Array - 1D

Arrays is a simple data structure that is a collection of 1 or more values arranged in a linear fashion called as one dimensional array. Each element is accessed using an index or key, bu generally in C, we call it index. The index starts from 0 , not 1. If there are 10 elements in the array, the index starts at 0 to index 9.

In C/C++, all elements of an array are same data type.

For example,

unsigned int data1 [4] ;

means data1 is the name of the array, it contains 4 elements and the data type is unsigned int. Index starts from 0 to 3.

You can individually assign values to the cells. For instance,

data1 [0] = 9 ;

data1 [1] = 19 ;

data1 [2] = 30 ;

data1 [3]= 8 ;

char data2 [2] ;

means data2 is the name of the array, it contains 2 elements/cells and the data type is all char. We assign the values to a char array something like this.

data2 [0] = 'A' ;

data2 [1] = 66 ; // 'B'

Please note that the cells in an array are contiguous in physical memory too. That means the address of data1 [0] is very next to data1 [1] that is next to data1 [2] and data1 [3] . This is guaranteed by the system.

Unlike Java, out of bound runtime errors will crash your program without any message.

data1 [4] will give runtime error

data1[-1] will give runtime error too

data2 [4] will also give runtime error. So, care must be taken to check of range of indices. There is no such try and catch like you have in Java. It is your responsibility to make sure you always access cells within the range.

How do we Initialize Arrays

You can initialize an array using a for loop , say

```
int data [ 4 ] ;
for ( i = 0 ; i < 4 ; i++ )
    data [ i ] = 0 ;
```

or Initializing an Array in a Definition with an Initializer List

- The elements of an array can also be initialized when the array is defined by following the definition with an equals sign and braces, {}, containing a comma-separated list of array initializers.

```
int data [ 4 ] = {  3, 5, 8, 0 } ;
```

This is valid declaration making the first cell set to 3, the second cell set to 5, the third cell to 8, and the fourth cell set to 0.

You can also declare the above declation as

```
int data [  ] = {  3, 5, 8, 0 } ;  // it is okay,  it is missing number between  [  ]
```

Here the size of the array is defined after processing the number of values defined within the curly braces.

- If there are *fewer* initializers than elements in the array, the remaining elements are initialized to zero.

```
int data [ 4 ]  =  {  3, 5 } ;
```

in this case, the cell at index 0 is set to 3, cell at index at 1 is set to 5, but the rest of the cells (at index 2 and 3) are set to zero

- // initializes entire array to zeros

int data[4] = {0} ;

This *explicitly* initializes the cell at the first element to zero, but sets the remaining three elements (at index 1, 2, 3) also to zero because there are fewer initializers than there are elements in the array.

int data[4] = {2} ;

In this case, this *explicitly* initializes the cell at the first element to two, but sets the remaining three elements (at index 1, 2, 3) to zero , not 2. **VERY IMPORTANT**

Most programmers make a mistake that arrays are automatically set to zero. It is not, arrays are not automatically initialized to zero. You must at least initialize the first element to zero for the remaining elements to be automatically zeroed.

Do not Do THIS:

Here is code that yields compiler error

int data [4] = { 3, 5, 7, 2, 4, 9 } ;

why ? Here data is declared with four cells, but initialized using 6 values, which is incorrect. The correct way in this case is

int data [6] = { 3, 5, 7, 2, 4, 9 } ;

or a better way is

int data [ ] = { 3, 5, 7, 2, 4, 9 } ; // array is automatically declared with 6 cells.

Initialize in any order:

C99 version (or gcc extensions) allows to initialize the cells in any order

for instance ,

int a[4] = { [2] = 29, [0] = 15 } ;

where index 2 is set to 29, index 0 is set to 15 and the rest of the array is set to zero.

DECLARING ARRAYS USING CONSTANTS

Generally it is customary to use constant to declare an array. That is,

#define NUM_OF_CELLS 10

The above line can be used to declare an array like

int data [NUM_OF_CELLS] ;

In the above line , the compiler replaces NUM_OF_CELLS with 10 (textual substitution) , making the line

int data [10] ;

This is done at the preprocessor time.

Arrays-2D (declaration and initialization)

```
#define ROWS 3

#define COLS 4



int main ( )

{

int i, j;

int data[ROWS][COLS] ; // just definition

int data[ROWS][COLS] = { 0 } ; // Initialize all cells to zero


// In the next definition,  it initializes the first row with 3 4 1 2 values, second row with 4 2 1 1 ,  third row as 5 2 1 2 values.

int data[ROWS][COLS] = {

                    { 3, 4, 1, 2 },

                    { 4, 2, 1, 1 },

                    { 5, 2, 1, 2 }  // PAY ATTENTION,  NO SEMICOLON to the LAST ROW

            } ;  //  YOU NEED SEMICOLON HERE THOUGH


// ANOTHER WAY INITIALIZE,  here the first row is initialized to all zero,  second row to 5 6 7 9.  third row with 9 3 in the first and second column, zero in the third and fourth column

int data[ROWS][COLS] = { { 0 },

                    { 5, 6, 7, 9 },

                    { 9, 3 }   // PAY ATTENTION,  NO SEMICOLON to the LAST ROW

            } ;
```

How to print 2D array ?

You need nested for loop.

```
for ( i = 0 ; i < ROWS ; i++ )
 {
   for ( j = 0 ; j < COLS ; j++ )
          printf ( "%d \n",  data[ i ] [ j ] ) ;
    printf ( "\n") ;  // You need this to print the end of line after printing each column
}
```

Remember: data is still a pointer constant.

Main Points on Arrays

1. The name of the array is a pointer constant. In other words, the name itself is a pointer and is always points to the first element.

2. The name of the array cannot be made to point elsewhere

3. The size of the array is always : number of cells times the size of the data type of the array.

ie for array of N cells of type char, it is N times sizeof (char)

for array of N cells of type int , it is N times sizeof (int)

for array of N cells of type double, it is N times sizeof (double)

or The generic formula is :

int number_of_cells = sizeof (array) / sizeof (array[0]);

4. The first index is always 0

5. Cells of an array to contiguous and it is guaranteed.

6. You should ways initialize array by yourself.

7. You can print the address of each cell using the %p . For instance if data is the name,

```
printf ( " %p \n ",   &data[ 0 ] ) ;
```

8. Though you can declare array having any dimension, internally they are stored as 1D array. Your 2D indices would get converted into 1D index at runtime.

For instance for an array of [5] [4] , if we want to access [i] [j] cell, then the actual cell is at (4 - i) * 5 + j

This is because there is no such thing as 2D memory in our system. All memory locations are linear as 1D

9. Just like Java objects are passed as reference, arrays are passed to functions as reference. More on this when we deal with functions.

10. You cannot change the dimension/size of an array once it is declared. If you want to change the size, you are out of luck.

11. If data is the name of an array, the address of the first cell can be printed as &data[0] or data

Multidimensional Arrays

- Arrays in C can have multiple indices.
- A common use of multidimensional arrays is to represent tables of values consisting of information arranged in *rows* and *columns,* In image processing, we use 2D arrays extensively.
- To identify a particular table element, we must specify two indices: The first (by convention) identifies the element's *row* and the second (by convention) identifies the element's *column*.
- Tables or arrays that require two indices to identify a particular element are called two-dimensional arrays.

- Multidimensional arrays can have more than two indices.
- This declaration int data [3] [4] is a two-dimensional array, data.
- The array contains three rows and four columns, so it's said to be a 3-by-4 array.
- In general, an array with *m* rows and *n* columns is called an *m*-by-*n* array

- A multidimensional array can be initialized when it's defined, much like a one-dimensional array.
- For example, a two-dimensional array int data [2][2] could be defined and initialized with

  int data**[2][2] = { {1, 2}, {3, 4} } ;**

- The values are grouped by row in braces.
- The values in the first set of braces initialize row 0 and the values in the second set of braces initialize row 1.
- So, the values 1 and 2 initialize elements data[0][0] and data [0] [1], respectively, and the values 3 and 4 initialize elements data[1] [0] and data [1] [1], respectively.

- *If there are not enough initializers for a given row, the remaining elements of that row are initialized to 0.* Thus,

  int data [2][2] = { {1}, {3, 4} };

would initialize data [0][0] to 1, data [0][1] to 0, data[1][0] to 3 and data[1][1] to 4.

Consider these declarations

```
int data1 [ 3 ] [ 4 ] = {  { 8, 7, 6 , 5 },
                           {4, 3, 2, 1 } ,
                           { 0xa, 0xb, 0xc, 0xd }        } ;
```

The above declaration is simple,

- The definition of data1 provides 12 initializers in three sublists.
- The first sublist initializes *row 0* of the array to the values 8, 7, 6, 5 ; and the second sublist initializes *row 1* of the array to the values 4, 3, 2, 1 and the third initialized the row 2 to 0xa, 0xb, 0xc, 0xd
- There is no comma for the last row

```
int data2 [ 3 ] [ 4 ] = {   8, 7, 6 , 5 ,
                            4, 3, 2, 1
                             } ;
```

In the above declaration,

- If the braces around each sublist are removed from the data2 initializer list, the compiler initializes the elements of the first row followed by the elements of the second row and so on.
- The definition of data2 provides eight initializers.
- The initializers are assigned to the first row, then the second row.
- Any elements that do not have an explicit initializer are initialized to zero automatically, so data2 [2] [0] and other cells in the third row are initialized to 0.

Now consider this

```
int data3 [ 3 ] [ 2 ] = { { 2, 1}, { 4 }  } ;
```

The definition of data3 provides three initializers in two sublists

The first row is initialized to 2 and 1. The second row, first cell is set to 4, but the second cell is set to 0. So, is all cells in the third row, set to 0.

As I mentioned in the class, it doesn't matter if you declare arrays as multidimensional arrays, they are still stored in one linear array. The compiler lets us access them as 1D too. Thus,

- The first index of a multidimensional array is not required either, but all subsequent indices are required.

```
int data [ ] [ 3 ] = {  {  4, 3, 1 }  , { 7, 5, 6}  } ;
```

- The compiler uses the rest of the indices to determine the locations in memory of elements in multidimensional arrays.
- All array elements are stored consecutively in memory regardless of the number of indices.
- In a two-dimensional array, the first row is stored in memory followed by the second row.
- Providing the index values in a parameter declaration enables the compiler to tell the function how to locate an element in the array.

Can you tell me how many rows are there in this next definition ?

```
int data  [  ] [ 3 ] = {  {  4, 3, 1 }  , { 7, 5, 6},  { 0 },  { 0 }  } ;
```

Introduction to Functions, prototypes and math functions

Modularizing Programs in C

- Most computer programs that solve real-world problems are much larger than the programs presented so far.
- The best way to develop and maintain a large program is to construct it into smaller pieces of programs, each of which is more manageable than the original program. When I say manageable, it means smaller programs are easy to write, easy to find problems or defects or bugs.
- This technique is called divide and conquer.
- We will talk about some of the key features of the C language that facilitate the design, implementation, operation and maintenance of large programs.

- Functions are used to modularize programs
- C programs are typically written by combining new functions you write with *prepackaged* functions available in the C standard library.
- The C standard library provides a rich collection of functions for performing common *mathematical calculations, string manipulations, character manipulations, input/output*, and many other useful operations.

- The functions printf, scanf are standard library functions.
- You can write your own functions to define tasks that may be used at many points in a program.
- These are sometimes referred to as programmer-defined functions.
- The statements defining the function are written only once, and the statements are hidden from other functions.
- Functions are invoked by a function call, which specifies the function name and provides information (as arguments) that the called function needs to perform its designated task

Math Library Functions

- Math library functions allow you to perform certain common mathematical calculations.
- you should use #include <math.h>
- Functions are normally used in a program by writing the name of the function followed by a left parenthesis followed by the argument (or a comma-separated list of arguments) of the function followed by a right parenthesis.
- For example, a programmer desiring to calculate and print the square root of 900.0 you might write

```
printf("%.2f", sqrt(900.0));
```

- When this statement executes, the math library function sqrt is called to calculate the square root of the number contained in the parentheses (900.0).

- The number 900.0 is the argument of the sqrt function.
- The preceding statement would print 30.00.
- The sqrt function takes an argument of type double and returns a result of type double.
- All functions in the math library that return floating-point values return the data type double.
- Note that double values, like float values, can be output using the %f conversion specification.
- Function arguments may be constants, variables, or expressions.
- If c1 = 13.0, d = 3.0 and f = 4.0, then the statement

```
printf( "%.2f ",    sqrt (c1 + d * f) )  ;
```

- calculates and prints the square root of 13.0 + 3.0 * 4.0 = 25.0, namely 5.00.

Some examples of math functions are :

ceil (x) - rounds to smallest integer not less than x.

- ceil (7.1) returns 8.0
- ceil (-7.8) returs -7.0

fabs (x) - absolute value of x as a floating point number

- fabs (8.9) returns 8.9
- fabs (-8.9) return 8.9

exp (x) - exponential function e power x .

floor (x) - rounds to the largest number not greater than x

sin (x), cos (x) , tan (x) - Trigonometric functions

Functions in C:

- Functions allow you to modularize a program.
- All variables defined in function definitions are local variables—they can be accessed *only* in the function in which they're defined.
- Most functions have a list of parameters that provide the means for communicating information between functions.
- A function's parameters are also local variables of that function.

- There are several motivations for creating many functions in a program.
- The divide-and-conquer approach makes program development more manageable.
- Another motivation is software reusability—using existing functions as *building blocks* to create new programs. A core feature in object orient programming.
- Software reusability is a major factor in the *object-oriented programming* movement that you'll learn more about in languages derived from C, such as C++, Java and C# (pronounced "C sharp").
- We use abstraction each time we use standard library functions like printf, scanf and pow.
- A third motivation is to avoid repeating code in a program.
- Packaging code as a function allows the code to be executed from other locations in a program simply by calling the function

The code written in a function shouldn't be hundreds of lines, rather very concise and brief to perform the task. The function name should also reflect the objective of the function. For instance sin (x) , the name of the math function reflect it is computing the sin of x.

Take this simple program

```
int square ( int ) ; // prototype
```

–The int in parentheses informs the compiler that square expects to *receive* an integer value from the caller.

- the int before the name informs the compiler that square will return a value of type int

```
main ( )
{
    printf ( " The square of 10 = %d \n",  square ( 10 ) ) ;
}

int square ( int x )  // function definition matches the prototype.
{
   return x * x ;   // body of the function
}
```

Whenever you define a function like square here, you have to declare the function prior to the definition. This declaration is known as prototype. In the prototype, we declare the signature. When the compiler sees the prototype declared, it will remember that the definition will come later so it won't complain when the function is called in the main function. Prototype is essential when the function is defined later than the call or the function is defined in another file.

- The compiler refers to the function prototype to check that any calls to square contain the *correct return type*, the *correct number of arguments*, the *correct argument types*, and that the *arguments are in the correct order*.
- The format of a function definition is

```
 return-value-type function-name(parameter-list)
{
  definitions
  statements
}
```

- The *function-name* is any valid identifier.
- The *return-value-type* is the data type of the result returned to the caller.

- The *return-value-type* void indicates that a function does not return a value.
- Together, the *return-value-type, function-name* and *parameter-list* are sometimes referred to as the function header.
- The *parameter-list* is a comma-separated list that specifies the parameters received by the function when it's called.
- If a function does not receive any values, *parameter-list* is void.
- A type must be listed explicitly for each parameter.

- The *definitions* and *statements* within braces form the function body, which is also referred to as a block.
- Variables can be declared in any block, and blocks can be nested

Return values;

- There are three ways to return control from a called function to the point at which a function was invoked.
- If the function does *not* return a result, control is returned simply when the function-ending right brace is reached, or by executing the statement

    return;

- If the function *does* return a result, the statement

    return *expression;*

returns the value of *expression* to the caller

Prototype

- An important feature of C is the function prototype.
- This feature was borrowed from C++.
- The compiler uses function prototypes to validate function calls.
- Early versions of C did *not* perform this kind of checking, so it was possible to call functions improperly without the compiler detecting the errors.
- Such calls could result in fatal execution-time errors or nonfatal errors that caused subtle, difficult-to-detect problems.

- Take this function prototype for maximum

// **function prototype**

int maximum(int x, int y);

- It states that maximum takes two arguments of type int and returns a result of type int.

the actual function is written as

```
int maximum( int x, int  y )
{
   if ( x >= y ) return x ; else return y ;
}
```

- Notice that the function prototype is the same as the first line of maximum's function definition.
- A function call that does not match the function prototype is a compilation error.
- An error is also generated if the function prototype and the function definition disagree.
- For example, if the function prototype had been written

void maximum(int x, int y);

- the compiler would generate an error because the void return type in the function prototype would differ from the int return type in the function header.

Argument Coercion and "Usual Arithmetic Conversion Rules"

- Another important feature of function prototypes is the coercion of arguments, i.e., the forcing of arguments to the appropriate type.
- For example, the math library function sqrt can be called with an integer argument even though the function prototype in <math.h> specifies a double parameter, and the function will still work correctly.
- The statement

printf(**"%.3f\n", sqrt(4));**

correctly evaluates sqrt(4) and prints the value 2.000.

- The function prototype causes the compiler to convert a *copy* of the integer value 4 to the double value 4.0 before the *copy* is passed to sqrt.
- In general, *argument values that do not correspond precisely to the parameter types in the function prototype are converted to the proper type before the function is called*.

- These conversions can lead to incorrect results if C's usual arithmetic conversion rules are not followed.
- These rules specify how values can be converted to other types without losing data.
- In our sqrt example above, an int is automatically converted to a double without changing its value. We saw this a type promotion.
- However, a double converted to an int *truncates* the fractional part of the double value, thus changing the original value.
- Converting large integer types to small integer types (e.g., long to short) may also result in changed values.
- The usual arithmetic conversion rules automatically apply to expressions containing values of two or more data types (also referred to as mixed-type expressions) and are handled for you by the compiler.
- In a mixed-type expression, the compiler makes a temporary copy of the value that needs to be converted then converts the copy to the "highest" type in the expression—the original value remains unchanged.
- The usual arithmetic conversion rules for a mixed-type expression containing at least one floating-point value are:
  > –If one of the values is a long double, the other is converted to a long double.
  > –If one of the values is a double, the other is converted to a double.
  > –If one of the values is a float, the other is converted to a float.

Function : variables - types and duration

- Earlier, we used identifiers for variable names.
- The attributes of variables include name, type, size and value.
- Actually, each identifier in a program has other attributes, including storage class, storage duration, scope and linkage.
- C provides the storage class specifiers: auto, register, extern and static.
- An identifier's storage class determines its storage duration, scope and linkage.
- An identifier's storage duration is the period during which the identifier exists *in memory*.
- Some exist briefly, some are repeatedly created and destroyed, and others exist for the program's entire execution

Scope:

- An identifier's scope is where the identifier can be referenced in a program.
- Some can be referenced throughout a program like global variables, others from only portions of a program like local variables in a function

Linkage:

- An identifier's linkage determines for a multiple-source-file program whether the identifier is known only in the current source file or in any source file with proper declarations.

Duration:

- The storage-class specifiers can be split into automatic storage duration and static storage duration.
- Keyword auto is used to declare variables of automatic storage duration.
- Variables with automatic storage duration are created when the block in which they're defined is entered; they exist while the block is active, and they're destroyed when the block is exited.

Local Variables

- Only variables can have automatic storage duration.

- A function's local variables (those declared in the parameter list or function body) normally have automatic storage duration.
- Keyword auto explicitly declares variables of automatic storage duration.
- Local variables have automatic storage duration by *default*, so keyword auto is rarely used.
- For the remainder of the text, we'll refer to variables with automatic storage duration simply as automatic variables

Static Storage Class

- Keywords extern and static are used in the declarations of identifiers for variables and functions of static storage duration.
- Identifiers of static storage duration exist from the time at which the program begins execution until the program terminates.
- For static variables, storage is allocated and initialized *only once*, *before* the program begins execution.
- For functions, the name of the function exists when the program begins execution.
- There are several types of identifiers with static storage duration: *external identifiers* (such as global variables and function names) and local variables declared with the storage-class specifier static.
- Global variables and function names are of storage class extern by default.
- Global variables are created by placing variable declarations *outside* any function definition, and they retain their values throughout the execution of the program.
- Global variables and functions can be referenced by any function that follows their declarations or definitions in the file.
- This is one reason for using function prototypes—when we include stdio.h in a program that calls printf, the function prototype is placed at the start of our file to make the name printf known to the rest of the file.
- Local variables declared with the keyword static are still known only in the function in which they're defined, but unlike automatic variables, static local variables retain their value when the function is exited.
- The next time the function is called, the static local variable contains the value it had when the function last exited.
- The following statement declares local variable count to be static and initializes it to 1.
- **static int count = 1;**
- All numeric variables of static storage duration are initialized to zero by default if you do not explicitly initialize them.

Here is a simple program using static variable

```
#include <stdio.h>
int getMyAge ( )
{
  static int  age = 80 ;
       //   age is static , it is initialized to 80 when the function is called for the first time.

  age++;   //   age is incremented to 81

  return age;  //   81 is returned.
}


main ( )
{

      printf (" Last year, My Age was %d \n", getMyAge ( )  )  ;
      printf ("This  year my age is %d \n", getMyAge ( ) ) ;
      printf ("Next year, my age would be %d \n", getMyAge ( ) ) ;

}
```

Pass by Value and Reference

Function - Passing variables by value

- In many programming languages, there are two ways to pass arguments—pass-by-value and pass-by-reference.
- When arguments are *passed by value*, a *copy* of the argument's value is made and passed to the called function.
- Changes to the copy do *not* affect an original variable's value in the caller.
- When an argument is passed by reference, the caller allows the called function to modify the original variable's value.
- Pass-by-value should be used whenever the called function does not need to modify the value of the caller's original variable.
- When we pass the address of a variable, we call the variable is passed by reference. When we pass the value of a variable, we call the variable is passed by value.
- A good example of pass by reference is scanf function we used a lot. The function requires the address of the variable. When the function returns, it updates ie modifies the value of the variable.
- a good example of pass by value is the printf function we used a lot. The function requires just the value of the variable to be printed. It serve no purpose for it to have the address of a variable to print.

