
123         i = shared.nget % NBUFF;
124         if (shared.buf[i] != shared.ngetval)
125             printf("error: buff[%d] = %d\n", i, shared.buf[i]);
126         shared.nget++;
127         shared.ngetval++;
128
129         sem_post(&shared.mutex);
130         sem_post(&shared.nempty);          /* 1 more empty slot */
131         *((int *) arg) += 1;
132     }
133 }
134 /* end consumer */
135
```

# Semaphore for Resource Allocation

- Pool of N problems
- Resource sharing among multiple processes / uninterrupted period
- Limit the highest number of resources in use at any time
- .....

# What we have learned so far...

- ls
- cd
- rm
- ps
- cat
- touch
- .....
- .....

# Copying a file or directory

- cp [options] <source> <destination>

```
unan@unan-VirtualBox:~/lecture36$ ls
dir1  dir2  somefile.txt
unan@unan-VirtualBox:~/lecture36$ cp somefile.txt dir1
unan@unan-VirtualBox:~/lecture36$ ls
dir1  dir2  somefile.txt
unan@unan-VirtualBox:~/lecture36$ cd dir1
unan@unan-VirtualBox:~/lecture36/dir1$ ls
file1.txt  somefile.txt
unan@unan-VirtualBox:~/lecture36/dir1$ █
```

# Moving a File or Directory

- mv [options] <source> <destination>

```
unan@unan-VirtualBox:~/lecture36$ mkdir dir3
unan@unan-VirtualBox:~/lecture36$ mv dir1 dir3
unan@unan-VirtualBox:~/lecture36$ ls
dir2  dir3  somefile.txt
unan@unan-VirtualBox:~/lecture36$ █
```

```
unan@unan-VirtualBox:~/lecture36$ mv ./dir2 ./dir2new
unan@unan-VirtualBox:~/lecture36$ ls
dir2new  dir3  somefile.txt
unan@unan-VirtualBox:~/lecture36$ █
```

# nl - line numbers

- nl [-options] [path]

```
unan@unan-VirtualBox:~/lecture36$ cat somefile.txt
Mahmut Unan 55
John Smith 45
John doe 67
Julie Doe 99
Zack Zetta 88
unan@unan-VirtualBox:~/lecture36$ nl somefile.txt
    1  Mahmut Unan 55
    2  John Smith 45
    3  John doe 67
    4  Julie Doe 99
    5  Zack Zetta 88
```

# cut

- cut [-options] [path]

```
unan@unan-VirtualBox:~/lecture36$ cut -f 1 -d ' ' somefile.txt
Mahmut
John
John
Julie
Zack
unan@unan-VirtualBox:~/lecture36$ █
```

# other useful data ops

- sed
  - sed stands for **Stream Editor** and it effectively allows us to do a search and replace on our data. It is quite a powerful command but we will use it here in its basic format.
  - **sed <expression> [path]**
- uniq
  - uniq stands for **unique** and its job is to remove duplicate lines from the data. One limitation however is that those lines must be adjacent (ie, one after the other)
  - **uniq [options] [path]**

# diff

```
unan@unan-VirtualBox:~/lecture36$ cat somefile.txt
Mahmut Unan 55
John Smith 45
John doe 67
Julie Doe 99
Zack Zetta 88
unan@unan-VirtualBox:~/lecture36$ cat somefile2.txt
Mahmut Unan 55
John Smith 45
John doe 67
Julie Doe 99
Zack Zetta 88
some extra line
unan@unan-VirtualBox:~/lecture36$ diff somefile.txt somefile2.txt
5a6
> some extra line
unan@unan-VirtualBox:~/lecture36$ █
```

# grep / egrep

- grep [options] <pattern> [files]
- egrep [options] <pattern> [files]