Here is the code snippet of a function swap

```
void swap (int , int  ) ;   // prototype of a function.  It takes two parameters of type int.
//  it returns nothing.  so we say void

void swap ( int a , int b )
{
   int temp = a ;
   a = b ;
```

```
   b = temp ;
   printf (" swap : The numbers are %d and %d \n ",  a,  b ) ;
}

int main ( )
{
     int x, y;
     printf ("Enter two numbers \n");
     scanf("%d %d", &x, &y);   // variables are Passed by reference
     swap (x, y) ;   // variables are passed by value
     printf ("main after calling swap:  %d and %d \n", x, y );
    return 0 ;
}
```

In the function swap, we swap the values, but the values in the main function are still not swapped. This is because we only sent the values to the function swap.

Function - Passing variables by reference

Example :

```
#include <stdio.h>
// function swap_1: pass by reference or pass address
void swap_1 ( int *, int * ); // prototype

void swap_1 ( int *x, int *y )
{
```

```
    int temp ;
    temp = *x ;
    *x = *y ;
    *y = temp ;
}

int main ( )
{
int a = 10, b = 20 ;
printf ( "Before swapping a=%d b=%d \n", a, b);
swap_1( &a, &b ) ; // watch passing the address
printf ( "After  swapping a=%d b=%d \n", a, b);
}
```

```
$ gcc ref1.c
$ ./a.out
Before swapping a=10 b=20
After  swapping a=20 b=10
```

Function - Passing Arrays

When you pass an array, you always have to pass the number of elements in that array to the function. For instance

```
void print_array ( int data [ ] ,  int count )
{
   int i ;
   // to traverse the array , you use the count variable in a loop
  for ( i = 0 ; i < count ; i++ )
    printf ( " %d  \n",   data [ i ]  )  ;
}
int main ( )
{
  int myDataArray [ 10 ] = { 4 } ;  //  assign 4 to index 0,  0 to rest of the cells
  print_array ( myDataArray ,  10 ) ;
 }
```

Shall we do another example with passing arrays ?

```
void bubble_sort ( int data [  ]  , int count )
{
    int i, j , temp ;
    for ( i = 0 ; i < count ; i++ )
        for ( j = 0 ; j < count - 1 ; j++ )
                if (  data [ j ] >  data [ j + 1 ] )
                     {
                         temp = data [ j ] ;
                         data [ j ] = data [ j + 1 ] ;
                         data [ j + 1 ] = temp ;
                      }
}
```

```
int main ( )
{
     int sort_data [ ] = {  5, 2, 9, 8, 3 }  ;
    bubble_sort ( sort_data, 5 )  ;
     print_array ( sort_data , 5 )   ;
}
```

Pass Arrays to Functions

- To pass an array argument to a function, specify the array's name without any brackets and also pass the number of cells in the array.
- For example, if array WeeklySalaries has been defined as

```
int  WeeklySalaries[ 52 ];
```

the function call

```
ComputeAnnualSalary( WeeklySalaries,  52)
```

passes array WeeklySalaries and its size to ComputeAnnualSalary.

- C automatically passes arrays to functions *by reference* —the called functions can modify the element values in the callers' original arrays.
- We know the name of the array evaluates to the address of the first element of the array.
- Because the starting address of the array is passed, the called function knows precisely where the array is stored.
- Therefore, when the called function modifies array elements in its function body, it's modifying the actual elements of the array in their *original* memory locations.

- For a function to receive an array through a function call, the function's parameter list must specify that an array will be received.
- For example, the function header for function ComputeAnnualSalary (that we called earlier in this section) might be written as
- **void** ComputeAnnualSalary **(int** data **[ ] , int size)**

indicating that ComputeAnnualSalary expects to receive an array of integers in parameter data and the number of array elements in parameter size.

- The size of the array is not required between the array brackets. See we have data with empty [ ]
- If it's included, the compiler checks that it's greater than zero, then ignores it.
- Specifying a negative size is a compilation error.
- Because arrays are automatically passed by reference, when the called function uses the array name data, it will be referring to the array in the caller (array WeeklySalaries in the preceding call)

```
void ComputeAnnualSalary   (  int data [  ] , int size )
{
        for ( i = 0 ; i < size ; i++ )
             annualSalary += data [ i ] ;
        printf ( "The annual Salary is %d \", annualSalary ) ;
}
```

Version 2: We know arrays are just like pointers, we could rewrite the function as

```
void ComputeAnnualSalary   (  int * data  , int size )
{
        for ( i = 0 ; i < size ; i++ )
             annualSalary +=  * ( data + i )  ; // NOTE IT.
        printf ( "The annual Salary is %d \", annualSalary ) ;
}
```

How individual cells are passed ?

- Although entire arrays are passed by reference, individual array cell elements are passed by value exactly as simple variables are.
- To pass an cell element of an array to a function, use the indexed name of the array element as an argument in the function call

```
PrintCellValue (  int cellValue ) {
   printf ( " The value is %d \n ", cellValue ) ;
}
```

.

```
//  The caller would call this function as

   for ( i = 0 ; i < 52 ; i ++ )

          PrintCellValue (  WeeklySalaries [ i ]  ) ;
```

Executing functions via Pointers

Now we will declare a pointer to a function , and execute the function using the pointer.

There are three steps in executing a function using a pointer.

Step 1 : Declare the pointer stating it points to a function and define the parameters and return types of the function.

for example, consider the following pointer definition.

```
int  ( * ptr )  ( int, int ) ;   // STEP 1
```

ptr is a pointer, it points to some function, the function takes two parameters of type int and returns a type of int.

Step 2 : Point the pointer to the actual function

```
  ptr = add ;      // STEP 2
```

add is a function that takes two parameters. Our ptr is now pointing to the function add.

Step 3: call the pointer with the argument values to execute the function.

```
        ptr ( 10, 5 ) ;    // STEP 3
```

now we are actually calling the function add passing two variables : 10 and 5

// DEMO EXAMPLE OF A POINTER TO A FUNCTION

```
#include <stdio.h>

int add ( int x , int y )

{
```

```
  return x + y ;
}

int sub ( int x, int y )
{
   return x - y ;
}

int main ( )
{
    // define the pointer that points to a function which takes two parameters and returns an int
    int  ( * ptr )  ( int, int ) ;   // STEP 1

   //  if we remove the (    ) around ptr,  like
   // int  *ptr  ( int , int )
  //  then  ptr is a function that takes two parameters and returns int pointer.  That is not what you want.  Putting ( ) around would make ptr is pointer to a function.

  // NOW YOU KNOW how to define a pointer to a function.

   ptr = add ;      // STEP 2
  int sum = ptr ( 10, 5 ) ;    // STEP 3


   ptr = sub ;     // STEP 2
   int minus = ptr ( 10 , 5 ) ;  // STEP 3
```

```
  printf ( "%d %d \n", sum, minus);



}


$ gcc ptr.c
$ ./a.out
15 5
```

Designing function parameters

When you write function, you have to ask :

What kind of variables should be function take as input - pass by value

what kind of variables should this function take as output/input - pass by reference

What kind of values should this function return: int, float, char, double , short, ...

what kind of address should this function return : char *, int *, float *, ...

Why do functions define const in front of the variables

Without asking these questions, you cannot write any functions. So, research these functions before attempting to complete the function.

Some example of function definitions.

A function may take a variable as reference as a parameter. When that happens, it predominantly means the function will change the value. Passing by reference means you are actually passing an address of that variable. An example:

char *strcat(char *dest, const char *src);

The const means the function will not change the value of src. The contents pointed by dest is changed by copying src to dest.

Another example :

char *strcpy(char *dest, const char *src);

The purpose seems to be same as strcat. dest is changed so it is passed as reference.

A function may take a variable as value as a parameter. When that happens, it predominanly means the function will not change the value, it merely wants a copy. Here is an example:

int abs(int j);
long int labs(long int j);
long long int llabs(long long int j);

A function may take have two or more variables as parameters. Some variables may be passed by reference, some variables may be passed by value.

char *strncpy(char *dest, const char *src, size_t n);

- n is passed by value

- dest is defined as pointers

A function may return a value.

int rand(void);

A function may not return a value - void.

void srand(unsigned int seed);

A function may return an address

char *strtok(char *str, const char *delim);

A function may return an address of type void *

void *malloc(size_t size);

One bad programming: DO NOT RETURN THE ADDRESS OF LOCAL VARIABLES. Why ?

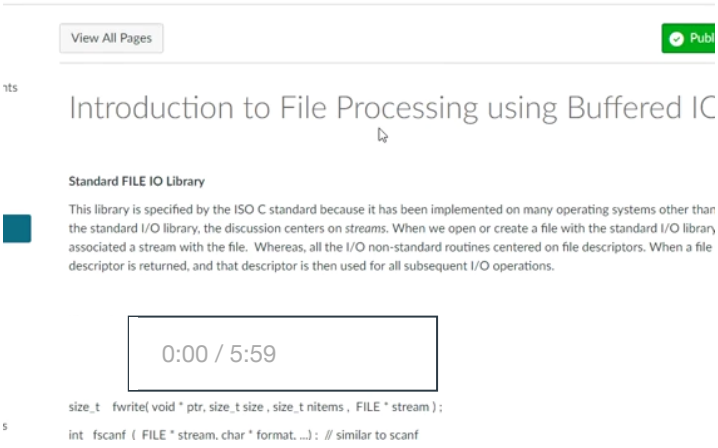
```
int * test (  )
{
   int x = 10;
   return &x ;  // why is this BAAAAAAAAD
}
```

Buffered IO

Standard FILE IO Library

This library is specified by the ISO C standard because it has been implemented on many operating systems other than the UNIX System. With the standard I/O library, the discussion centers on *streams*. When we open or create a file with the standard I/O library, we say that we have associated a stream with the file. Whereas, all the I/O non-standard routines centered on file descriptors. When a file is opened, a file descriptor is returned, and that descriptor is then used for all subsequent I/O operations.

Here is the video:



The main functions that we will learn are:

Here is the list of functions that use file stream to open, read, write and close:

size_t fread(void * ptr, size_t size , size_t nitems , FILE * stream) ;

size_t fwrite(void * ptr, size_t size , size_t nitems , FILE * stream) ;

int fscanf (FILE * stream, char * format, ...) ; // similar to scanf

int fprintf (FILE * stream, char * format, ...) ; // similar to printf

char * fgets (char * str, int size, FILE * stream) ;

int fputs (char * s, FILE * stream) ;

int sscanf (char * s, char *format, ...) ; // input is a buffer, not from file nor stdin

int sprintf (char * str, char * format, ...) ; // output to a buffer, not to a file nor to stdout

When we open a stream, the standard I/O function fopen returns a pointer of type FILE. This is a structure that contains all the information required by the standard I/O library to manage the stream.

To reference the stream, we pass its FILE pointer as an argument to each standard I/O function. We'll refer to a pointer to a FILE object, the type FILE *, as a *file pointer*.

Additionaly, when a program is launched, three streams are predefined and automatically available:: stdin, stdout, and stderr. The file pointers are defined in the <stdio.h> header.

The unistd.h header also defines similar file descriptors as : STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO.

Some facts about standard buffered IO functions in C are :

- C views each file simply as a sequential stream of bytes
- Each file ends either with an end-of-file marker or at a specific byte number recorded in a system-maintained, administrative data structure.
- When a file is opened, a stream is associated with it.
- Three files and their associated streams are automatically opened when program execution begins—the standard input, the standard output and the standard error. The standard input, standard output and standard error are manipulated using file pointers stdin, stdout and stderr.
- stdin is a FILE pointer defined in /usr/include/stdio.h , it is a data going into the program (usually from a device such as keyboard from the user). In our case scanf ("%d", &count) and scanf (stdin, "%d", &count) are same statements.

  stdout is a FILE pointer defined in /usr/include/stdio.h , it is a data going out from the program (usually to the display).
- For us printf ("%d \n", count) and printf (stdout, "%d \n", count) are same statements.

stderr is a FILE pointer , it is error whose default output is stdout, but could be redirected to the file or elsewhere.

- Streams provide communication channels between files and programs.
- For example, the standard input stream enables a program to read data from the keyboard, and the standard output stream enables a program to print data on the screen.
- Opening a file returns a pointer to a FILE structure (defined in <stdio.h>) that contains information used to process the file.
- This structure includes a file descriptor, i.e., an index into an operating system array called the open file table.
- Each array element in this open file table, contains a file control block (FCB) that the operating system uses to administer a particular file.
- The standard library provides many functions for reading data from files and for writing data to files.
  - Function fgetc, like getchar, reads one character from a file. This function receives a FILE pointer as an argument for the file from which a character will be read.
  - The call fgetc(stdin) reads one character from stdin—the standard input. This call is equivalent to the call getchar().
  -
  - Function fputc, like putchar, writes one character to a file.
  - Function fputc receives as arguments a character to be written and a pointer for the file to which the character will be written.
  - The function call fputc('a', stdout) writes the character 'a' to stdout—the standard output. This call is equivalent to putchar('a').
  - Several other functions used to read data from standard input and write data to standard output have similarly named file-processing functions. The fgets and fputs functions, for example, can be used to *read a line from a file* and *write a line to a file*, respectively.
- We will also talk about the file-processing equivalents of functions scanf and printf—fscanf and fprintf.
- C imposes no structure on a file. Thus, notions such as a record of a file do not exist as part of the C language.
-

Standard IO Library functions

Reading and Writing a Stream

Once we open a stream, we can choose from among three types of unformatted I/O:

1. Character-at-a-time I/O. We can read or write one character at a time, with the standard I/O functions handling all the buffering, if the stream is buffered.
2. Line-at-a-time I/O. If we want to read or write a line at a time, we use fgets and fputs. Each line is terminated with a newline character, and we have to specify the maximum line length that we can handle when we call fgets.
3. Direct I/O. This type of I/O is supported by the fread and fwrite functions. For each I/O operation, we read or write some number of objects, where each object is of a specified size. These two functions are often used for binary files where we read or write a structure with each operation.

Input Functions to read Character-at-a-time

Three functions allow us to read one character at a time.

#include <stdio.h>

int getc(FILE *fp);

int fgetc(FILE *fp);

int getchar(void);

All three return: next character if OK, EOF on end of file or error

The function getchar is defined to be equivalent to getc(stdin). Getc is implemented as a macro while fgetc is a function.

These three functions return the next character as an unsigned char converted to an int. The reason for specifying unsigned is so that the high-order bit, if set, doesn't cause the return value to be negative. The reason for requiring an integer return value is so that all possible character values can be returned, along with an indication that either an error occurred or the end of file has been encountered. The constant EOF in <stdio.h> is required to be a negative value. Its value is often -1.

This representation also means that we cannot store the return value from these three functions in a character variable and later compare this value with the constant EOF. Note that these functions return the same value whether an error occurs or the end of file is reached. To distinguish between the two, we must call either ferror or feof.

#include <stdio.h>

int ferror(FILE *_fp_);

int feof(FILE *_fp_);

Both return: nonzero (true) if condition is true, 0 (false) otherwise

Output Functions to write Character-at-a-time:

Output functions are available that correspond to each of the input functions we've already described.

#include <stdio.h>

int putc(int _c_, FILE *_fp_);

int fputc(int _c_, FILE *_fp_);

int putchar(int _c_);

All three return: _c_ if OK, EOF on error

As with the input functions, putchar(c) is equivalent to putc(c, stdout), and

putc can be implemented as a macro, whereas fputc cannot be implemented as a

macro.

Read Line-at-a-Time I/O

Line-at-a-time input is provided by the two functions, fgets and gets.

#include <stdio.h>

char *fgets(char *_buf_, int _n_, FILE * _fp_);

char *gets(char *_buf_);

Both return: buf if OK, NULL on end of file or error

Both specify the address of the buffer to read the line into. The gets function reads from standard input, whereas fgets reads from the specified stream. With fgets, we have to specify the size of the buffer, *n*. This function reads up through and including the next newline, but no more than *n * 1* characters, into the buffer. The buffer is terminated with a null byte. If the line, including the terminating newline, is longer than *n * 1*, only a partial line is returned, but the buffer is always null terminated. Another call to fgets will read what follows on the line. The gets function should never be used. The problem is that it doesn't allow the caller to specify the buffer size.

Even though ISO C requires an implementation to provide gets, you should use fgets instead.

Write Line-at-a-time is done using fputs and puts.

int fputs(const char *_str_, FILE *_fp_);

int puts(const char *_str_);

Both return: non-negative value if OK, EOF on error

(https://sierra.instructure.com/courses/322669/pages/sscanf-sprintf?module_item_id=6013138)

The function fputs writes the null-terminated string to the specified stream. The null byte at the end is not written. Note that this need not be line-at-a-time output, since the string need not contain a newline as the last non-null character. Usually, this is the case — the last non-null character is a newline—but it's not required. The puts function writes the null-terminated string to the standard output, without writing the null byte. But puts then writes a newline character to the standard output.

The puts function is not unsafe, like its counterpart gets. Nevertheless, we'll avoid using it, to prevent having to remember whether it appends a newline. If we always use fgets and fputs, we know that we always have to deal with the newline character at the end of each line.

Binary I/O

If we're doing binary I/O, we often would like to read or write an entire structure at a time. To do this using getc or putc, we have to loop through the entire structure, one byte at a time, reading or writing each byte. We can't use the line-at-a-time functions, since fputs stops writing when it hits a null byte, and there might be null bytes within the structure. Similarly, fgets won't

work correctly on input if any of the data bytes are nulls or newlines. Therefore, the following two functions are provided for binary I/O.

```
#include <stdio.h>
size_t fread (void *ptr, size_t size, size_t nobj, FILE *fp);
size_t fwrite (const void *ptr, size_t size, size_t nobj,FILE *restrict fp);
```

Both return: number of objects read or written

These functions have two common uses:

1. Read or write a binary array. For example, to write elements 2 through 5 of a floating-point array, we could write

```
float data[10];
if (fwrite(&data[2], sizeof(float), 4, fp) != 4)
err_sys("fwrite error");
```

Here, we specify *size* as the size of each element of the array and *nobj* as the number of elements.

2. Read or write a structure. For example, we could write

```
struct {
    unsigned char age ;
    char name[NAMESIZE];
} person  ;
if ( fwrite(&person  , sizeof(person  ),  1 , fp )  != 1 )
     printf ("fwrite error \n");
```

Here, we specify *size* as the size of structure and *nobj* as 1 (the number of objects to write).

Both fread and fwrite return the number of objects read or written. For the read case, this number can be less than *nobj* if an error occurs or if the end of file is

encountered. In this situation, ferror or feof must be called. For the write case, if the return value is less than the requested *nobj*, an error has occurred.

Function file FOPEN AND MODES TO OPEN, and FCLOSE

Function fopen

Standard I/O routines do not operate directly on file descriptors. Instead, they use their own unique identifier, known as the file pointer. Inside the C library, the file pointer maps to a file descriptor. The file pointer is represented by a pointer to the FILE typedef, which is defined in .

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

The syntax is:

```
FILE *fopen(const char *path, const char *mode);
```

In Standard IO library, a FILE pointer is returned. This pointer is a representative of a stream. We say a stream is attached interfacing the user program and the actual file.

The actual path to the structure definition of FILE is : /usr/include/libio.h

The actual structure

```
struct _IO_FILE {


};
```

Streams can : input, output and/or both

Modes for reading, writing and appending :

- Files may be opened in one of several modes .
- To create a file, or to discard the contents of a file before writing data, open the file for writing using mode set to "w"
- To read an existing file, open it for reading ("r").
- To add records to the end of an existing file, open the file for appending with mode set to "a"
- To open a file so that it may be written to and read from, open the file for updating in one of the three update modes—"r+", "w+" or "a+".
- Mode "r+" opens an existing file for reading and writing.
- Mode "w+" creates a file for reading and writing.
- If the file already exists, it's opened and its current contents are discarded.
- Mode "a+" opens a file for reading and writing—all writing is done at the end of the file.
- If the file does not exist, it's created.
- Each file open mode has a corresponding binary mode (containing the letter b) for manipulating binary files.

Upon success, fopen( ) returns a valid FILE pointer. On failure, it returns NULL, and sets errno appropriately.

Function FCLOSE:

The fclose( ) function closes a given stream:

int fclose (FILE *stream);

Any buffered and not-yet-written data is first flushed. On success, fclose( ) returns 0. On failure, it returns EOF and sets errno appropriately.

Unbuffered IO

Unbuffered File IO

There are functions available for file I/O—open a file, read a file, write a file, and so on. Most file I/O on a UNIX system can be performed using only five functions: open, read, write, lseek, and close. These functions are referred to as unbuffered IO. The term *unbuffered* means that each read or write invokes a system call in the kernel. We talked about the system call steps before.

NOTE: These unbuffered I/O functions are not part of ISO C, but are part of POSIX.1.

Systemwide, kernel has a data structure that stores list of open file descriptors mapping to a file name. All open files are referred to by file descriptors. A file descriptor is a non-negative integer that the kernel uses to identify the files accessed by a process. When we open an existing file or create a new file, the kernel returns a file descriptor to the process. When we want to read or write a file, we identify the file with the file descriptor that was returned by open as an argument to either read or write.

NOTE: Remember , in buffered IO, we refer open files as streams using FILE data type. Whereas in unbuffered IO, we use file descriptors.

UNIX System shells associate file descriptor 0 (STDIN_FILENO) with the standard input of a process, file descriptor 1 (STDOUT_FILENO) with the standard output, and file descriptor 2 (STDERR_FILENO) with the standard error. This convention is used by the shells and many applications, many applications would break if these associations weren't followed.

File Descriptor	**Purpose**	**POSIX Name**	**Stdio stream**
0	standard input	STDIN_FILENO	**stdin**

1	standard output	STDOUT_FILENO **stdout**
2	standard error	STDERR_FILENO **stderr**

These constants are defined in the <unistd.h> header.

The following are the five key system calls for performing file I/O. You have to include this include file

#include <unistd.h>

The five system calls we read are:

- **open**
- **lseek**
- **write**
- **read**
- **close**

OPEN

The open() system call either opens an existing file or creates and opens a new file.

#include <sys/stat.h>

#include <fcntl.h>

int open(const char *_path_, int _oflag_, ... /* mode_t _mode_ */);

1. The *path* parameter is the name of the file to open or create.

2. The oflag argument takes various values, we can bitwise OR them together, notable and frequently used are

- O_RDONLY Open for reading only.
- O_WRONLY Open for writing only.
- O_RDWR Open for reading and writing.
- O_APPEND Append to the end of file on each write
- O_CREAT Create the file if it doesn't exist.
- O_EXCL Generate an error if O_CREAT is also specified and the file already exists.
- O_TRUNC If the file exists and if it is successfully opened for either write-only or read–write, truncate its length to 0

3. The optional last argument is shown as ..., because the number and types of the remaining arguments may vary. The last argument is used only when a new file is being created. If you look at our man pages, it states "mode specifies the permissions to use in case a new file is created. This argument must be supplied when O_CREAT is specified in flags; if O_CREAT is not specified, then mode is ignored."

Some common values of mode are

`S_IRWXU -` This is equivalent to '(S_IRUSR | S_IWUSR | S_IXUSR)'.

`S_IRGRP -` Read permission bit for the group owner of the file. Usually 040.

`S_IWGRP -` Write permission bit for the group owner of the file. Usually 020.

`S_IXGRP -` Execute or search permission bit for the group owner of the file. Usually 010.

`S_IRWXG -` This is equivalent to '(S_IRGRP | S_IWGRP | S_IXGRP)'.

On RETURN: On success, open() returns a file descriptor that is used to refer to the file in subsequent system calls. If an error occurs, open() returns –1 and errno is set accordingly.

Some Errors from open() function

If an error occurs while trying to open the file, open() returns –1, and errno identifies the cause of the error. The following are some possible errors that can occur (notable being EACCES):

EACCES The file permissions don't allow the calling process to

	open the file in the mode specified by flags. Alternatively, because of directory permissions, the file could not be accessed, or the file did not exist and could not be created.
EISDIR	The specified file is a directory, and the caller attempted to open it for writing. This isn't allowed. (On the other hand, there are occasions when it can be useful to open a directory for reading)
EMFILE	The process resource limit on the number of open file descriptors has been reached
ENFILE	The system-wide limit on the number of open files has been reached
ENOENT	The specified file doesn't exist, and O_CREAT was not specified, or O_CREAT was specified, and one of the directories in pathname doesn't exist or is a symbolic link pointing to a nonexistent pathname (a dangling link).

lseek Function

For each open file, the kernel records a file offset, sometimes also called the read/write offset or pointer. This is the location in the file at which the next read() or write() will commence. The file offset is expressed as an ordinal byte position relative to the start of the file. The first byte of the file is at offset 0. The file offset is set to point to the start of the file when the file is opened and is automatically adjusted by each subsequent call to read() or write() so that it points to the next byte of the file after the byte(s) just read or written. Thus, successive read() and write() calls progress sequentially through a file. The lseek() system call adjusts the file offset of the open file referred to by the file descriptor fd, according to the values specified in offset and whence. By default, this offset is initialized to 0 when a file is opened, unless the O_APPEND option is specified.

An open file's offset can be set explicitly by calling lseek.

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

Returns new file offset if successful, or –1 on error

The interpretation of the *offset* depends on the value of the *whence* argument.

- If *whence* is SEEK_SET, the file's offset is set to *offset* bytes from the beginning of the file.
- If *whence* is SEEK_CUR, the file's offset is set to its current value plus the *offset*. The *offset* can be positive or negative. In other words, relative to the current position.
- If *whence* is SEEK_END, the file's offset is set to the size of the file plus the *offset*. The *offset* can be positive or negative.

Because a successful call to lseek returns the new file offset, we can seek zero bytes from the current position to determine the current offset:

off_t currentPosition = currpos = lseek (fd, 0, SEEK_CUR);

Here are some other examples of lseek() calls, along with comments indicating

where the file offset is moved to:

lseek (fd, 0, SEEK_SET) ; /* Start of file */

lseek (fd, 0, SEEK_END) ; /* Next byte after the end of the file */

lseek (fd, -1, SEEK_END) ; /* Last byte of file */

lseek (fd, -10, SEEK_CUR) ; /* Ten bytes prior to current location */

lseek (fd, 1000, SEEK_END) ; /* 1001 bytes past last byte of file causing holes */

Write Function

Data is written to an open file with the write function.

ssize_t write (int *fd*, const void **buffer*, size_t *nbytes*) ;

Returns: number of bytes written if OK, 1 on error

The arguments to write() are similar to those for read(): buffer is the address of the data to be written; nbytes is the number of bytes to write from buffer; and fd is a file descriptor referring to the file to which data is to be written.

For a regular file, the write operation starts at the file's current offset. If the O_APPEND option was specified when the file was opened, the file's offset is set to the current end of file before each write operation. After a successful write, the file's offset is incremented by the number of bytes actually written.

```
char address[32] = "1600 Washington Blvd" ;

   int fw = open ( "write_out", O_WRONLY | O_CREAT, S_IRWXU );
   if( fw < 0 )
   {
      perror ( "ERROR in OPENING FILE ");
      exit (1) ;
   }

   lseek ( fw, 0, SEEK_SET); // set the pointer at the start of the file
   write ( fw, address, strlen ( address) );

   close (fw);
```

read Function

The read() system call reads data from the open file referred to by the descriptor fd.

ssize_t read (int *fd*, void **buffer*, size_t *nbytes*) ;

Returns number of bytes read, 0 on EOF, or –1 on error

The *nbytes* argument specifies the maximum number of bytes to read. (The size_t data type is an unsigned integer type.) The buffer argument supplies the address of the memory buffer into which the input data is to be placed. This buffer must be at least count bytes long.

Note: These IO calls don't allocate memory for buffers that are used to return information to the caller. Instead, we must have buffer already allocated and must pass a pointer to this allocated memory buffer and correct size.

A successful call to read() returns the number of bytes actually read, or 0 if end-offile is encountered. On error, the usual –1 is returned. The ssize_t data type is a signed integer type used to hold a byte count or a –1 error indication.

There are several cases in which the number of bytes actually read is less than the amount requested:

- When reading from a regular file, if the end of file is reached before the requested number of bytes has been read. For example, if 30 bytes remain until the end of file and we try to read 100 bytes, read returns 30. The next time we call read, it will return 0 (end of file).
- When reading from a terminal device. Normally, up to one line is read at a time.
- When reading from a network. Buffering within the network may cause less than the requested amount to be returned.
- When reading from a pipe or FIFO. If the pipe contains fewer bytes than requested, read will return only what is available.
- When interrupted by a signal and a partial amount of data has already been read.

The read operation starts at the file's current offset. Before a successful return, the offset is incremented by the number of bytes actually read.

Closing a File: close()

The close() system call closes an open file descriptor, freeing it for subsequent reuse by the process. When a process terminates, all of its open file descriptors are automatically closed.

int **close**(int fd) ;

Returns 0 on success, or –1 on error

Closing a file also releases any record locks that the process may have on the file. When a process terminates, all of its open files are closed automatically by the kernel. Many programs take advantage of this fact and don't explicitly close open files. However, It is usually good practice to close unneeded file descriptors explicitly, since this makes our code more readable and reliable in the face of

subsequent modifications. Furthermore, file descriptors are a consumable resource, so failure to close a file descriptor could result in a process running out of descriptors. This is a particularly important issue when writing long-lived programs that deal with multiple

files, such as shells or network servers. Just like every other system call, a call to close() should be bracketed with errorchecking code, such as the following:

```
if ( close ( fd ) == -1 )
   exit (  ) ;
```

buffered and unbuffered

What is buffered vs unbuffered ?

The output from functions like printf , fprintf will not be instant. The actual output will be buffered in the kernel. When the buffer is full or when you have \n character, the system would write it out to the output using the write function.

But when you the function write , the actual output will not be buffered in the kernel, the system call will be called upon to output to the stdout.

Messages that you would write to the stderr will always be write and there is no buffer involved. In our case, stderr is piped to the stdout. So, any stderr messages such as "segmentation fault (core dumped)" exception messages will be instant and there won't be any delay in outputting the data.

Now how do we prove that printf uses buffer and write doesn't ?

So we will write a program

Buffered Program :

```
main  ( )
{
 printf ( "Hello World" )  ;  // buffered output, there is no \n
 int *ptr ;
*ptr = 20;  // yields segmentation fault
}
```

In the above program, we are trying to print the string "Hello World" first followed by *ptr = 20 . Because printf didn' have \n character, it buffers data. Before it could write it out to stdout, it executes the *ptr = 20 statement. But this *ptr = 20 is illegal, it crashes the program. When it crashes, the exception message "segementation fault (core dumped) " takes precedence because it is exception and it gets piped to stderr which in turn piped to stdout. You would see this message , not the Hello World message which is still sitting in kernel somewhere.

Unbuffered Output :

In this program, we use write function instead of printf function. See the following program:

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
int main (  )
{
   write (STDIN_FILENO, "Hello World", strlen ("hello world") ) ; // unbuffered
   int *ptr ;
  *ptr = 10;  // segmentation fault
}
```

If you run this program, you will see Hello World will be written first followed by segmentation fault core dumped message. That shows write is unbuffered.

Now which is efficient ?

In case you are reading one char by char and using unbuffered functions, it will be inefficient because you are interrupting the CPU and unnecessarily clogging the IO. Unbuffered function are efficient when you want to write or read vast amount of data into your buffer. But then, it is beyond the scope of our course to run experiments to prove one way or other.

Return values of system function: read

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

On success, the number of bytes written into buf is returned.

On error, the call returns -1, and errno is set.

The file position is advanced by the number of bytes read from fd.

read function may return a positive non-zero value less than len . When only less than len bytes are available , system call may have been interrupted by a signal, pipe may have broken.

When read returns with value 0 to indicate end of file, no bytes are read. EOF is not error. It simply means there are no bytes to read past the EOF. When you are read data from a socket, then read trying to read len bytes , but there are no data to read, it then get blocked until there is a data.

In the case of EOF, there is no data at all.

In the case of blocking, there is a wait for the data to become available.

When you get -1, you evaluate the error and probably re-read from the descriptor

The call to read can return with many possibilities:

- if it is -1 and errno is set to EINTR, indicates signal interrupted from reading.
- it it is zero, EOF reached.
- if the call returns value less than len, but greater than 0, there are bytes read into the buffer.
- if the call returned a value equal to len, all requested bytes have been read.

When you want to read from a socket and don't want to get blocked when there are no data is available, you issue non-blocking call via the open function. Then read will return with a value -1 and error is to EAGAIN.

Linux File Systems and Commands

This documents describes our Linux system and the commands to use to retrieve those information. All highlighted text are commands you could give to get the same information

Linux System Information:

The command uname would print the system information

cprog> uname -a

You can type various options to get these information

uname –r (for release)

uname –s (for kernel release)

uname –p (for processor)

uname –a (Print certain system information. With no OPTION, same as -s)

You can also the same information by running the command

cat /proc/version

Hardware Information:

There are various commands for this,

lscpu - lscpu gathers CPU architecture information like number of CPUs, threads, cores, sockets, NUMA nodes, information about CPU caches, CPU family, model, byte order and prints it in a human-readable format.

lshw – lshw is a small tool to extract detailed information on the hardware configuration of the machine.

lspci - lspci is a utility for displaying information about PCI buses in the system and devices connected to them.

lsusb - lsusb is a utility for displaying information about USB buses in the system and the devices connected to them.

In the lshw and lscpu output, you will find could of interesting information. Namely, the number of disks, CPU, IDE (# of disks), memory

The actual path of these commands can be determined by typing

which lscpu

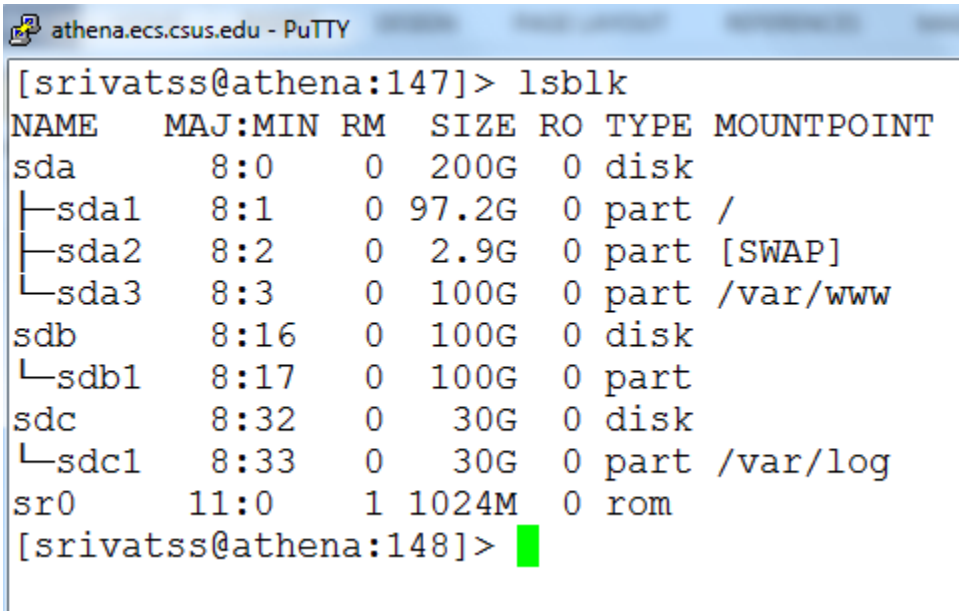
would output /usr/bin/lscpu

In our system, we find there is only one disk (it might change in future).

we can get more information about the disk and partitions and the mounts.

To get the partitions in our system, we type

lsblk



and you get the above information. The device drivers usr the major and minor numbers uniquely identify the partition, the major and minor numbers each are 16bit numbers. We will ignore this for now. We look at the mounts points which is what we should be looking. These partitions are mapped to a path using which we access the partitions.

Files are stored in a partition, partitions are formatted, and the formatted devices are mounted to a folder name. User have access to mount points.

To see the various filesystems, we have a linux command – mount. I have piped the output of mount to /dev .

c-prog>mount | grep ^/dev

```
[srivatss@athena:148]> mount | grep ^/dev
/dev/sda1 on / type ext4 (rw)
/dev/sdc1 on /var/log type ext4 (rw)
/dev/sda3 on /var/www type ext4 (rw)
[srivatss@athena:149]>
```

The process of redirecting the output of one command to another command is called piping and is represented as | (vertical bar)

Note: ext2, ext3, ext4 are types of filesystems and the format is very different. Generally the boot is formatted in ext2 format.

You can also open the file /etc/fstab and this files shows all the mount points.

Disk usage:

df –hT

df displays the amount of disk space available on the file system, h stands for human readable format, T stands for Type. Currently, the output is seen as

```
[srivatss@athena:152]> df -hT
Filesystem           Type   Size  Used Avail Use% Mounted on
/dev/sda1            ext4    96G   86G  5.1G  95% /
tmpfs                tmpfs  2.0G  2.0M  2.0G   1% /dev/shm
/dev/sdc1            ext4    30G  611M   29G   3% /var/log
/dev/sda3            ext4    99G   35G   59G  38% /var/www
gaia.ecs.csus.edu:/class
                     nfs    1.5T  1.2T  207G  86% /gaia/class
gaia.ecs.csus.edu:/home
                     nfs    1.5T  1.2T  207G  86% /gaia/home
titan.ecs.csus.edu:/software
                     nfs     96G   49G   42G  54% /titan/software
[srivatss@athena:153]>
```

You can see the file system on the left, the size, used and availability and mount point.

What is ROOT: Root is the top most directory under which all other directories reside, note the actual file system they point may be on different file system. You can navigate to the root directory type

cd /

and navigate to sub-directories as shown in the screenshot.

Various file folders under Linux starting from root.

Command	Comment
cd /	will take you to the root folder. do ls to see all folder. bin is one folder
cd /bin	here all basic shell commands exist. Commands such as ls, mkdir, pwd,
cd /sbin	here all system related commands exist. We will skip here
cd /home	All user accounts are stored. Staff accounts are stored in /home/staff/ and student accounts are stored in /home/student/
cd /lib	all library files are stored here
cd /var	A folder to store temporary files (usually log files which generally removed / purged after sometime)
cd /etc	Here important configurations system files are stored. Files such as user account and password (/etc/passwd) , file partitions (etc/fstab)

Where is User Accounts and Password file:

The user ID, name, password, groupID and the default home directory and shell are stored in a file : /etc/passwd

If you open the file, each line looks like this,

ssrivatsa:x:2464:2470:Sankar Srivatsa:/home/student/ssrivatsa:/bin/bash

where

ssrivatsa	user loginID
x	indicates presence of a password
2464	user ID
2470	group ID, accounts may belong to a group
full name	seen in the finger command,
/home/student/ssrivatsa	path to the home directory , one can change the default home directory to another directory using usermod command
/bin/bash	default shell program to launch during login

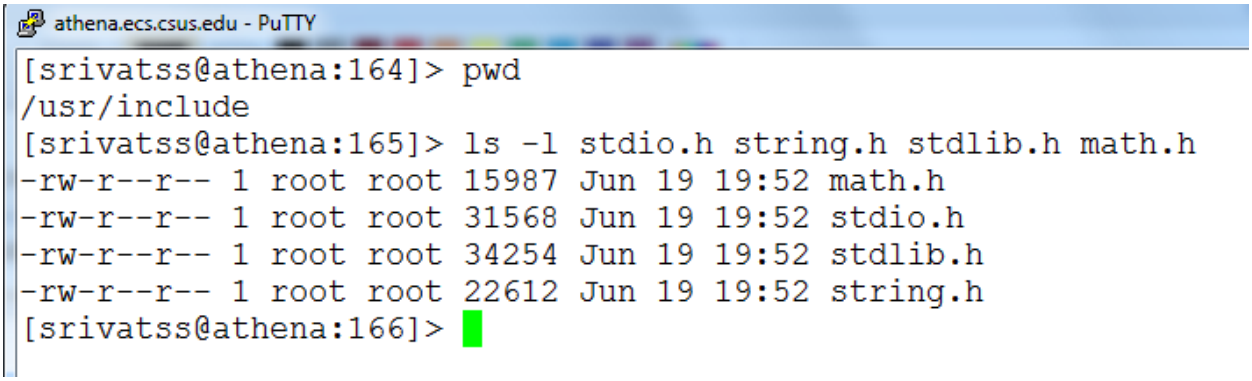
You can get all these information using the command - finger loginid (ex finger ssrivatsa)

You can get the user id and group id of yours using getuid ( ) and getgid ( ) functions

```
c-prog>cat user_id.c
#include <stdio.h>

int main ( )
{
  printf ( "User ID = %d group ID=%d \n", getuid(), getgid ( ) );

 return 0;
}
c-prog>
c-prog>
c-prog>gcc user_id.c
c-prog>
c-prog>
c-prog>./a.out
User ID = 2319 group ID=2319
c-prog>
```

Where are C library files stored ?

Some of the header files you may use in your programs are stored in /usr/include folder. Navigate to /usr/include



Types of Files and Directories:

There are two main types of files : regular files and directories. Technically, directories are stored as files , but we don't see them as files, rather folders. The various types are

Regular File	The most type of file. There is no distinction between binary or text file. The distinction to be made is left to the application
Directory File	A file that contains the names of other files and pointer to the information on these files. Any process that has read permissions can read the contents of the directory
Block special files	a type of file providing buffered IO to devices such as disk drives
Character file	a type of file providing unbuffered IO.
Socket file	Used for programming network communications (web…)
symbolic files	A type of file that points to another file..
FIFO	used to communicate between processes

Virtual Memory and Physical Memory

The virtual memory is stored as 4K pages , in a linked list data structure - vma_struct.

struct vm_area_struct {

struct mm_struct * vm_mm; /* The address space we belong to. */
unsigned long vm_start; /* Our start address within vm_mm. */
unsigned long vm_end; /* The first byte after our end address
within vm_mm. */

/* linked list of VM areas per task, sorted by address */
struct vm_area_struct *vm_next;

pgprot_t vm_page_prot; /* Access permissions of this VMA. */
unsigned long vm_flags; /* Flags, see mm.h. */

....
}

Each page is mapped to a unique base address in the page table. Each process has its own page table which contains the base addresses for each Virtual page. The base address is added to the virtual page to yield the actual physical address. The physical memory is accessed via the data structure mm_struct.