```
unan@unan-VirtualBox:~/lecture36$ grep 'o' somefile.txt
John Smith 45
John doe 67
Julie Doe 99
unan@unan-VirtualBox:~/lecture36$ grep -c 'o' somefile.txt
3
```

# Wildcards

- \* - zero or more characters
- ? - a single character
- [ ] - a range of characters

# Networking / Communication

- SSH
  - SSH username@ip-address or hostname
- Ping
  - ping hostname
  - ping ipaddress

```
unan@unan-VirtualBox:~/lecture36$ ping www.google.com
PING www.google.com (64.233.185.99) 56(84) bytes of data.
64 bytes from yb-in-f99.1e100.net (64.233.185.99): icmp_seq=1 ttl=63 time=23.2 ms
64 bytes from yb-in-f99.1e100.net (64.233.185.99): icmp_seq=2 ttl=63 time=24.8 ms
64 bytes from yb-in-f99.1e100.net (64.233.185.99): icmp_seq=3 ttl=63 time=22.6 ms
```

- **ftp hostname**

```
unan@unan-VirtualBox:~/lecture36$ ftp ftp.javatutorialhub.com
Connected to javatutorialhub.com.
220----- Welcome to Pure-FTPd [privsep] [TLS] -----
220-You are user number 1 of 50 allowed.
220-Local time is now 15:39. Server port: 21.
220-This is a private system - No anonymous login
220-IPv6 connections are also welcome on this server.
220 You will be disconnected after 15 minutes of inactivity.
Name (ftp.javatutorialhub.com:unan): █
```

- **telnet**
  - **telnet hostname**

# Linux Administrator

- Add new user
  - `sudo adduser mynewuser`
- Disable an account
  - `sudo passwd -l mynewuser`
- Delete an account
  - `sudo userdel -r mynewuser`
- Add user to usergroup
  - `sudo usermod -a -G home mynewuser`

# finger

- This command is used to **procure information of the users on a Linux machine.**
- You can use it on both local & remote machines
- The syntax 'finger' gives data on all the logged users on the remote and local machine.
- The syntax 'finger username' specifies the information of the user.

# wget / curl

- wget <https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.17.2.tar.xz>
- curl <https://www.example.com/>

<https://curl.se/docs/manual.html>

<https://linuxize.com/post/wget-command-examples/>

# gzip / tar/ bzip2

- gzip filename
- gzip -k filename
- gunzip
- tar [options] [archive-file] [file or directory to be archived]
- bzip2 somefilename
- bunzip2 somefilename

# su / sudo

- su <username>
- su root
- sudo apt install curl

# mount

- **mount -t type <device> <directory>**

```
unan@unan-VirtualBox:~/lecture36$ mount
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
udev on /dev type devtmpfs (rw,nosuid,noexec,relatime,size=1987276k,nr_in
6819,mode=755)
devpts on /dev/pts type devpts (rw,nosuid,noexec,relatime,gid=5,mode=620,
e=000)
tmpfs on /run type tmpfs (rw,nosuid,nodev,noexec,relatime,size=403088k,mo
/dev/sda5 on / type ext4 (rw,relatime,errors=remount-ro)
securityfs on /sys/kernel/security type securityfs (rw,nosuid,nodev,noexe
ime)
```

# shred