```
struct mm_struct (http://lxr.linux.no/linux+v2.6.28.1/+code=mm_struct) {
    struct vm_area_struct (http://lxr.linux.no/linux+v2.6.28.1/+code=vm_area_struct) * mmap (http://lxr.linux.no/linux+v2.6.28.1/+code=mmap) ;            /* list of VMAs */
    struct rb_root (http://lxr.linux.no/linux+v2.6.28.1/+code=rb_root) mm_rb (http://lxr.linux.no/linux+v2.6.28.1/+code=mm_rb)
;
    struct vm_area_struct (http://lxr.linux.no/linux+v2.6.28.1/+code=vm_area_struct) * mmap_cache (http://lxr.linux.no/linux+v2.6.28.1/+code=mmap_cache) ;     /* last find_vma result */
```

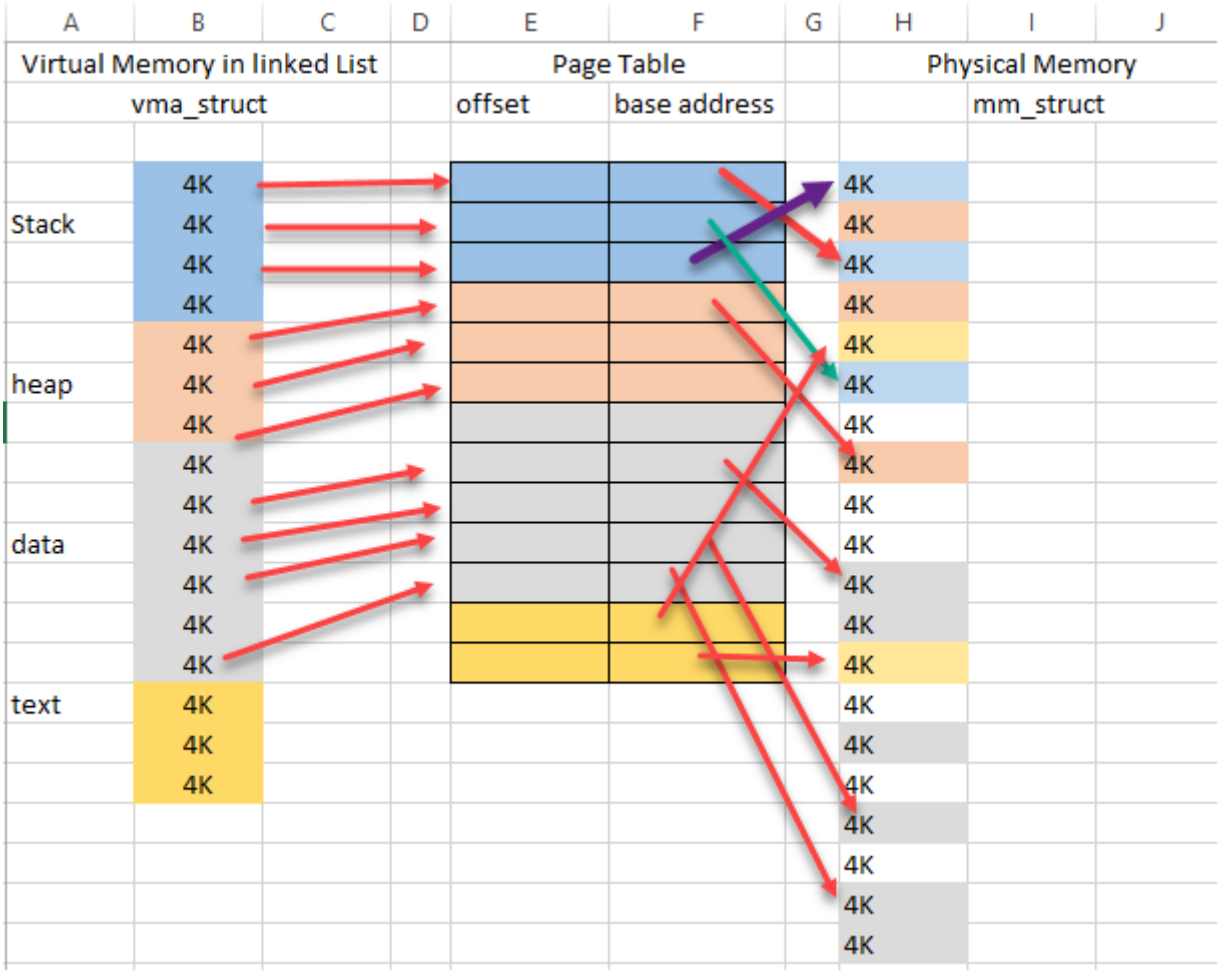
...

} ;

See how the virtual pages are accessed via a pointer mmap

If for a virtual page is mapped to a base address, but the physical page is not present in RAM, a page fault occurs. The IO manager will have to bring the page from the disk to the physical RAM. But page fault is beyond the scope of this course.

If all RAM is occupied, then page faults occurs a lot losing efficienty and slow performance.



You can view the virtual address in the directory: /proc/process-id/smaps

If your process id is = 14065

then you can do this command to see the virtual pages:

cat /proc/14065/smaps

What is a.out ?

A executable program such as a.out is a file containing a range of information that describes how to construct a process at run time. This information includes the following:

- Binary format identification : Each program file includes meta information describing the format of the executable file. Most UNIX implementations (including Linux) employ the Executable and Linking Format (ELF). This enables the kernel to interpret information in the file.
- Machine-language instructions: These encode the algorithm of the program.
- Program entry-point address: This identifies the location of the instruction at which execution of the program should commence.
- Data: The program file contains values used to initialize variables and also literal constants used by the program (e.g., strings).
- Symbol and relocation tables: These describe the locations and names of functions and variables within the program. These tables are used for a variety of purposes, including debugging and run-time symbol resolution (dynamic linking).
- Shared-library and dynamic-linking information: The program file includes fields listing the shared libraries that the program needs to use at run time and the pathname of the dynamic linker that should be used to load these libraries.
- Other information: The program file contains various other information that describes how to construct a process

What is a Process:

A process is an instance of an executing program. Several processes can be running the same program. While running the program, kernel appends additional information required to run the program. Information such as kernel data structures that maintain information about the state of the process. The information recorded in the kernel data structures include various identifier numbers (IDs) associated with the process, virtual memory tables, the table of open file descriptors, information relating to signal delivery and handling (we will talk about Signals soon), process resource usages and limits, the current working directory, and a host of other information. Each running program will have a ID and usually launched by another programs aka parent process. In our system and in all systems, the init process (aka Kernel) has ID = 1 and all processes will be child of the his process.

Process ID and parent process ID

Each process has a unique integer process identifier (PID). Each process also has a parent process identifier (PPID) attribute, which identifies the process that requested the kernel to create this process.

#include <unistd.h>

pid_t getpid(void);	Returns: process ID of calling process
pid_t getppid(void);	Returns: parent process ID of calling process

uid_t getuid(void);	Returns: real user ID of calling process
uid_t geteuid(void);	Returns: effective user ID of calling process
gid_t getgid(void);	Returns: real group ID of calling process
gid_t getegid(void);	Returns: effective group ID of calling process

With the exception of a few system processes such as init (whose process ID is always 1), process ID will very different next time you run the program. The Linux kernel limits process IDs to being less than or equal to 32,767. When a new process is created, it is assigned the next sequentially available process ID. Each time the limit of 32,767 is reached, the kernel resets its process ID counter so that process IDs are assigned starting from low integer values.

Unix Commands

Processes

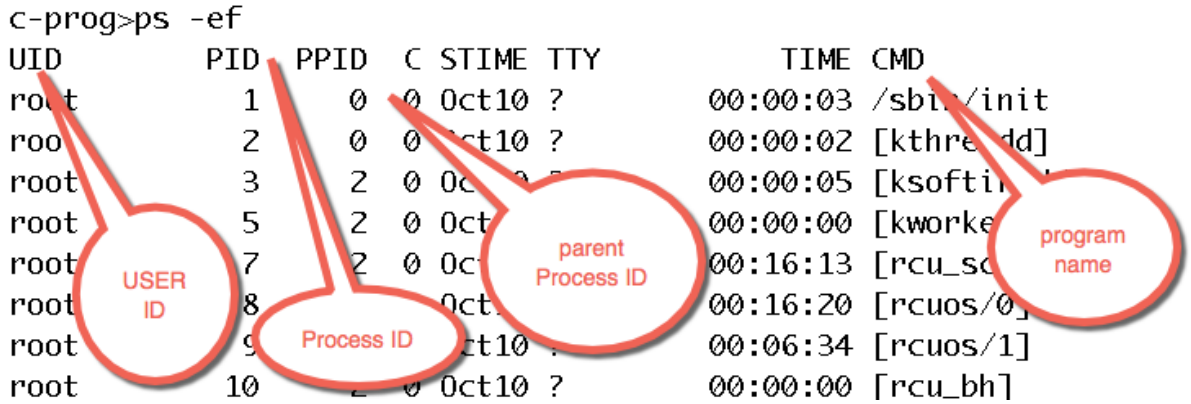
A program is a set of instructions in passive state stored in a file. A process executes this program in active state and is executed by a processor. Every process in Linux gets a process ID.

The process ID of a process can be printed using the system function getpid ( ) and the parent process ID as getppid ( ) . You can also print the process ID with the command

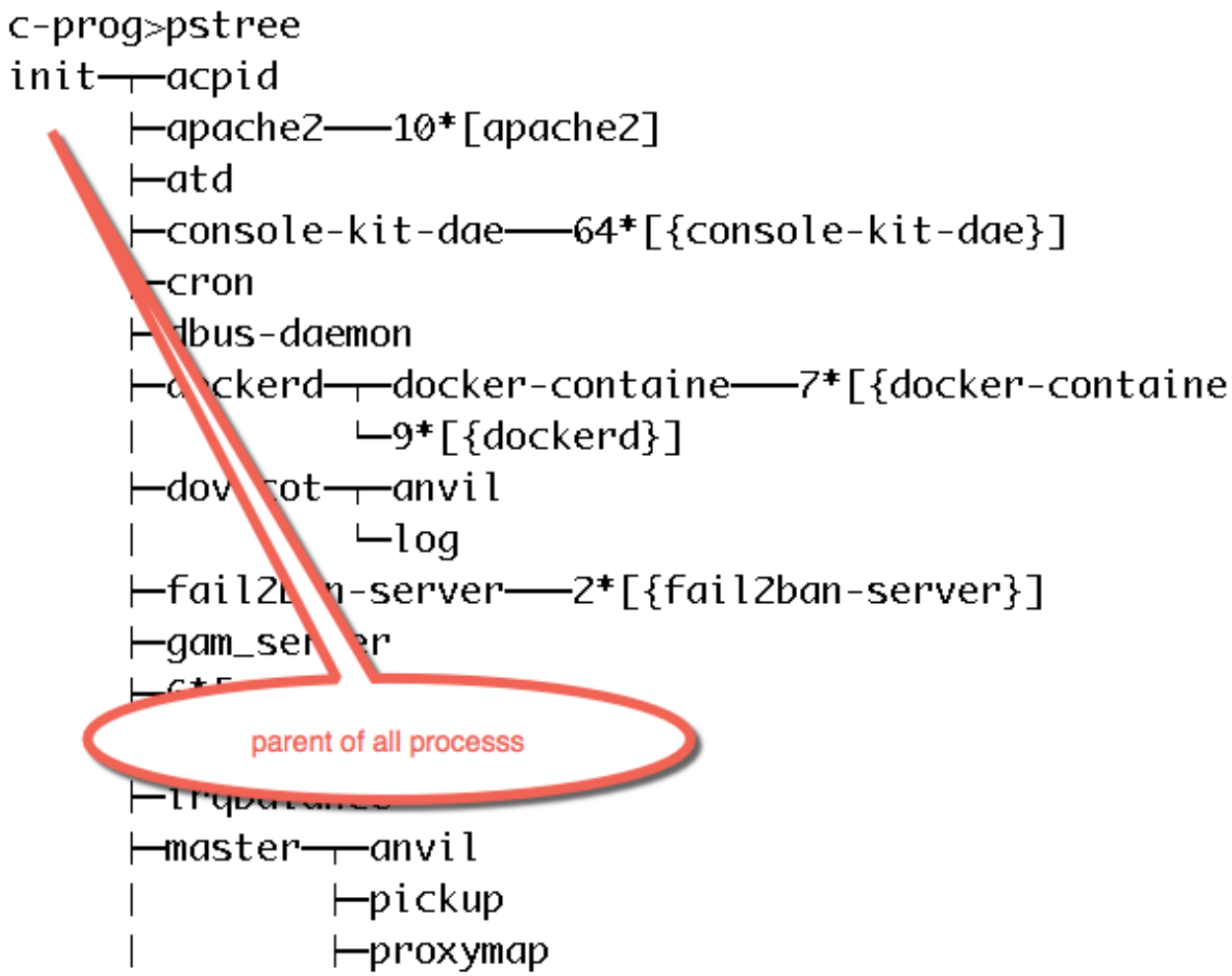
ps -ef

Every process is a child of another process , except the init process.

see the screen shot



You can view the processes in a tree structure using the command pstree command.



You can view the processes created by you, by typing

c-prog>ps -u <USERID>

PID TTY TIME CMD

15113 ? 00:00:00 sshd

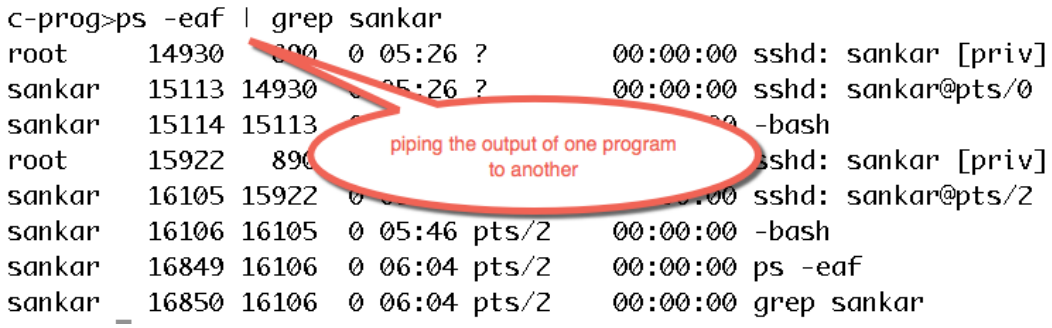
15114 pts/0 00:00:00 bash

16105 ? 00:00:00 sshd

16106 pts/2 00:00:00 bash

16913 pts/2 00:00:00 ps

or you could view in a long format using the pipe command to ps -eaf



When a process is executed by the CPU, the process accesses all registers and RAM. Because there are several programs share CPU time, Operating System uses a scheduling algorithm (a popular being Round robin) which gives equal time slice to all processes and processes may have different priority to run. Some processes may have higher priority than others. When a process is given the CPU time, it might finish execution during the time slice it is allocated or it may be swapped out with another process that is next in the pipeline. This is called preempted.

Try this command

top

The program that is running is flagged as R and the programs in waiting/sleeping flagged as S. You quit the top program by typing q .

Memory Layout of a C Program

A program may contain global variables (initialized and uninitialized) and functions. When you launch your program, it is loaded into memory . While running, our program can be swapped/switched with another, and we don't want to lose the status of our program. To store the status of our program, our program is compartmentalized into various sections. We can get the size of each of these sections using the command

size a.out <enter>

Let us examine the sections. The various sections are given below

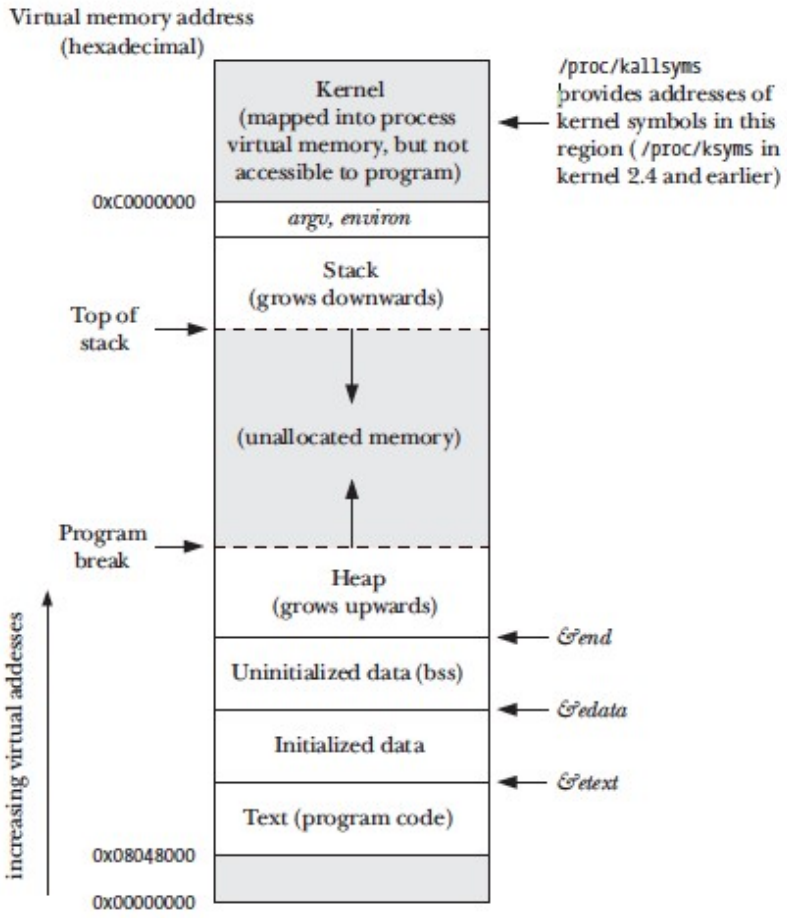


Figure 6-1: Typical memory layout of a process on Linux/x86-32

Text Area: The text segment contains the machine-language instructions of the program run by the process. The text segment is made read-only so that a process doesn't accidentally modify its own instructions via a bad pointer value. Since many processes may be running the same program, the text segment is made sharable so that a single copy of the program code can be mapped into the virtual address space of all of the processes.

Stack Area: This section is meant for functions. When your program calls functions, the instructions, local variables, return address and other information is loaded in this area.

Heap Area: If your programs allocates dynamic memory, this memory is allocated in the heap area. As your program allocates more memory, all this is allocated in the heap area. Managing this area is very complicated.

Stack vs Heap: Management of this stack area is little easier than Heap area because sometime your program will deallocate the dynamic memory in heap causing gaps resulting in memory gaps. Sometimes, your program may not deallocate the dynamic memory resulting in memory leaks. The stack area grows linearly from top to bottom and management of this memory is little easier.

BSS Area:

This uninitialized data segment contains global and static variables that are not explicitly initialized. Before starting the program, the system initializes all memory in this segment to 0. For historical reasons, this is often called the bss segment, a name derived from an old assembler mnemonic for "block started by symbol." The main reason for placing global and static variables that are initialized into a separate segment from those that are uninitialized is that, when a program is stored on disk, it is not necessary to allocate space for the uninitialized data. Instead, the executable merely needs to record the location and size required for the uninitialized data segment, and this space is allocated by the program loader at run time

Data section : This initialized data segment contains global and static variables that are explicitly initialized. The values of these variables are read from the executable file when the program is loaded into memory.

We can more information about these segments using command

```
objdump –-f -h a.out
```

We will write several versions of a simple program adding variables in each version and check how the sections of the code varies in size. Here is the table , the left column is our program, the center column summarizes the changes we made, the right column shows the size of Text, Data and BSS sections. The command to get the size of these sections is size a.out

Summary of our work		
Program : mem.c	Type of variables in the program	size a.out<enter>

#include <stdio.h>
int main ( )
{
}
	no variables.	text data bss
		1076 496 16

#include <stdio.h>
int age = 20;
int main ( )
{
}
	one global variable initialized.	text data bss
		1076 500 16

#include <stdio.h>
int age = 20;
int myAge ;
main ( )
{
}
	one global variable initialize
	one global variable uninitialized
		text data bss
		1076 500 24

#include <stdio.h>
int age = 20;
int myAge;
main ( )
{
int mainAge = 30;
}
	one global variable initialize
	one global variable uninitialized
	one local variable
		text data bss
		1092 500 24

Global Variables : etext, edata and end

Most UNIX implementations (including Linux) provides three global symbols: etext , edata , and end . These symbols can be used from within a program to obtain the addresses of the next byte past, respectively, the end of the program text, the end of the initialized data segment, and the end of the uninitialized data segment. To make use of these symbols, we must explicitly declare them, as follows:

.

#include <stdio.h>
extern etext, edata, end ;
main ( )
{
printf ("End of Text segment %10p \n", &etext);
printf ("End of Data segment or start of BSS %10p \n", &edata);
printf ("End of BSS data %10p \n", &end);
 	
}
	/* &etext gives the address of the end of the program text / start of initialized data */

You can get information about these variables using man command

man end

```
NAME
       etext, edata, end - end of program segments

SYNOPSIS
       extern etext;
       extern edata;
       extern end;

DESCRIPTION
       The addresses of these symbols indicate the end of various program seg
       ments:

       etext  This is the first address past the end of the text segment  (the
              program code).

       edata  This  is  the first address past the end of the initialized data
              segment.

       end    This is the first address past the end of the uninitialized data
              segment (also known as the BSS segment).
```

when you run this program

a.out<enter>

the output is

End of Text segment 0x400626

End of Data Segment or start of BSS 0x601020

End of BSS data 0x601030

Virtual Memory Management:

All the addresses we print in our programs are virtual address, ie they are not real physical addresses. Linux employs a technique known as virtual memory management. The aim of this technique is to make efficient use of both the CPU and RAM (physical memory). A virtual memory scheme splits the memory used by each program into small, fixed-size units called pages . Correspondingly, RAM is divided into a series of page frames of the same size. At any one time, only some of the pages of a program need to be resident in physical memory page frames; these pages form the so-called resident set . Copies of the unused pages of a program are maintained in the swap area —a reserved area of disk space used to supplement the computer's RAM—and loaded into physical memory only as required. When a process references a page that is not currently resident in physical memory, a page fault occurs, at which point the kernel suspends execution of the process while the page is loaded from disk into memory. On x86-32, pages are 4096 bytes in size. We can get the page size in our system using the program here

```
#include <stdio.h>
#include <unistd.h>
int main ( )
{
long sz = sysconf(_SC_PAGESIZE);
 printf ( "page size = %ld \n", sz );

}
c-prog>gcc pagesize.c
c-prog>./a.out
page size = 4096
```

In our system, the page size is 4K.

How are virtual addresses mapped to physical addresses then ?
The kernel maintains a page table for each process (see the Figure below). The page table describes the location of each page in the process's virtual address space (the set of all virtual memory pages available to the process). Each entry in the page table either indicates the location of a virtual page in RAM or indicates that it currently resides on disk.

Virtual memory management separates the virtual address space of a process from the physical address space of RAM. This provides many advantages :

- Processes are isolated from one another and from the kernel, so that one process can't read or modify the memory of another process or the kernel. This is accomplished by having the page-table entries for each process point to distinct sets of physical pages in RAM (or in the swap area).
- Where appropriate, two or more processes can share memory. The kernel makes this possible by having page-table entries in different processes refer to the same pages of RAM. This could happen when we fork a process, the child and parent may share the same pages until one of them updates the memory, then the copy of the memory is created , this is copy-on-write concept.
- The implementation of memory protection schemes is facilitated; that is, pagetable entries can be marked to indicate that the contents of the corresponding page are readable, writable, executable, or some combination of these protections.
- Where multiple processes share pages of RAM, it is possible to specify that each process has different protections on the memory; for example, one process might have read-only access to a page, while another has read-write access.
- Programmers, and tools such as the compiler and linker, don't need to be concerned with the physical layout of the program in RAM.
- Because only a part of a program needs to reside in memory, the program loads and runs faster.
- One final advantage of virtual memory management is that since each process uses less RAM, more processes can simultaneously be held in RAM. This typically leads to better CPU utilization, since it increases the likelihood that, at any moment intime, there is at least one process that the CPU can execute.

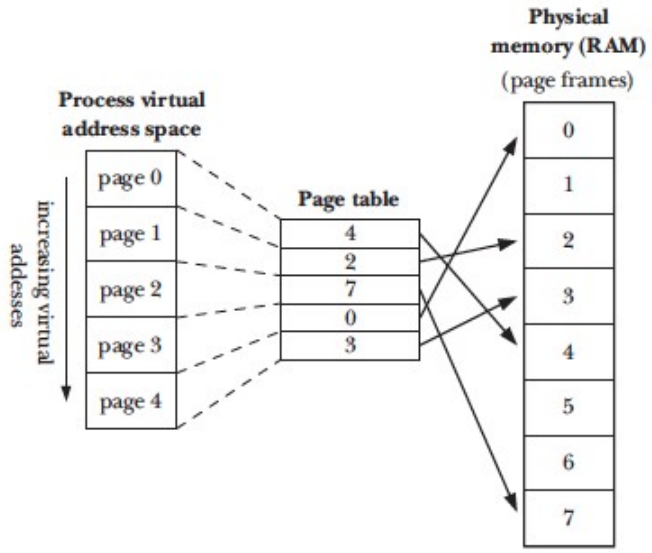


Figure 6-2: Overview of virtual memory

SYSTEM CALL

User vs Kernel Space :

Before we talk about system call operations, we need to understand that most CPUs typically operate in at least two different modes: user mode and kernel mode (sometimes also referred to as supervisor mode). When running in user mode, the CPU can access only memory that is marked as being in user space; attempts to access memory in kernel space result in an exception. For example, a code

```
int x= 0;

scanf ( "%d", x) ;
```

will result in exception because the user program is trying to assign a value to address 0 which may be in kernel space. The actual code should be of course

```
scanf ( "%d", &x) ;
```

This ensures that user processes are not able to access the instructions and data structures of the kernel, or to perform operations that would adversely affect the operation of the system. When running in kernel mode, the CPU can access both user and kernel memory space.

System Calls A system call is a controlled entry point into the kernel, allowing a process to request the kernel perform some action on the process's behalf. The kernel makes a range of services accessible to programs via the system call application programming interface (API). These services include, for example, creating a new process, performing I/O, and creating a pipe for interprocess communication. (The syscalls(2) manual page lists the Linux system calls and in our linux system the /usr/include/asm/unistd_32.h) Before going into the details of how a system call works, some notable points are :

- A system call changes the processor state from user mode to kernel mode, so that the CPU can access protected kernel memory.
- The set of system calls is fixed. Each system call is identified by a unique number. (This numbering scheme is not normally visible to programs, which identify system calls by name.) Each number is nothing but an array index to a data structure called System Call Table. Luckily, we don't have to worry about the numbers, we only have to know the system call name and parameters.
- Before kernel is starting to execute the system call, the user program may have a set of arguments that specify information to be transferred from user space to kernel space and vice versa.

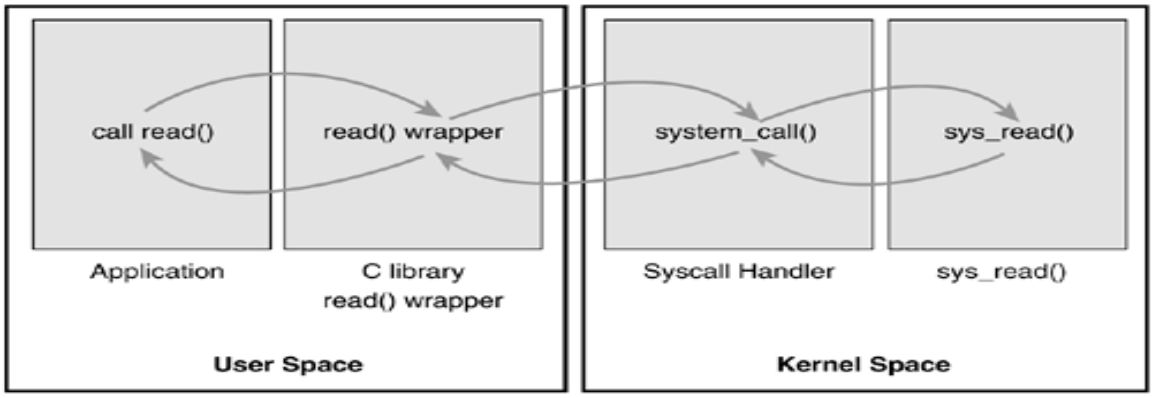
From a programming point of view, invoking a system call looks much like calling a C function. However, behind the scenes, many steps occur during the execution of a system call.

syscall ( ) function - How does the system execute library calls in kernel space ?

From a programming point of view, invoking a system call looks much like calling a C function. We know a function has a name and may have a bunch of input parameters. In the system, each system function name is mapped to a unique number. We will see how this works :

IN USER SPACE:

1. Because we cannot call the system function directly, a wrapper function is given to us. The application program makes a system call by invoking this wrapper function indirectly in the C library.



2. Because Kernel expects all arguments of the function in specific registers in the CPU, the wrapper function copies the arguments to these registers. These registers are mentioned below

Architecture	arg1	arg2	arg3	arg4	arg5	arg6
i386	ebx	ecx	edx	esi	edi	ebp
x86_64	rdi	rsi	rdx	r10	r8	r9
x32	rdi	rsi	rdx	r10	r8	r9

3. The wrapper function then copies the unique number onto a specific CPU register eax

4. Lastly, the wrapper function executes a trap machine instruction 0x80 which causes the processor to switch from user mode to kernel mode. New architectures use syscall. The program now in kernel space.

IN KERNEL SPACE :

5. Now in kernel space, in response to the trap to location 0x80, the kernel invokes its system_call() routine to execute the function on behalf of the user program.

This routine :

a) Saves register values onto the kernel stack. That moves data to its internal memory.

b) VALIDATES THE FUNCTION CALL: Checks the validity of the system call number.

c) VALIDATES THE ARGUMENTS: If the system call service routine has any arguments, it first checks their validity.

d) Invokes the system call service routine mapped to the unique number.

e) Once the routine finishes, the service routine returns

d) Restores register values from the kernel stack and places the system call return value on the stack.

e) Returns to the wrapper function, simultaneously returning the processor to user mode.

6. If the return value of the system call service routine indicated an error, the wrapper function sets the global variable errno using this value. The wrapper function then returns to the caller, providing an integer return value indicating the success or failure of the system call.

Here is the simple C program :

```
int main ( )

{

 printf ( " Hello World \n"  ) ;  gets converted into Syscall by the compiler

 return 0 ;

}
```

The assembly code for the above C code is something like this:

```
.global _start
   .text
_start:
   # write(1, message, 13)
       mov     $4, %eax              # system call 4 is write
       mov     $1, %ebx              # file handle 1 is stdout
       mov     $message, %ecx        # address of string to output
       mov     $13, %edx             # number of bytes to write
       int     $0x80                 # invoke operating system code

       # exit(0)
       mov     $1, %eax              # system call 1 is exit
       xor     %ebx, %ebx            # we want return code 0
       int     $0x80                 # invoke operating system code
message:
       .ascii  "Hello, World\n"
```

ERROR HANDLING

When a system function encounters an error, it notifies the caller about the error using a negative value -1. There is no additional information about the error. But additional information can be viewed through the global integer variable errno, and is usually set by the system call to a constant value that gives additional information. Note: We don't set this value, it is set by the system.

On Linux, the error constants are listed in the errno manual page. You can also view the values in the header file <errno.h>. Valid error numbers are all nonzero; errno is never set to zero by any system call or library function.

errno is defined by the ISO C standard to be a modifiable lvalue of type int, and must not be explicitly declared like this;

extern int errno;

The value returned is valid only immediately after an errno-setting function indicates an error (usually by returning -1). Developers should quickly process this error and make suitable action. Because the variable can be modified during the successful execution of another or this function.

The errno variable may be read or written directly; it is a modifiable lvalue. The value of errno maps to the textual description of a specific error. A preprocessor #define also maps to the numeric errno value. For example, the preprocessor define EACCESS equals 1

Some examples of the description mapped to errno are

EACCES Permission denied (POSIX.1)

EISDIR Is a directory (POSIX.1)

ENAMETOOLONG Filename too long (POSIX.1)

The C library provides a handful of functions for translating an errno value to the corresponding textual representation. This is needed only for error reporting, and the like; checking and handling errors can be done using the preprocessor defines and errno directly.

The two functions we need to be aware are

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

1. char *strerror(int errnum);

   Returns: pointer to message string

This function takes the errnum argument, which is typically the errno value and returns a pointer to the string.

The other function , perror function produces an error message on the standard error, based on the current value of errno, and returns.

#include <stdio.h>

2. void perror(const char *msg) ;

This outputs the string (usually passed by the application) pointed to by msg, followed by a colon and a space, followed by the error message corresponding to the value of errno, followed by a newline.

Here is the sample output

```
c-prog>cat perror.c
 #include <errno.h>
 #include <stdio.h>
 #include <string.h>
 int main ( )
 {
   // open a non-existent file
   FILE *fp = fopen ( "WhereIsMyfile.txt", "r");

   printf ( " Error number is %d \n", errno  ) ;
   printf ( " Error message is %s \n", strerror ( errno ) ) ;

   if ( fp == NULL ) {
         perror ( "Some ERROR OPENING FILE " ) ;
   }

 }

c-prog>gcc perror.c
c-prog>./a.out
 Error number is 2
 Error message is No such file or directory
Some ERROR OPENING FILE : No such file or directory
c-prog>
```

Another example is :

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

int main ( )
{

     char *mBuf;
     int memsize = -1 ;

      if ((mBuf = malloc(memsize)) == NULL)
      {
        printf ( "error NO %d \n", errno ) ;
        perror("Malloc ");
        printf ( "Malloc occurred %s \n", strerror ( errno) ) ;
        exit(1);
      }

}
[srivatss@athena:194]> gcc err.c
[srivatss@athena:195]> ./a.out
error NO 12
Malloc : Cannot allocate memory
Malloc occurred Cannot allocate memory
```

Introduction to Pointers

Introduction to Pointer

Pointers are variables whose values are *memory addresses*. Normally, a variable directly contains a specific value. A pointer, on the other hand, contains an *address* of a variable that contains a specific value. In this sense, a variable name *directly* references a value, and a pointer *indirectly* references a value

Pointers, like all variables, must be defined before they can be used. Pointer variables are defined with asterisk prefixed with the name of the variable, like shown in here

```
int *ptr ;
```

ptr is now a pointer variable. It is not yet pointing to an address , so it is either having a garbage value or NULL. A pointer with the value NULL points to *nothing.* A pointer may be initialized to NULL, 0 or an address.

```
ptr = NULL ;
```

How do we store address ? Remember, pointers store address of another variable. Then, we will define a new variable, say

```
int x = 100 ;
```

We learnt from scanf function, where we sent the address of a variable using &. The address of x is &x, right ?

Now, make ptr point to variable x by this next statement,

```
ptr =  & x ;
```

The &, or address operator, is a unary operator that returns the address of its operand. In the above statement, we are taking the address of x (using the unary operator & like we used to in the scanf function). Because ptr is a pointer variable , it is perfectly okay to store the address of a variable (in this case x).

But it is not okay to do this

ptr = x ; // why ?

because the value of x (which is just 100) will be stored in ptr. That is, ptr is pointing to an address 100. The compiler won't complain about it, but during runtime, 100 is a memory location (either in the kernel to which we don't have permission to access or 100 may be some random address) which our programs don't have access to and most of time the program crashes. Some of the crashes are mainly the result of accessing a memory location that is not in our space.

Also, note : The operand of the address operator must be a variable; the address operator *cannot* be applied to constants or expressions.

for instance,

ptr = &100 ; // NOT OKAY.

ptr = 100 ; // not OKAY

NOTE: POINTERS CAN ONLY STORE ADDRESS OF ANOTHER VARIABLE.

Because pointers can be made to point to different addresses, we show how pointers can be made to change to point to different memory locations. This is the power of pointers.

To print just address ptr is storing, we say

printf (" The address in ptr %p \n ", ptr) ;

Did you notice %p , a conversion specifier. Also, did you notice use of just pointer variable ptr.

Dereference a Pointer

Dereferencing A Pointer that appears on RHS

We know how to store the address of a variable in a pointer variable.

int *ptr , x = 10 ;

// we declared a pointer variable and a regular variable x = 100

ptr = &x ;

In pointer world, we say pointer ptr is pointing to variable x. Using ptr, we can access the value of x; How ? This is done using the asterisk operator *. It is coincidence that we declare a pointer variable using asterisk and dereference it using an asterisk operator. But they mean differently: one is to declare a pointer variable and the other to mean to dereference.

Deference a pointer variable occurring on the RHS of an equal operator means we are fetching the value the pointer is pointing to. We use the * operator.

For instance ,

say

y = 200 ;

ptr = &y ; // ptr is pointing to y

the statement

x = *ptr ;

means

1. fetch the address stored in the pointer variable ptr, in this case the address of y

2. then, get the value from that address, in this case 200.

This is called dereference.

Dereferencing A Pointer that appears on LHS

Now the supervisor says, can you deliver this letter to this house you are watching.

x = 100;

ptr = &y ; // assignment

*ptr = x ; // dereference on LHS

then the last statement means

store the value of x to where ever ptr is pointing to.

Another example using dereferene operator :

- printf(**"%d", *ptr);**

prints the value of variable y, namely 100.

Here we fetch the value as if it is occurring on the RHS.

- Using * in this manner is called dereferencing a pointer. The operation means different when it occurs on RHS and LHS

Take this example:

y = *ptr + 100 ;

fetching the value where ptr is pointing and add 100.

Other examples ,

y = 100 + *ptr ;

y = *ptr * 2 ;

y = 2**ptr ; // this is not 2 power y like in some languages

This is incorrect

&ptr = &x ;

y = &ptr + 2 ; // this is incorrect unless y is also a pointer.

Here is the video:

Dereferencing Pointers (https://youtu.be/9z-2kUWGWMg)

(https://youtu.be/9z-2kUWGWMg)

Declare and initialize in one statement

Initializing Pointers during declaration

So far we are used to declaring pointers as

int x = 100 ;

int *ptr = &x ;

Instead of declaring a pointer and then assigning an address, we can declare and initialize a pointer in one statement like this

int *ptr = &x ;

This means, we are assigning the address of x to pointer ptr while defining the pointer itself. In this kind of statement, we are completely dealing with addresses.

Alternatively, we could do this too

int x = 100, *ptr = &x ;

Other declarations could be:

int y = 100 ;

int *ptr1 = &y , *ptr2 ;

int x, *ptr3 = &x ;

Now, consider this.

int *ptr4 = ptr1 ; // Watch this out. ptr1 has a memory address , which is the address of a variable y. We are assigning this memory address to ptr4

In other words, ptr1 and ptr4 are pointing to variable y.

but

int *ptr4 = *ptr1 is mistake because *ptr1 is 100. We are assigning address 100 to ptr4 which is incorrect. Right ?

Valid and Invalid pointer assignments

Incorrectly Assigning values to pointers

int *ptr ;

int x ;

1. ptr = 100 ;

would be terribly wrong because the data 100 is stored in the pointer variable. That means, ptr is pointing to the address 100. 100 may be the memory in Kernel space resulting in system error and crashing the program.

2.

x = 1023;

ptr = x ;

again it is wrong. Why ? The value of x is a data value. You cannot store in a pointer variable. ptr would then be pointing to address 1023 which is again in kernel space, not user space.

3.

ptr = x + y ;

again terrible. Why ? you are adding data values of x and y and storing them in a pointer variable.

Here is the video of incorrect assignments of pointer.

4. x = ptr ;

is again wrong. You are taking the address stored in pointer ptr and storing in a regular int variable x. This probably a typo on the part of the programmer, he probably meant

x = *ptr ;

5.

ptr = &x + 10 ;

This is generally okay. Why ? We are getting the address of x and adding 10 to that address. The resulting value is again an address and storing that address in ptr. But avoid this kind of assignment ,

we generally do arithmetic operations with pointers only when they are pointing to arrays or dynamic memory locations, not with variables like this.

6.

*ptr = &x ;

is wrong. Why ? Because we are taking the address of x and storing to a location by dereferencing the ptr.

The programmer may have meant

ptr = &x ; or

*ptr = x ;

Some good examples on how to use pointers

Consider the following statements,

int x = 200, y = 100, *ptr1, *ptr2 ;

we have two variables of type int, two pointer variables of type int.

1.

ptr = &x ;

is perfectly okay. Why ? We are taking the address of x and storing it in the pointer variable. In other words, ptr is now pointing to x.

2.

*ptr = x ;

is good. We are placing the value of x in the location ptr is pointing. But make sure ptr is pointing to a valid location. If ptr is not pointing to any location, you will runtime error.

3.

y = *ptr ;

is good too. Again, make sure ptr is pointing to a valid memory location

4.

*ptr = x + y ;

is good. We are adding x and y, copying the sum to a location where ptr is pointing . Again, make sure ptr is pointing to.

5.

x = *ptr + 5 ; is good too.

Pointer Arithmetic with Arrays

Consider this array

char cData[4] = { 'A', 'B', 'C', 'D' } ;

Here cData is called pointer constant. cData is also the array name pointing to the first cell.

NOTE: Though cData is also an array, it is also a pointer constant.

you cannot change cData to point elsewhere. It is stuck pointing to the first cell.

Because array name is acting like a pointer, we can assign the array to a pointer too. Like shown here

char *ptr = cData ;

is a perfectly valid statement and ptr is pointing to cData.

Deferencing it

*ptr would yield 'A'.

Now, when we write

ptr = ptr + 1 ;

We are adding one to the address in ptr. Basically we are incrementing or advancing the pointer to the next address. In this case , ptr is now pointing to the next cell in the array.

dereferencing *ptr would then fetch 'B' for us.

The above statement can also be written as

ptr++ ; // is same as ptr = ptr + 1 .

ptr++ would then advance to the next cell.

*ptr would then fetch 'C'

ptr++ one more time would point to the next cell

*ptr would fetch 'D'

Consider this array

int iData [4] = { 0x1, 0x2, 0x3, 0x4 } ;

say the address of

first cell = 4400

second = 4404

third cell = 4408

fourth cell = 4412

int *iPtr = iData ; // point iPtr to the address 4400..

Here iPtr is pointing to iData

iPtr++ ;

would advance the pointer to the second cell.

iPtr++ would advance to the third cell.

So, when it comes to pointer arithmetic, say ptr + 1 is not just add one to the pointer. The resulting value is based on the type of the pointer times the number.

For instance, say ptr is pointing to an address 4400 like above.

ptr + x is

ptr + (x times sizeof (datatype of pointer)) = ptr + (x * sizeof (int))

For ptr + 1 = it is ptr + (1 * 4) = 4400 + 4 = 4404 which is the address of second cell

ptr + 2 = ptr + (2 * 4) = 4400 + 8 = 4408 is the address of the third cell

You can take this example and run it yourself

```
#include <stdio.h>
int main ( )
{
        int *iPtr ;
        int iData[4] = {1, 2, 3, 4};
        iPtr = iData;
         printf("before %p\n", iPtr);
         iPtr = iPtr + 1;
         printf("After %p\n", iPtr);
}
```

Dynamic Memory Allocation functions: malloc, free, calloc, realloc, memove

Memory can be allocated from two pools: stack and Heap. When we declared variables in a function, we allocated memory from the stack space.

when we declare say a variable

int x ;

The four bytes is allocated from the stack , predetermined during compile time.

But, programmers may not know how much memory the program requires when they are writing the program. So , during runtime, we need memory to be allocated. Also, Self referential structure require to be allocated dynamically. malloc is one system function that you would call to allocate memory. This memory is allocated from heap space.

Step 1:

The sysetm function to allocate memory during runtime is malloc

int x= 8;

malloc (x) ; // this allocates 8 bytes

malloc (x+4) ; // allocates 12 bytes

malloc (1024) ; // allocates 1024 bytes

STEP 2 :

When the malloc returns, it returns a void pointer to the memory allocated. We cannot traverse the memory using a void pointer. So we need to cast the memory allocated to one of the known types: int, char, short, or any user defined structure.

Say we need to traverse the memory using a int.

We do this by casting the void pointer to a int pointer like shown.

```
int *ptr ;

ptr = ( int  * ) malloc ( 2* sizeof ( int)  ) ;
```

In the above statement, the void pointer that is returned by malloc, is type casted to int * pointer. Then, we assign it to ptr which is int *.

In short, the above line does three things, all at once:

1. allocate a memory of 8 bytes = 2 times sizeof (int)
2. cast the memory to int *
3. make a int pointer ptr point to that memory

Because we allocated 2 int (= 8 bytes) , we can say ptr is now pointing to the first int (4 bytes) .

Now, we can assign a value to the ptr.

```
*ptr = 0x0a0b0c0d ;
```

which assigns 0x0a0b0c0d to the first int.

When we do ++ptr, we have advanced the pointer to the second int.

Now we assign

```
*ptr = 0xd0c0b0a ;
```

this means we have assigned 0xd0c0b0a to the *ptr which in this case it is pointing to the second int.

Though we explained with just two ints, in real life, the number of memory allocated is vastly huge.