```
unan@unan-VirtualBox:~/lecture36$ ls
dir2new  dir3  somefile2.txt  somefile.txt
unan@unan-VirtualBox:~/lecture36$ shred somefile.txt
unan@unan-VirtualBox:~/lecture36$ ls
dir2new  dir3  somefile2.txt  somefile.txt
unan@unan-VirtualBox:~/lecture36$ cat somefile.txt
♦♦U♦♦
♦Γ♦♦♦♦♦A♦♦♦e♦_♦♦♦♦♦
]♦\♦♦mB♦♦_♦R♦U♦♦♦♦.♦($♦[♦9♦♦+♦f♦D, '♦♦♦ir♦u♦@♦♦♦g♦44♦q♦♦i^♦w♦qR1♦O♦♦)♦mP^♦♦N♦i
Z♦xk♦♦♦0
♦/♦($♦fa♦♦k♦C[♦3D"♦♦\8♦"♦1♦y♦T♦n♦p♦]♦♦♦MP)♦a#P♦|R♦♦♦2r♦i+♦♦v♦t,-♦
n♦♦v♦f♦9♦~♦u♦♦♦♦|g♦♦♦♦♦CZ!G♦:♦3♦K♦e♦ZK♦♦♦F♦♦ i♦UW♦C♦W♦
♦q@♦;n♦%♦p♦♦♦s,06♦I,♦b
Bk@♦su♦♦♦Kw♦Fpo♦uB♦♦♦♦♦♦♦L♦p!ψ♦!f♦)T♦♦♦♦o"l f♦♦♦♦;♦<♦♦♦GA♦Y♦♦♦♦Üjf♦L♦♦
♦♦=E♦♦j!♦♦y♦r♦ x#♦♦♦A♦?♦q♦*♦♦nM=♦L♦♦
♦♦♦b♦jT(u8|♦(E
♦\L>,B♦Y;♦Y♦2♦s♦E06a♦tU♦♦♦Z♦♦}♦[*♦;♦mQjB♦k♦♦♦"VX@♦
j♦+h♦f♦♦♦M♦-♦♦♦♦i♦[HgN♦♦♦♦d♦_♦b♦♦♦♦@♦d,9♦
♦E♦劉♦
♦[♦j♦AAY♦♦♦P♦奸♦
```

# verbose info / delete

- `shred -u -v <fileName>`

# last

```
unan@unan-VirtualBox:~/lecture36$ last
unan      :0          :0          Thu Nov 19 22:47    gone - no logout
reboot   system boot  5.4.0-42-generic Thu Nov 19 22:47    still running
unan      :0          :0          Thu Nov 19 20:13 - crash (02:34)
reboot   system boot  5.4.0-42-generic Thu Nov 19 20:12    still running
unan      :0          :0          Mon Oct 12 00:19 - crash (38+20:53)
reboot   system boot  5.4.0-42-generic Mon Oct 12 00:18    still running
unan      :0          :0          Mon Sep 14 16:52 - crash (27+07:26)
reboot   system boot  5.4.0-42-generic Mon Sep 14 16:51    still running
unan      :0          :0          Tue Jul 28 11:49 - crash (48+05:02)
reboot   system boot  5.4.0-42-generic Tue Jul 28 11:49    still running

wtmp begins Tue Jul 28 11:49:20 2020
```

# ifconfig

- The command ifconfig stands for interface configurator. This command enables us to initialize an interface, assign IP address, enable or disable an interface. It displays route and network interface.
- You can view IP address, MAC address and MTU (Maximum Transmission Unit) with ifconfig command.
- A newer version of ifconfig is ip command. ifconfig command works for all the versions.

**Syntax:**

ifconfig

# ip

```
unan@unan-VirtualBox:~/lecture36$ ip
Usage: ip [ OPTIONS ] OBJECT { COMMAND | help }
      ip [ -force ] -batch filename
where  OBJECT := { link | address | addrlabel | route | rule | neigh | ntable |
                  tunnel | tuntap | maddress | mroute | mrule | monitor | xfrm
                  |
                  netns | l2tp | fou | macsec | tcp_metrics | token | netconf
                  |
                  ila |
                  vrf | sr | nexthop }
OPTIONS := { -V[ersion] | -s[tatistics] | -d[etails] | -r[esolve] |
              -h[uman-readable] | -iec | -j[son] | -p[retty] |
              -f[amily] { inet | inet6 | mpls | bridge | link } |
              -4 | -6 | -I | -D | -M | -B | -O |
              -l[oops] { maximum-addr-flush-attempts } | -br[ief] |
              -o[neline] | -t[imestamp] | -ts[hort] | -b[atch] [filename]
              |
              -rc[vbuf] [size] | -n[etns] name | -N[umeric] | -a[ll] |
              -c[olor] }
```