Just to summarize: When we allocate memory using malloc, we get the memory from the heap, not from stack.

malloc int pointer (https://www.youtube.com/watch?v=896nrvtA_xE&feature=youtu.be)

(https://www.youtube.com/watch?v=896nrvtA_xE&feature=youtu.be)

Let us take another example.

```
struct _user {
    char name [ 12 ] ;
    int age ;
} ;
```

let us declare a structure pointer of type

```
struct    _user     *sPtr ;
```

Now we will allocate memory whose size is twice the size of a structure _user.

```
sPtr =  ( struct   _user   *  )  malloc  ( 2 * sizeof (  struct    _user  ) ) ;
```

now sPtr is now pointing to the dynamic memory whose size is twice the size of the structure.

sPtr is pointing to the first structure and we could assign values such as

sPtr->age = 10 ;

strcpy (sPtr->name, "John");

How do I advance the pointer to the next structure?

sPtr++ would make it point to the next cell, we could assign the values

sPtr->age = 20

strcpy (sPtr->name, "Jenny");

If we do sPtr++ one more time, the sPtr would be pointing beyond the allocated memory. When you do this, you may end up in a crash. So, it is programmers responsibility to make sure the pointers are within the allocated memory size.

malloc a structure (https://youtu.be/Rnwt--h7x7Y)

(https://youtu.be/Rnwt--h7x7Y)

memove :

#include <string.h>

void *memmove(void *dest, const void *src, size_t n);

The memmove() function copies n bytes from memory area src to memory area dest. The memory areas may overlap: copying takes place as though the bytes in src are first copied into a temporary array that does not overlap src or dest, and the bytes are then copied from the temporary array to dest.

calloc:

void *calloc(size_t nmemb, size_t size);

The calloc() function allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory. The memory is set to zero. If nmemb or size is 0, then calloc() returns either NULL, or a unique pointer value that can later be successfully passed to free().

realloc:

void *realloc(void *ptr, size_t size);

The realloc() function changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will not be initialized. If ptr is NULL, then the call is equivalent to malloc(size), for all values of size; if size is equal to zero, and ptr is not NULL, then the call is equivalent to free(ptr). Unless ptr is NULL, it must have been returned by an earlier call to malloc(), calloc() or realloc(). If the area pointed to was moved, a free(ptr) is done.

Here is the sample program :

/// (file) **dynamic.c (https://csus.instructure.com/courses/94454/files/14915339/download?wrap=1)** ⬇ (https://csus.instructure.com/courses/94454/files/14915339/download?download_frd=1)

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

#define MAX_SIZE 32
#define INCREASE_SIZE 10

int main ( void )
{

    unsigned char *src ;
    unsigned char *dest ;
    src = ( unsigned char * ) malloc ( MAX_SIZE ) ;
```

```
    dest = ( unsigned char * ) malloc ( MAX_SIZE ) ;

     memset ( src, 255, MAX_SIZE);  // assign 255 to cells

     memmove ( dest, src, MAX_SIZE);   // copy the values to dest pointer

     int i = 0;
     for ( ; i < MAX_SIZE ; i++ )   {
          printf ( "value at i=%d is %d \n", i, dest[i]);
     }

      dest = ( unsigned char * ) realloc ( dest, MAX_SIZE+INCREASE_SIZE );  // INCREASE
      memset ( dest + MAX_SIZE,  64,    INCREASE_SIZE);  // initialize with 64

      for (    ;    i < MAX_SIZE+INCREASE_SIZE   ;    i++ )    {
              printf ( "value at i=%d is %d \n", i, dest[i]);   // printing the whole dest
      }
    // two ways to free the memory.  The standard one is:
     free ( src ) ;


}
```

Processes Creation : Fork ( ) (video attached)

fork ( ) is a function that creates a new program with new process and new process ID. This results in two processes running. One is a parent process and the other is a child process.

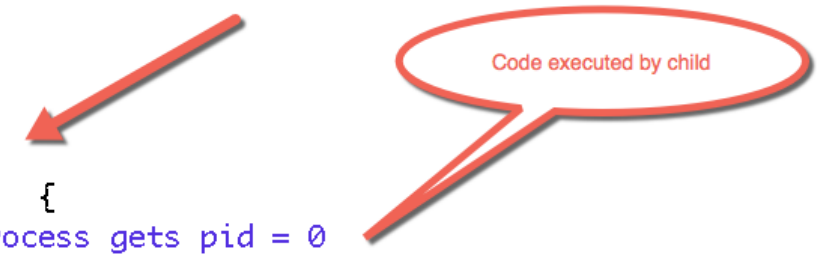
CHILD PROCESS CREATION USING FORK :

What the function fork ( ) does is , it dynamically clones the parent process bit by bit including status of variables. The entire memory, registers and status are all cloned and is identical to the parent process. Now at this time, we have two processes and they are identical. The child process has a process ID too. The execution of the child process starts after the line next to fork ( ) while the parent process also continues execution after the line next to fork ( ) .

When a fork function call returns creating a child process, **the return value is zero for the child process and non-zero for the parent process.**

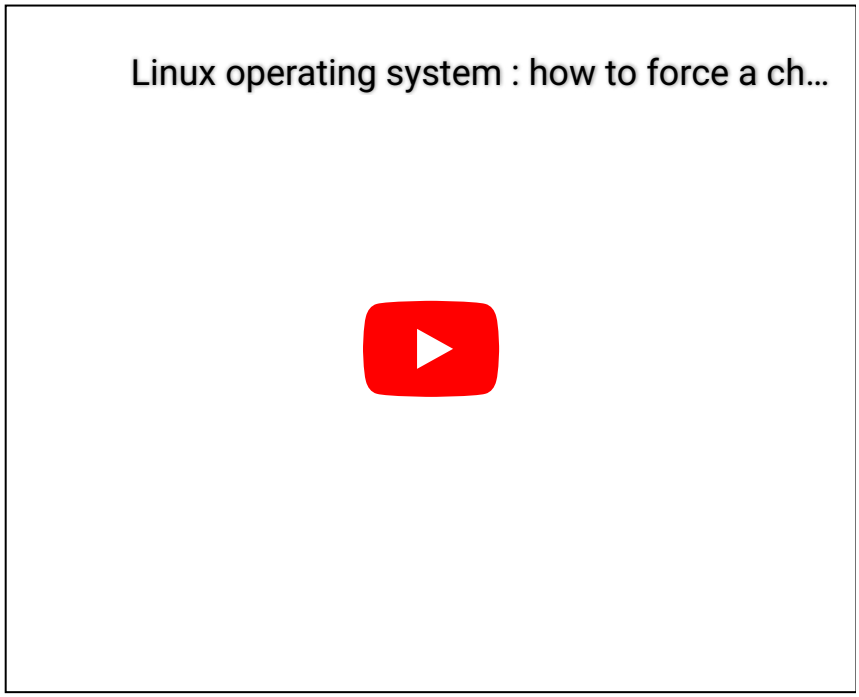
Look at this simple code : Figure

```
 2 #include  <stdio.h>
 3 #include  <sys/types.h>
 4
 5 void  main( void )
 6 {
 7       pid_t  pid;
 8
 9       pid = fork();
10       if (pid == 0)  {
11         // Child process gets pid = 0
12         printf("I am the child process, my id is:  %d ***\n", getpid(  )  );
13       }
14
15 }
```



In line 9, fork is called. When the function returns, the child process gets a return value of zero. The parent process gets the process ID of the child as the return value. Now, we have two programs executing at the same program. A very classic case of two processes running same program but executing two different blocks.

how to force child to execute a code after fork (https://www.youtube.com/watch?v=oiXhV6mwDgA)



Minimize Video

NOTE: **For the child processes, the value returned is zero. For the parent process, the value returned is the PROCESS ID of the child.**

when you run the above program, the child processes will continue executing the printf statement because the pid is zero. The parent process will not print the printf statement because the pid value is non-zero and is the process ID of the child.

In the next new program, now we have lines of code that is created to the else part.

```
void  main( void )
{
     pid_t  pid;

     pid = fork();
     if (pid == 0)  {
       printf("I am the child process, My id is: %d \n", getpid(  )  );
     } else {
       printf("I am the parent process My id is: %d \n", getpid(  )  );
       printf("My Child's id is:  %d ***\n", pid  ) ;
     }
}

c-prog>gcc ver1.c
c-prog>./a.out
I am the parent process My id is: 31079
My Child's id is:  31080 ***
I am the child process, My id is: 31080
c-prog>
```

We expanded the program to include a piece of code for the parent to execute. As you can see, the else part of the if statement is executed by the parent. The variable pid value printed by the parent is the child process ID. In the child process, you can print the process ID of the child using getpid ( ).

The main point is, when a fork is called :

it returns a value zero to the Child Process

and its return the procecss ID of the child to the parent.

There are two processes that are running now.

All variables , open files, system tables, almost everything is a duplicate in the two processes.

In the above program, the parent process is executing it's block first and doesn't wait for the child. Most of the time, the parent process should wait and check on the child process and it's statuses. This is done using the function wait.

Why should I wait for my child process ?

There might be a situation when the child would terminate before the parent would call wait. In this situation, the parent process has no way to figure out what happened to the child process. In this situation, the child process would remain a zombie state and resources allocated to the child would remain active in the kernel and wont be released for future processes. The kernel keeps a copy of the state of the zombie child. When the parent calls wait , the kernel responds to the function using the state it has and then releases the resources and child that is terminated will not be in a zombie state.

So, we need to call wait function in the parent after a fork function. See the note section of the man pages:

"A child that terminates, but has not been waited for becomes a "zombie". The kernel maintains a minimal set of information about the zombie process (PID, termination status, resource usage information) in order to allow the parent to later perform a wait to obtain information about the child. As long as a zombie is not removed from the system via a wait, it will consume a slot in the kernel process table, and if this table fills, it will not be possible to create further processes. If a parent process terminates, then its "zombie" children (if any) are adopted by **init (https://linux.die.net/man/8/init) process**, which automatically performs a wait to remove the zombies"

What could we do inside the Child process after a fork ?

Remember, after fork, there are two separate processes running. We could add and execute any code inside these processes. In real life, server side programming involve the parent forks a child to serve a request from a client. The child process serves the request from a client and dies after it is done serving. For example, http request from your browser will be a request to the server. The

server forks a child, the child serves the webpage to the browser and dies. The server goes on to the next request.

simple Algorithm of a web server :

```
do   {
request_received_from_browser ( ) ;
pid = fork ( )
   if ( pid == 0 ) {    //child process
     child serves the web page and dies
   }
  else  {  // Parent processes
     go back to serve more browser requests
 }

}  while ( 1 ) ;
```

Disadvantages with FORK : Creating fork is a very expensive operation: the system has to copy the entire program, it has to create a new process, manage the child and parent relationships, save the status of all variables. If two processes have to communicate (what is known as interprocess communication) , an extensive programming effort has to be put for the two processes to share data or work collaboratively. Most of these efforts have issues with portability too and management of code across system is a nightmare for software developers.

Open File Descriptors:

When a fork() is performed, the child receives duplicates of all of the parent's file descriptors. These duplicates are made in the manner of dup(), which means that corresponding descriptors in the parent and the child refer to the same open file description. These attributes of an open file are

shared between the parent and child. For example, if the child updates the file offset, this change is visible through the corresponding descriptor in the parent.

This is demonstrated in this example:

```
void  main(void)
{
     FILE *fp = fopen ( "data.txt", "w");

     pid_t   pid1,  pid;
     int     status;

     if ((pid1 = fork()) < 0) {
          printf("Failed to create child process 1\n");
          exit(1);
     }
     else if (pid1 == 0)  {
         sleep ( 4 );
         printf ( "  Child Process PID %d \n", getpid() );
         fprintf (fp, " Child created Process PID %d \n", getpid() );
         exit(0);
     }

     printf("Parent created  child PID1 %d \n", pid1);
     fprintf(fp, "Parent created  child PID1 %d \n", pid1);

     pid = wait(&status);
     printf("*** Parent sees child PID %d terminated ***\n", pid);
     fprintf(fp, "*** Parent sees child PID %d terminated ***\n", pid);
     exit(0);
}
```

FILE descriptor shared

```
[srivatss@athena:105]> gcc fileSharing.c
[srivatss@athena:106]> ./a.out
Parent created  child PID1 27546
  Child Process PID 27546
*** Parent sees child PID 27546 terminated ***
```

The system wide open descriptor table is demonstrated in this figure:

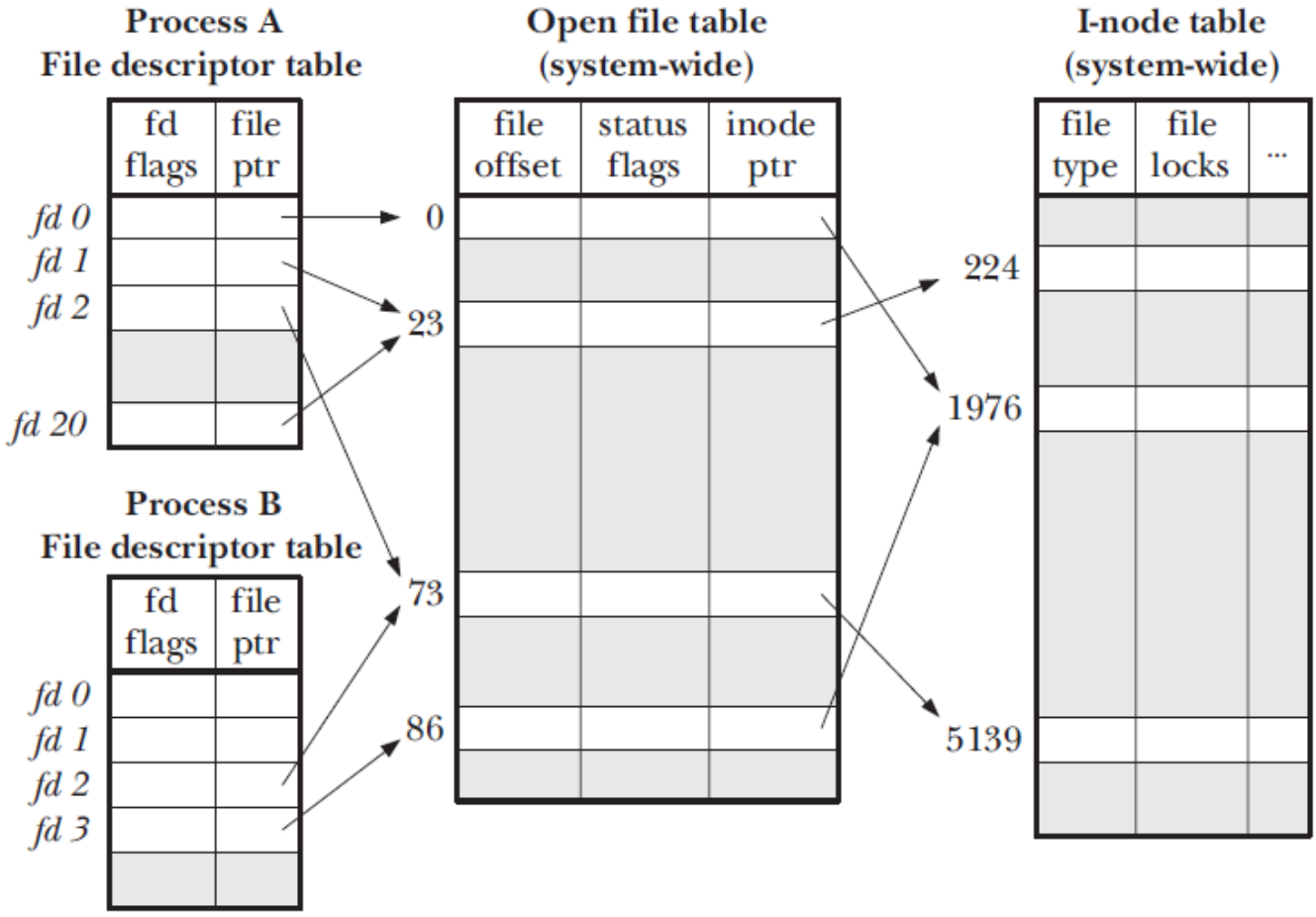
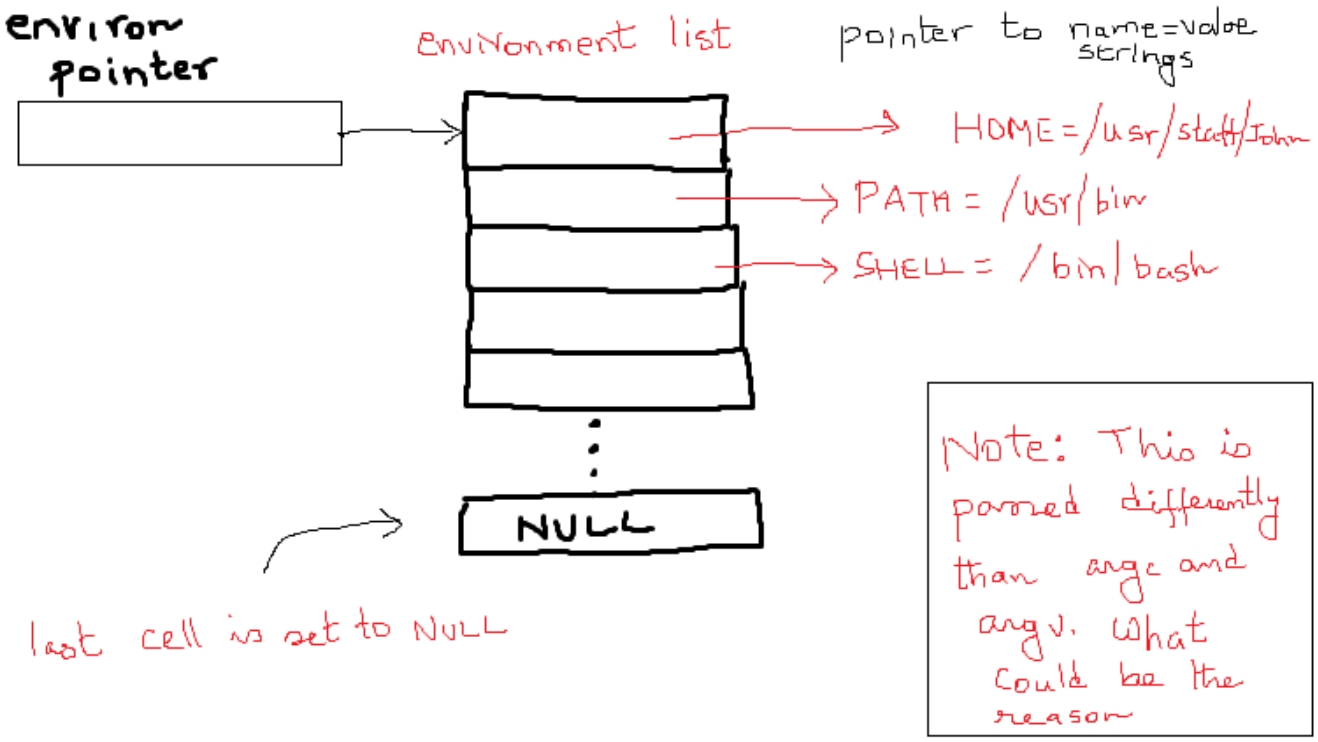


Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

environ variable



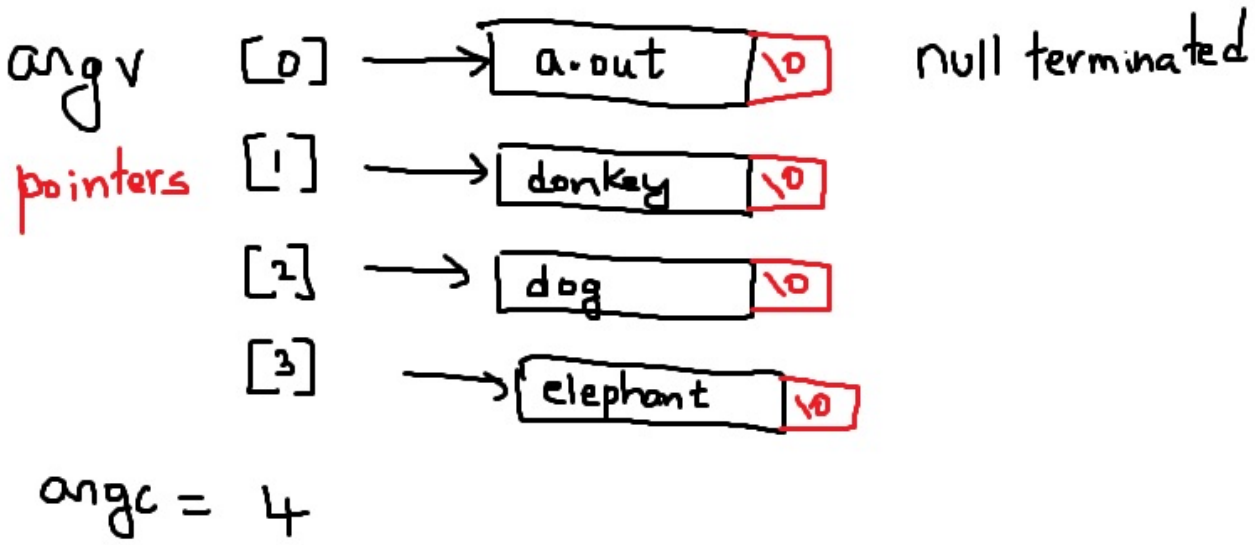
Sample program is:

```
#include <stdio.h>
#include <stdlib.h>
extern char **environ; // pointer to pointer global variable
                       // copied to our memory address by the OS

int main( int argc, char *argv [ ] )
{

    int i = 0;

     for ( i = 0 ; environ[i] != NULL  ; i++ )
       printf ("value at %d is %s \n",  i, environ[i] );

}
```

environ global variable is checked for not NULL

passing arguments to C program (argc argv)

a.out donkey dog elephant



To print the arguments:

```
for ( i = 1 ; i < argc ; i++ )
  printf ( "  The argument are %s \n", argv[i]);
```

Wait functions

Why do we need wait function:

here is the video:

https://www.youtube.com/watch?v=Tt5qpp3141U (https://www.youtube.com/watch?v=Tt5qpp3141U)

(https://www.youtube.com/watch?v=Tt5qpp3141U)

Sample Program using wait function

Some documentation about wait function in FORK

int status ;

wait (&status) ;

waitpid (pid_t pid, &status, options);

wait (&status) is same as waitpid (-1 , &status, 0) ;

pid = -1 means do wait for any of the children

option = 0 means do not wait if any child has not exited.

The two system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose
state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait

allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state (see NOTES below)

Sample Code using wait ( ), please note the highlighted calls are macros. You can actually see the expansion of the macro using -E option. The sample code creates a child. The child sleeps for 60 seconds. Parent process waits to check the status of the child.

```
int main( void )
{
      int status;
      pid_t pid;

    pid = fork();


    if (pid == 0) {
        printf(" *** Child process is %d ***\n", getpid( ) );
       sleep ( 60) ;
   }  else      {
                  wait ( &status);
                  if ( WIFEXITED ( status) )
                       printf ( "Child exited normally %d \n", WEXITSTATUS (status) );
                  else if ( WIFSIGNALED ( status) )
                      printf ( "Child exited by a Signal #%d \n", WTERMSIG (status) );
                }
 }
```

1. Try to run this program in the background. wait for 60 seconds.

2. Again, try to run the program in the background. terminate the client process using the kill command. Kill is the command to send a signal to another process in the same group. There are various signals Linux supports. We will discuss them soon

Sample code using waitpid Function

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

void main( void )
{
       int status;
       pid_t pid;

     pid = fork();
     if (pid == 0) {
       printf("Child process is %d \n", getpid( ) );
       sleep ( 2 );
    }  else       {
       waitpid (pid,  &status, WUNTRACED | WCONTINUED );
       if ( WIFEXITED ( status) )
           printf ( "Child exited normally %d \n", WEXITSTATUS (status) );
       else if ( WIFSIGNALED ( status) )
           printf ( "Child exited by a Signal #%d \n", WTERMSIG (status) );
    }
}
```

Process Creation: System ( ) function call

system ("ls") ;

This will execute the command ls

system ("pwd"); This will execute the command pwd

How system works : it forks, executes exec functions in the child process. While the child executes the function, the parent waits.

Exec Family of Functions (video attached)

There is a system function called exec (6 of them). exec functions will execute any program you pass it to it. You also need to pass parameters if any.

When you launch the new program mentioned in the exec, the new program will replace the caller in memory, though the process ID remains the same. It is as if you launched a new program.

Here is the document :

exec-family-functions.docx (https://csus.instructure.com/courses/94454/files/14915326/download?wrap=1)

Here is the video :

linux operating system fork execl family of functions execl execlp execle
(https://www.youtube.com/watch?v=duYojgFuT3s)

(https://www.youtube.com/watch?v=duYojgFuT3s)

EXEC FAMILY OF FUNCTIONS

A parent program can call a exec family of functions to launch a new program. The new program will replace the parent program, but not the process. The new program will get the process ID of the parent program. The segments such as text, initialized data, uninitialized data (bss), and stack of the calling process are overwritten according to the contents of the newly loaded program.

The parent program will not be able to check the status of the exec function on success. Only on error, the parent program can check and take remedial actions. We will discuss six functions, the difference between these functions is just in the format of the parameters passed. We also provided an example for each function.

execl - this function takes the command path, name and optional parameters and NULL parameters. execl ("/bin/ls", "ls", "-l", NULL)	execv – Very identical to execl, except the name and optional parameters and NULL parameters are passed separately in a array char *args [ ] = { "ls", "-l", NULL }; execv ("/bin/ls", args);
execle - In addition to execl, this also takes the environment variables in a NULL terminated string array.	execve - In addition to execv, this also takes the environment variables in a NULL terminated string array.
execlp – Same as execl , except it recognizes the path of the command using the $PATH variable. Absolute path is optional	execvp - Same as execv , except it recognizes the path of the command using the $PATH variable Absolute path is optional

The functions listed on the left side differ from the right side on one thing: function on the right side take optional parameters of the command in the array of strings. Whereas, you have to list them individually for functions on the left hand side, See the highlighted text in row 1. Functions that end in ‘e’ take the environment variables , and functions that end ‘p’ recognizes the $PATH variables for you to omit the absolute path of the command.

EXECL

```
1.  int execl       (   const char *file, const char *arg0, ..., NULL);
```

file : is the filename of the file that contains the executable image of the new process.
a**rg0, ..., NULL** : is a variable length list of arguments that are passed to the new process image. Each argument is specified as a null-terminated string, and the list must end with a NULL pointer. The first argument, arg0, is required and must contain the name of the executable file for the new process image. If the new process image is a normal SAS/C main program, the list of arguments will be passed to argv as a pointer to an array of strings.

Example 1:

```
main  ( )

{
  execl ( "/usr/bin/cal", "cal", "2017", NULL );
}

another example, echo prints SYSTEM PROGRAMMING

execl ( "/bin/echo", "echo", "SYSTEM PROGRAMMING", NULL );
```

EXECLE

```
2. int execle  ( const char *file,
                 const char *arg0, ..., NULL,
                 char *const envp [ ]  ) ;
```

file : is the filename of the program that is to be launched.

arg0, ..., NULL : is a variable length list of arguments that are passed to the new program. Each argument is specified as a null-terminated string, and the list must end with a NULL pointer. The first argument, arg0, is required and must contain the name of the executable file for the new process image. The number of strings in the array is passed to the main() function as argc.

envp is a pointer to an array of pointers to null-terminated character strings. A NULL pointer is used to mark the end of the array. Each character string pointed to by the array is used to pass an environment variable to the new process image. Each string should have the following form:

" *var* = *value* "

Example 1:

```
main  ( )
{
  char *env[ ] =
      { "SYSTEM PROGRAMMING", NULL };

   execle ( "/bin/echo", "echo", env[0], NULL, env );
}

another example
execle ( "/bin/echo", "echo", environ[3], NULL, environ );
```

EXECLP

```
3. int execlp(const char *path, const char *arg0, ..., NULL);
```

path : identifies the location of the new process in the system. If the path argument contains a slash (/), it is assumed that either an absolute or a relative pathname has been specified. If the path argument does not contain a slash, the directories specified by the PATH environment variable are searched in an attempt to locate the file.

arg0, ..., NULL : is a variable length list of arguments that are passed to the function. Each argument is specified as a null-terminated string, and the list must end with a NULL pointer. The first argument, arg0, is required and must contain the name of the executable file.

Example 3:

```
main  ( )
{
  execlp ( "ls", "ls", "-lF", NULL );
}

another example
execlp ( "./echo_1.sh", "./echo_1.sh", "Jack", "Sam", "Pam", NULL );
```

EXECV

```
4. int execv(const char *file, char *const argv[] ) ;
```

file : is the filename of the file that contains the executable image of the new process.
argv : is a pointer to an array of pointers to null-terminated character strings. A NULL pointer is used to mark the end of the array. Each character string pointed to by the array is used to pass an argument to the new process image. The first argument, argv[0], is required

Example:

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

EXECVE

5. **int execve (const char **filename*, char *const *argv [ ] ,**
char *const *envp [ ]) ;

filename : is the filename of the program to be launched by the function
argv : is a pointer to an array of pointers to null-terminated character strings. A NULL pointer is used to mark the end of the array. Each character string pointed to by the array is used to pass an argument to the new process image. The first argument, argv[0], is required and must contain the name of the executable file for the new process image.
envp : is a pointer to an array of pointers to null-terminated character strings. A NULL pointer is used to mark the end of the array. Each character string pointed to by the array is used to pass an environment variable to the new process image.

Example :

```
main  ( )
{
 char *paramList[ ] = { "echo", environ[4], NULL };
 execve ( "/bin/echo", paramList, environ );
}
```

EXECVP

```
6. int execvp ( const char *path, char *const argv [ ] ) ;
```

path: identifies the location of the new program. If the path argument contains a slash (/), it is assumed that either an absolute or a relative pathname has been specified. If the path argument does not contain a slash, the directories specified by the PATH environment variable are searched in an attempt to locate the file.
argv : is a pointer to an array of pointers to null-terminated character strings. A NULL pointer is used to mark the end of the array. Each character string pointed to by the array is used to pass an argument to the new process image. The first argument, argv[0], is required and must contain the name of the executable file for the new process image.

Example:

```
main  ( )
{
 char *paramList[ ] =  { "ls", "-l", NULL} ;
  execvp ( "ls", paramList );
}
```

What is Static and Dynamic Linked Programs (with Video)

Libraries are files containing the object files of various programs that are used to make compilation much faster. In C, there are two main types of libraries: *static* and *dynamic*. *Static libraries* are specifically called in the linking phase of compilation and tend to be both bigger and slower than dynamic libraries.

Dynamic libraries, on the other hand, do not require the code to be copied. They are not loaded automatically when a program starts, and are, instead, linked when the program is run. This causes them to be more efficient and smaller than static libraries. *Dynamic libraries* are also known as *shared libraries* because the code is shared by the programs that use it; each program, however, maintains its own stack and heap, keeping all running programs separate from one another.

Statically Linked Programs

In statically-linked programs, all code is contained in a single executable module. Static libraries are files that contain object files called modules or members. Library references are more efficient because the library procedures are statically linked into the program. Static linking increases the file size of your program, and it may increase the code size in memory if other applications, or other copies of your application, are running on the system. Static libraries are used when **stability** and **loadtime** are most desired. There are no dependencies required — if you have successfully compiled the library with no linking errors, it will work indefinitely. Static libraries are typically named with an .a extension. .a stands for archive.

Creating a Static Library using the archive command ar

gcc -c questions.c verifyResponse.c getARandom.c

ar rcs lib**ngdemo**.a getARandom.o questions.o verifyResponse.o

gcc main.c -o **ngb** -L. -l**ngdemo**

./ngb

The option in the ar command are:

r inserts files into the archive, replacing any existing members whose names matches that being added. New members are added at the end of the archive.

s creates and updates the map that cross-references symbols to the members in which they are defined

c creates the archives if it doesn't exist from files, suppressing the warning ar would putput if archive doesn't exist

Dynamically Linked Programs

The operating system provides facilities for creating and using dynamically linked shared libraries. With dynamic linking, external symbols referenced in user code and defined in a shared library are resolved by the loader at load time. When you compile a program that uses shared libraries, they are dynamically linked to your program by default.

The idea behind shared libraries is to have only one copy of commonly used routines and to maintain this common copy in a unique shared-library segment. These common routines can significantly reduce the size of executable programs, thereby saving disk space. The shared library code is not present in the executable image on disk, but is kept in a separate library file. Shared code is loaded into memory once in the shared library segment and shared by all processes that reference it. Dynamically linked libraries therefore reduce the amount of virtual storage used by your program, provided that several concurrently running applications (or copies of the same application) use the procedures provided in the shared library. They also reduce the amount of disk space required for your program provided that several different applications stored on a given system share a library. Other advantages of shared libraries are as follows:

- Load time might be reduced because the shared library code might already be in memory.
- Run-time performance can be enhanced because the operating system is less likely to page out shared library code that is being used by several applications, or copies of an application, rather than code that is only being used by a single application. As a result, fewer page faults occur.
- The routines are not statically bound to the application but are dynamically bound when the application is loaded. This permits applications to automatically inherit changes to the shared libraries, without recompiling or rebinding.

Disadvantages of dynamic linking include the following:

- From a performance viewpoint, there is "glue code" that is required in the executable program to access the shared segment. There is a performance cost in references to shared library routines of about eight machine cycles per reference. Programs that use shared libraries are usually slower than those that use statically-linked libraries.
- Dynamically linked programs are dependent on having a compatible library. If a library is changed (for example, a new compiler release may change a library), applications might have to be reworked to be made compatible with the new version of the library. If a library is removed from the system, programs using that library will no longer work

How Libraries are loaded by ldconfig with admin privileges

On GNU glibc-based systems, including all Linux systems, starting up an ELF binary executable automatically causes the program loader to be loaded and run. On Linux systems, this loader is named /lib/ld-linux.so.X (where X is a version number). This loader, in turn, finds and loads all other shared libraries used by the program.

The list of directories to be searched is stored in the file **/etc/ld.so.conf**.

Searching all of these directories at program start-up would be grossly inefficient, so a caching arrangement is actually used. The program **ldconfig** by default reads in the file /etc/ld.so.conf, sets up the appropriate symbolic links in the dynamic link directories (so they'll follow the standard conventions), and then writes a cache to /etc/ld.so.cache that's then used by other programs. This greatly speeds up access to libraries. The implication is that ldconfig must be run whenever a DLL is added, when a DLL is removed, or when the set of DLL directories changes; running ldconfig is often one of the steps performed by package managers when installing a library. On start-up, then, the dynamic loader actually uses the file /etc/ld.so.cache and then loads the libraries it needs.

Creating a Shared Library without admin privileges

Creating a shared library is easy. First, create the object files that will go into the shared library using the gcc -fPIC or -fpic flag. The -fPIC and -fpic options enable ``position independent code'' generation,
a requirement for shared libraries. We outline simple steps to create a shared library without having to have superuser permission.

The compilation generates object files (using -c), and includes the required -fPIC option:

```
gcc -fPIC -c questions.c
gcc -fPIC -c verifyResponse.c
gcc -fPIC -c getARandom.c
gcc -shared -o libngdemo.so getARandom.o verifyResponse.o questions.o
gcc -fPIC -c main.c
gcc -o ngb main.o ./libngdemo.so
```

Here is the video:

https://www.youtube.com/watch?v=jiShZsP1IHg (https://www.youtube.com/watch?v=jiShZsP1IHg)

(https://www.youtube.com/watch?v=jiShZsP1IHg)

Concept behind calling library C routines

This webpage is evolving , so it is not in any quiz, assignment or exams. THIS IS ENTIRELY FYI

You might have to wonder how did the executable finds printf or any other library functions when I didn't include the library code in my C language. I didn't include the library during compilation too.

So , if you have a code like this

printf ("Hello \n") ?

printf ("World \n") ?

Where did my system find the code for printf ? How is it finding it ? You are absolutely right and as a computer scientist, you should wonder.

The compiler doesn't know where the printf code exists because the library Code will be loaded at any random place. So, it is going to insert a FILL IN THE BLANK for the dynamic loader to fill the blank.

Consider, all the fill in the blanks created by the compiler will be a table called Global Address Table (GOT) for the dynamic linker (DL) to fill it up. But compiler maintains another table called Procedure linkage table (PLT) where all procedures you called are listed, one entry for each function. Along with the procedure name in the PLT , a pointer to the GOT table is maintained. Something like this (I am trivializing it here)

```
main ( )
{
 printf .....
 printf
```

scanf .....

}

PLT

printf Index 0 to GOT

scanf index 1 to GOT

GOT

0 LOAD DL and FILL THIS BLANK

1 LOAD DL and FILL THIS BLANK

As you can see , GOT indexes 0 and 1 are calling dynamic loader to fill the blanks. When your main function is executing at printf, it goes to PLT and finds index 0 at GOT. But at GOT index 0, dynamic loader is launched and kicks in.

Dynamic loader now does the finds the library you need. You can also find this by giving this command

ldd a.out <enter>

DL goes on the hunting trip to find the printf function and of course it's address you called in the C library. It replaces the address at GOT 0 with the actual address of printf and executes printf for you. But you thought you are doing all this work :)

Here is the video I made on the gist of the DL

https://www.youtube.com/watch?v=r4auCn-_axU (https://www.youtube.com/watch?v=r4auCn-_axU)

A real world example to better explain this :

You made a reservation at the hotel. On the day of the arrival, you go to the checkin counter. The guy at the counter gives you room key and number. You spend the next day and return to your hotel. Do you ask the guy for the room number or do you already know it ? This is how DL, acting like the guy at the counter. The first time you call printf, DL kicks in and gives you the address. Next time you call, you remember the address and execute printf directly without calling DL.

Once the DL is done finding the printf, it will update the GOT table like this

GOT

0 Here is the address of printf (next time, don't bother me :))

1 LOAD DL and FILL THIS BLANK

I will upload the actual assembly code and the video later <stay tuned >

Declaration of String and Initialization

There are many ways to declare a string:

Method 1:

char state [6] ; // just a declaration of an array

is an array of characters. We initialize the individual cells using single quote characters, like

state[0] = 'U' ;

state[1] = 't' ;

state[2] = 'a' ;

state[3] = 'h' ;

state[4] = '\0' ; // IT IS OUR RESPONSIBILITY TO ASSIGN A NULL

Note, the last cell should be tagged with NULL character '\0' so we know the end of the string. The NULL character will not be part in the computation of the length of the string. For instance, the length of the string above is 4 (excluding the NULL character) .

Method 2:

In this definition,

char states [6] = "UTAH " ; // PAY ATTENTION TO THIS.

There are several things happening behind the scenes here. First, a constant string is made in memory "UTAH" . Secondly, the array variable "states" is defined and allocated. Thirdly, the array is initialized by copying the constant string "UTAH" . In effect, we are defining a two strings: one constant and one array of characters.

Note here , we defined a string "UTAH" with double quotes, not single quotes. constant strings should be within double quotes. When you change the contents of the array, you are not changing the constant string, but the contents of the array only. The constant string cannot be changed

METHOD 3

In this method, we use the scanf function. Remember, scanf function requires the address of the variable to be passed along with the conversion specifier. In strings, the conversion specifier is %s referring to strings. We know by declaration of an array, the array name is itself a pointer constant and it is also the address of the first cell.

Luckily, we just have to pass this array name as the address of the array.

For instance,

char state [6] ;

in the above definition, state is the array name and is also the address of the first cell and is also a pointer constant.

Now, our scanf function becomes

scanf ("%s", state) ;

not scanf ("%s", &state) ; // which is incorrect

METHOD 4:

In this method, we simply use the strcpy function to initialize the array. The strcpy function takes two parameters in the form

char * strcpy(char *destinationBuffer, char *sourceBuffer

);

char state[6] = "UTAH" ;

char myState [6] ;

strcpy (myState, state) ;

The destination string should be as big as the source or can be longer. The strcpy copies including the NULL character from the source buffer. The function also returns the pointer to the destination array.

METHOD 5:

Method 5:

Much similar to method 1, we can declare and initialize an string as

```
char states [ ]  = { 'U', 't', 'a', 'h', '\0' } ;
```

Here we declare 5 cell array and initialize the cells using the character including the NULL character.

But if do

```
char states [  6 ]  = { 'U', 't', 'a', 'h' } ;
```

then you can skip the NULL characters because the fifth and the sixth cell will automatically assigned zero, the NULL character.

String Library

Sample code

```
// THE CODE WE WROTE IN THE VIDEO
        char str1[20] = "CSUS STATE" ;
        char ch = 'b' ;


        if ( strchr ( str1, ch ) )
          printf ( "ch is present str1 \n");
        else // NULL  = 0
          printf ( "ch is not present str1 \n");
```

The various functions we would use in programs are:

Video on strchr

https://youtu.be/uYUWf_YYE4U (https://youtu.be/uYUWf_YYE4U)

(https://youtu.be/uYUWf_YYE4U)

How to use strcat and strcpy :

https://youtu.be/OoTvf1PPaQ0 (https://youtu.be/OoTvf1PPaQ0)

(https://youtu.be/OoTvf1PPaQ0)

How to use strstr:

https://youtu.be/S_uAtSdXnso (https://youtu.be/S_uAtSdXnso)

(https://youtu.be/S_uAtSdXnso)

Sample code to use strstr

The prototype of strstr function is defined as

```
char *    strstr(const char *s1, const char *s2) ;
```

The strstr() function locates the first occurrence of the null-terminated string s2 in the null-terminated string s1. The strcasestr() function is similar to strstr(), but ignores the case of both strings.

RETURN VALUES FROM MAN PAGES: If s2 is an empty string, s1 is returned; if s2 occurs nowhere in s1, NULL is returned; otherwise a pointer to the first character of the first occurrence of s2 is returned.

// THE CODE WE WROTE IN THE VIDEO

```
char str1[20] = "Sac State";
```

```
  char str2[20] = "ate";


if ( strcasestr ( str1, str2) ) // THIS IGNORES CASES
    printf ("%s is a substring of %s\n", str2, str1);
else // NULL
    printf ("%s is not a substring of %s\n", str2, str1);
```

How could we implement strstr :
strstr implementation (https://www.youtube.com/watch?v=NtYJ49ZsNhY)

(https://www.youtube.com/watch?v=NtYJ49ZsNhY)

ctype.h functions

To use these functions, include

#include <ctype.h>

In many programming tasks, you may want to determine if a character is a letter, digit, symbol and so on.

Types of character sets in our ASCII Table are

- alphabetic characters : A-Z, a-z
- alpha numeric character : A-Z, a-z, 0-9
- digits : 0 – 9
- hexadecimal 0-9, A-F, a-f
- lower case a – z, upper case A – Z
- punctuation - ~ @ # $ % ^ & * ( ) - _ + { } | \ < > ? / : ; " ' , .
- space characters
- control characters

int isalpha (int value) ;

The isalpha() function returns non-zero if the argument value is >=65 and <=90 and >= 97 and <=122 tests false and returns zero otherwise.

int tolower (int value) ;

This converts the upper case letter to lower case letter. if the letter is not upper case, the value is returned unchanged.

int toupper (int value) ;

This converts the lower case letter to upper case letter. if the letter is not lower case, the value is returned unchanged.

int islower (int value) ;

The islower ( ) function returns non-zero if the argument is a lower case letter and returns zero if the argument tests true

int isupper (int value) ;

The isupper ( ) function returns non-zero if the argument is a upper case letter and returns zero if the argument tests true

int isalnum (int value) ;

This function check if the argument is any character in the set A-Za-Z0-9. In a way, we can rewrite this function as

```
if ( isdigit ( ch ) || isalpha ( ch ) )
    return non-zero ;
else
    return zero ;
```

int isprint (int c) ; This functions tests if the character is printable or not. The following are printable characters

<space> ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [\ ] ^ _ ` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~

int ispunct (int c) ; this function checks if the character is a punctuation character or not . The following are punctuation characters

! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ` { | } ~

int iscntrl (int c) ; This function checks if the argument is a control character. A control character is any ASCII value from 0 to 31 and the ASCII value 127.

To use all these functions, you want to include the header file

<ctype.h>

t y p e s ---------- c h a r a c t e r --------- s e t s (https://www.youtube.com/watch?v=cxABDXqC63M)

(https://www.youtube.com/watch?v=cxABDXqC63M)

STRUCTURES

Structures are derived data types—they're constructed using objects of other types. Structure is also a variable. A Structure can also be viewed as a container to hold many types of variables. Unlike arrays which hold only one type of variable, structures enable us to store multiple variables.

The various topics we will talk are

- Declare them with and without Tags
- Access members of structures
- Initialize structures
- Computing Size of structures, use sizeof operator
- Assign one structure to another
- Compare one structure with another - X, compare
- Pass by Value (default)
- Pass by Reference - send the address
- Having Pointers as members
- accessing structures using pointers
- accessing array of structures

we define a structure type as,

```
struct   _point {
    int x ;  // member variables
    int y ; // member variables
}  ;
```

Here struct is a keyword to describe a structure variable. x and y are member variables , each is a type of int. _point is a tag. Structure definitions should always end semicolon.

Member variables need not be same type too. Consider this structure,

```
struct  _profile {
     int age;
    char name [ 10 ] ;
```

} ;

In the above definition, we have a char variable and an int variable as members. _profile is a tag.

NOTE: There is no memory allocated to the structure, because it is just a type definition.

To define new structure variables using the above definitions, we could do

```
struct   _point  point;
struct   _profile   user1;
```

Here, point is a structure variable and user1 is also a structure variable.

So, what is the type of the variable - point ? It is struct _point and the type of user1 is struct _profile .

Using the definition of the struct _point , I can define new variables as

```
struct  _point  pt1,   pt2,  *ptr ;
```

In the above definition, we have pt1 and pt2 as structure variables and ptr is a pointer of type struct _point .

NOTE: No two variables of a struct can have the same name.

How do we access member variables using DOT operator

The Dot operator is represented by the symbol . To write values to the member variables, we use the DOT operator, like this

```
pt1.age = 20 ;
strcpy ( pt1.name, “John”) ;  // Note we cannot do pt1.name = "John"
```

similarly , we can read values of the variables like

```
int x = pt1.age ;
char myAge [10 ] ;
```

strcpy (myAge, pt1.name) ;

Although structure names should be distinct, we may use the same name for members in different structures. By using the DOT operator, we can access the member variables uniquely.

The dot operator is mandatory to access a member variable even if the name of the variable is not defined elsewhere.

Declaring structures with and without tag names.
Consider

```
struct {
   int x ;
   int y ;
} x, y, z ;
```

is same as

```
struct  _point {
   int x ;
   int y ;
} x, y, z ;
```

But the latter method with tag name is better and preferred as we can declare more variables like this

```
struct _point  a, b, c ;
```

The other advantage is you can pass the struct variable to functions as the type is struct _point ;

The disadvantage with declaring structures without tags is

- you cannot define new variables and
- you cannot pass them to functions.

IT IS ALWAYS RECOMMENDED TO DECLARE STRUCTURES WITH TAG NAMES

Initialize Structure

We will discuss several methods to assign individual members with values.

Method 1: This is tedious way of initializing.

```
// method 1 : initialize individual members
  pt1.age = 40;
  strcpy ( pt1.name, "John") ;
```

In the above method, we use the dot operator. Pay attention to the use of strcpy method to copy.

Method 2: We could also initialize the members using declaration such as

```
struct _profile p1  = { 30, "John" } ;  // order is important
```

As discussed, this method depends on the order of definition of the member variables, compiler will not make a guess if you mix the order like
```
struct _profile p1  = { "John" , 30} ;  // this is incorrect.
```

That will result in compiler warning

Method 3: The other method is to assign one structure to another structure

```
struct  _profile p4 = p1 ;  // is already initialized, so we copy into p4
```

technically, this copies bit by bit from p1 to p4.

The last method is of course , we use the scanf function to initialize the initialize each and every individual members. but this is not recommended.

Sizeof Structures

One possible gotcha with structures is , you cannot sum the size of individual members to determine the size of a structure. It is machine dependent, it creates members on the word boundary, especially if it contains non-char type members. Generally , it is 4 bytes.

Let us consider a simple case:

```
struct  _user {
    int age ;
    char name[12] ;
 } ;
```

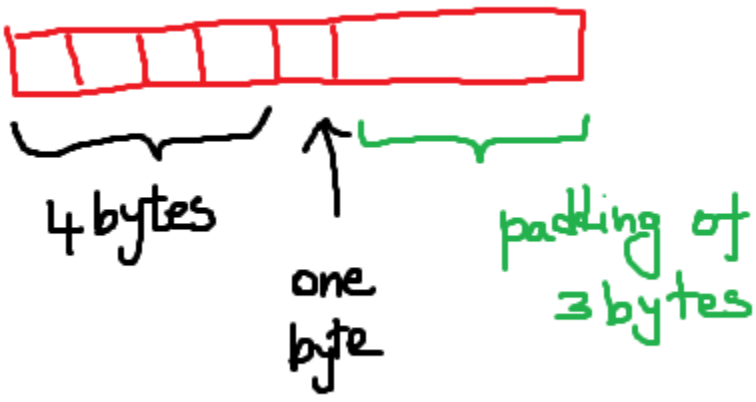
The size of this structure can readily be determined to be : 20 bytes , because int (4 bytes) plus 12 chars. This is very trivial.

But not so when it contains char and int variables , such as

```
struct  _user {
    int age ;
    char gender;
 } ;
```

In this case, it is not 5 bytes, it could be 8 bytes (or 6 bytes in some sysems) with a padding of 3 additional bytes next to the gender variable.

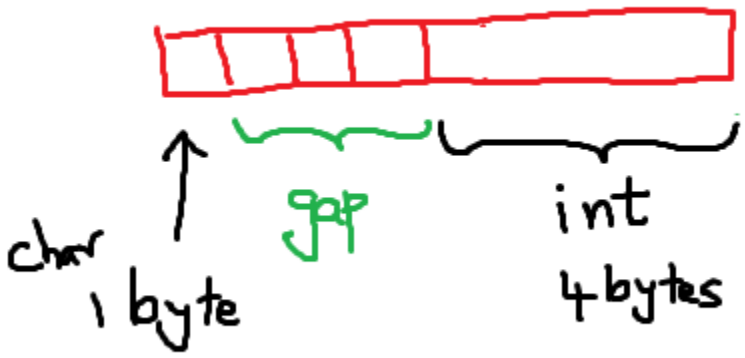
Here is the picture :



But if you change the order of the members

```
struct _person {
    char gender ;
    int age ;
 } ;
```

there will be a gap of 3 bytes.



The values in these gaps and paddings will be garbage. This is one of the main reasons you cannot compare two similar structures.

ASSIGN STRUCTURES and COMPARE STRUCTURE

Because structures have gaps or paddings, and these have garbage values, we could assign a structure variable to another similar type structure as it just a bit by bit copy. But you cannot compare two similar structures because during initialization it may have garbage values. But we cannot compare structures using any of the relation operators such as ==, <=, <, >, >=

Using Typedef structures

C provides the typefef construct , which lets the programmer provide a synonym for either a built-in or user defined data type. Although typedef may be used with any data tye, structures are generally used with typedef. typedef are just an alias to a type. For instance,

typedef int MyINT ;

MyInt becomes a synonym for int. Subsequently, we can declare new int type variables as

MyINT age ;

In the above definition age is a variable of type MyINT. Note, the syntax of a typedef: First comes the keyword typedef, then the data type and followed by the user provided name for this data type.

A typedef is used only to create a synonym for a data type. By defining a tyepdef, we are not allocating any memory.

Similarly, we could define a structure using typedef as

```
typedef  struct {
   int x,  y ;
} Point_t  ;
```

To define a new variable - point, we simply say
Point_t p1, *ptr ;

It is not generally recommended to define a structure without a tag name though typedef definition above is sufficient. So, let us refine the above definition as

```
typedef   struct   _point  {
   int x,  y ;
} Point_t  ;
```

Now, I can define variables as
Point_t   p1, p2, *ptr ;

The above definition is generally used in nested structures as we will see soon.

Nested Structure

Structures can have other type of structures as members , this is known as nested structures.

```
typedef struct _cars {
    char make [ 12 ];
    char model [ 12 ];
} Cars_t;

typedef struct _person {
   char name[12];
   int age;
   Cars_t cars [ 2] ;
   char ch; // padding of 3 bytes , each cell
 } Person_t;
```

Pointers to Structures
C provides pointers to structure variables and a special pointer operator -> for accessing members of structures.
Consider this structure

```
typedef struct _ram  {
  float price;
  int size ;
} Ram_t ;

Ram_t  p1, p2, *ptr;
```

we have two structure variables p1 and p2 of type Ram_t . We also have a pointer ptr defined.

```
/* we assign the address of p1 to ptr */
ptr = &p1 ;

/* assign values to price and ramSize */
(*ptr).price = 43.99;
( *ptr).size = 32 ; // GB
```

Some explanation : First we dereference ptr to get the address of p1. Then, we use . operator to access the individual members.

We need the parenthesis to enclose *ptr because the dot operator has higher precedence than the asterisk, resulting in an operation *(ptr.price) which is definitely wrong because price is not a pointer.

Because the syntax (*ptr).price is very clumsy, C provides alternate syntax with -> (pointer operator) like

ptr->price = 43.99
ptr->size = 32

Please note there shouldn't be any space between – and >

This pointer to structure and accessing members using -> is little tricky, so lot of practice is needed.

Structure : Pass By Value and Pass by Reference

We can pass structures to functions by value much similar to other variables. We do not pass the contents, but the copy of the variable. The invoked function can change the copy, but not the actual value of the variable that was passed.

Consider this example,

```
typedef struct _profile {
     char name [ 16 ] ;
     int age ;
}  Person ;

void printProfile ( struct _profile p1 )
{
    // a copy of the struct is passed in
    p1.age ++ ;  // no effect in the main function variable p.age

     printf  ( " age = %d name=%s \n" , p1.age, p1.name ) ;
}
main ( )
{
     Person p = { "Sam", 10 } ;
```

```
    printProfile ( p ) ;
}
```

If an array is present in the structure as a member, the array is also sent as value when the structure is passed to function.

Structures can also be passed by reference.

Array of Structures

consider this simple int array

```
int data[3] = { 30, 40, 50 };
int *ptr = data ;  // make ptr point to data
```

We can print the values of data using the pointer ptr

```
for ( i = 0 ; i < 3  ; i++ )  {
     printf ( " %d \n", *ptr ) ;
    ptr++ ; // advance the pointer to the next cell
}
```

*In the above code, *ptr prints the cell data and ptr++ is advanced to the next cell because the ptr is type int and the array is also type int.*

We can define array of structures similarly.

We can define array of structures

```
struct _data {
  char name[12] ;
  int age ;
 } ;

struct _data *ptr ,  dataArray [ 3 ] =
            { "Carolyn", 30, "Barry", 50, "Pamela", 40 };
```

Here I defined an array of structure and Initialized them during declaration too.

NOTE HOW I defined them. You have to pay utmost care in making sure the order of the definition of the members.

Now let us point ptr to the first cell or the address of the first cell and first member

```
  ptr = dataArray ;
  // ptr = &dataArray[0] ; // alternate way
```

Generally the first method is preferred

```
// METHOD 1 ,  we use the pointer arithmetic to browse
// each and every cell
for ( i = 0 ; i < 3 ; i++ )  {
    printf ( "%d %s \n", ptr->age, ptr->name);
    ptr++;
  }

// METHOD 2, we use pointer offset to navigate
  ptr = dataArray ;
  for ( i = 0 ; i < 3 ; i++ )
    printf ( "%d %s \n", (ptr+i)->age, (ptr+i)->name) ;

// METHOD 3,  we use pointer as an array
  ptr = dataArray ;
  for ( i = 0 ; i < 3 ; i++ )
    printf ( "%d %s \n", ptr[ i ] .age, ptr [ i ] .name);

// METHOD 4, using the array itself
  for ( i = 0 ; i < 3 ; i++ )
    printf ( "%d %s \n",
              dataArray[i].age, dataArray[i].name);

// METHOD 5 // not a elegant method
  ptr = dataArray ;
  for ( i = 0 ; i < 3 ; i++ )
    printf ( "%d %s \n", (*(ptr+i)) .age, (*(ptr+i) ). name);
```

If you are using array, you could use Method 4. If you are using pointer, Method 1 can be used. They are the standard method of accessing members variables using pointers

Processes Creation : Fork ( ) (video attached)

fork ( ) is a function that creates a new program with new process and new process ID. This results in two processes running. One is a parent process and the other is a child process.

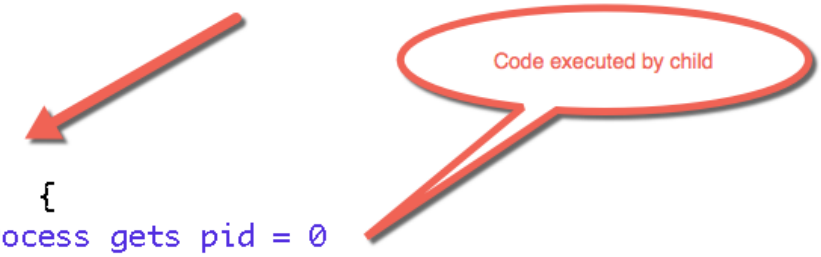
CHILD PROCESS CREATION USING FORK :

What the function fork ( ) does is , it dynamically clones the parent process bit by bit including status of variables. The entire memory, registers and status are all cloned and is identical to the parent process. Now at this time, we have two processes and they are identical. The child process has a process ID too. The execution of the child process starts after the line next to fork ( ) while the parent process also continues execution after the line next to fork ( ) .

When a fork function call returns creating a child process, **the return value is zero for the child process and non-zero for the parent process.**

Look at this simple code : Figure

```
 2 #include  <stdio.h>
 3 #include  <sys/types.h>
 4
 5 void  main( void )
 6 {
 7      pid_t  pid;
 8
 9      pid = fork();
10      if (pid == 0)  {
11        // Child process gets pid = 0
12        printf("I am the child process, my id is:  %d ***\n", getpid(  )  );
13      }
14
15 }
```



In line 9, fork is called. When the function returns, the child process gets a return value of zero. The parent process gets the process ID of the child as the return value. Now, we have two programs executing at the same program. A very classic case of two processes running same program but executing two different blocks.

how to force child to execute a code after fork (https://www.youtube.com/watch?v=oiXhV6mwDgA)

(https://www.youtube.com/watch?v=oiXhV6mwDgA)

NOTE: **For the child processes, the value returned is zero. For the parent process, the value returned is the PROCESS ID of the child.**

when you run the above program, the child processes will continue executing the printf statement because the pid is zero. The parent process will not print the printf statement because the pid value is non-zero and is the process ID of the child.

In the next new program, now we have lines of code that is created to the else part.

```
void  main( void )
{
     pid_t  pid;

     pid = fork();
     if (pid == 0)  {
       printf("I am the child process, My id is: %d \n", getpid(  )  );
     } else {
       printf("I am the parent process My id is: %d \n", getpid(  )  );
       printf("My Child's id is:  %d ***\n", pid  ) ;
     }
}

c-prog>gcc ver1.c
c-prog>./a.out
I am the parent process My id is: 31079
My Child's id is:  31080 ***
I am the child process, My id is: 31080
c-prog>
```

We expanded the program to include a piece of code for the parent to execute. As you can see, the else part of the if statement is executed by the parent. The variable pid value printed by the parent is the child process ID. In the child process, you can print the process ID of the child using getpid ( ).

The main point is, when a fork is called :

it returns a value zero to the Child Process

and its return the procecss ID of the child to the parent.

There are two processes that are running now.

All variables , open files, system tables, almost everything is a duplicate in the two processes.

In the above program, the parent process is executing it's block first and doesn't wait for the child. Most of the time, the parent process should wait and check on the child process and it's statuses. This is done using the function wait.

Why should I wait for my child process ?

There might be a situation when the child would terminate before the parent would call wait. In this situation, the parent process has no way to figure out what happened to the child process. In this situation, the child process would remain a zombie state and resources allocated to the child would remain active in the kernel and wont be released for future processes. The kernel keeps a copy of the state of the zombie child. When the parent calls wait , the kernel responds to the function using the state it has and then releases the resources and child that is terminated will not be in a zombie state.

So, we need to call wait function in the parent after a fork function. See the note section of the man pages:

"A child that terminates, but has not been waited for becomes a "zombie". The kernel maintains a minimal set of information about the zombie process (PID, termination status, resource usage information) in order to allow the parent to later perform a wait to obtain information about the child. As long as a zombie is not removed from the system via a wait, it will consume a slot in the kernel process table, and if this table fills, it will not be possible to create further processes. If a parent process terminates, then its "zombie" children (if any) are adopted by **init (https://linux.die.net/man/8/init) process**, which automatically performs a wait to remove the zombies"

What could we do inside the Child process after a fork ?

Remember, after fork, there are two separate processes running. We could add and execute any code inside these processes. In real life, server side programming involve the parent forks a child to serve a request from a client. The child process serves the request from a client and dies after it is done serving. For example, http request from your browser will be a request to the server. The server forks a child, the child serves the webpage to the browser and dies. The server goes on to the next request.

simple Algorithm of a web server :

```
do   {
request_received_from_browser ( ) ;
pid = fork ( )
   if ( pid == 0 ) {    //child process
     child serves the web page and dies
   }
  else  {  // Parent processes
```

```
        go back to serve more browser requests
    }

} while ( 1 ) ;
```

Disadvantages with FORK : Creating fork is a very expensive operation: the system has to copy the entire program, it has to create a new process, manage the child and parent relationships, save the status of all variables. If two processes have to communicate (what is known as interprocess communication) , an extensive programming effort has to be put for the two processes to share data or work collaboratively. Most of these efforts have issues with portability too and management of code across system is a nightmare for software developers.

Open File Descriptors:

When a fork() is performed, the child receives duplicates of all of the parent's file descriptors. These duplicates are made in the manner of dup(), which means that corresponding descriptors in the parent and the child refer to the same open file description. These attributes of an open file are shared between the parent and child. For example, if the child updates the file offset, this change is visible through the corresponding descriptor in the parent.

This is demonstrated in this example:

```
void  main(void)
{
     FILE *fp = fopen ( "data.txt", "w");

     pid_t   pid1,  pid;
     int     status;

     if ((pid1 = fork()) < 0) {
          printf("Failed to create child process 1\n");
          exit(1);
     }
     else if (pid1 == 0)  {
         sleep ( 4 );
         printf ( "  Child Process PID %d \n", getpid() );
         fprintf (fp, " Child created Process PID %d \n", getpid() );
         exit(0);
     }

     printf("Parent created  child PID1 %d \n", pid1);
     fprintf(fp, "Parent created  child PID1 %d \n", pid1);

     pid = wait(&status);
     printf("*** Parent sees child PID %d terminated ***\n", pid);
     fprintf(fp, "*** Parent sees child PID %d terminated ***\n", pid);
     exit(0);
}
```

FILE descriptor shared

```
[srivatss@athena:105]> gcc fileSharing.c
[srivatss@athena:106]> ./a.out
Parent created  child PID1 27546
  Child Process PID 27546
*** Parent sees child PID 27546 terminated ***
```

The system wide open descriptor table is demonstrated in this figure:

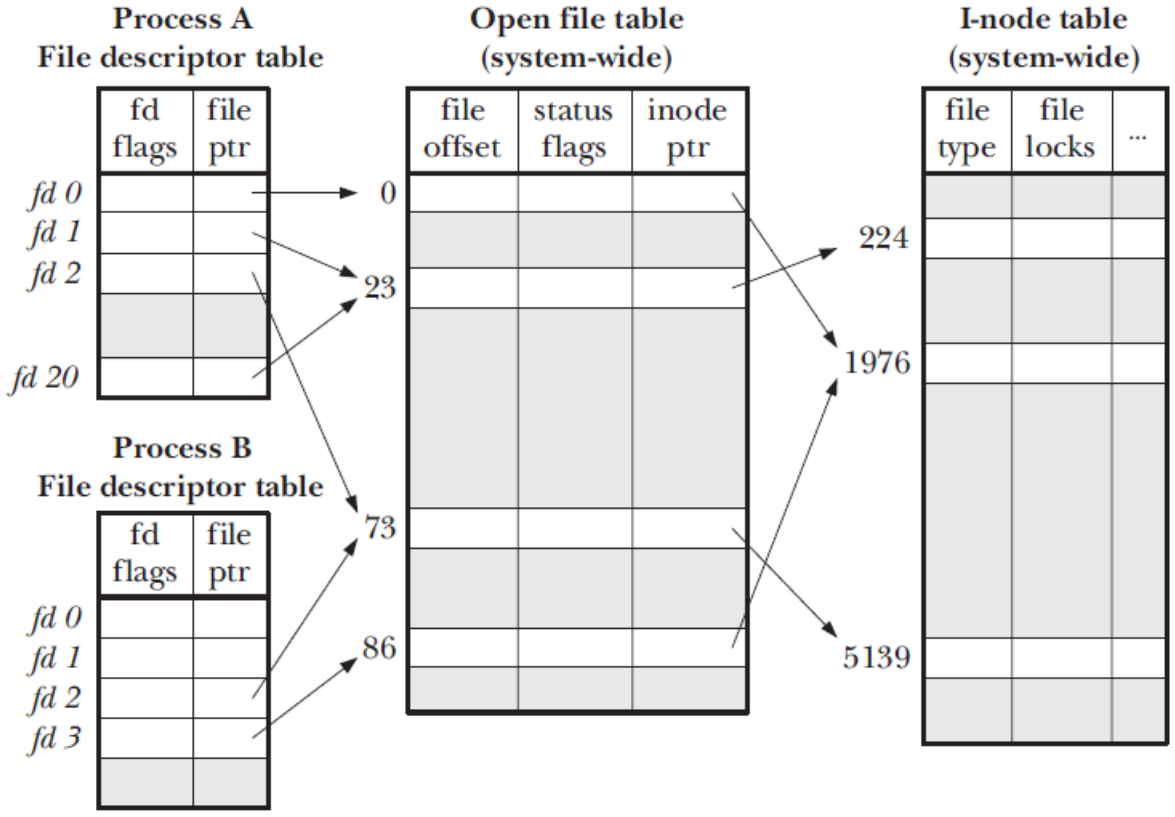
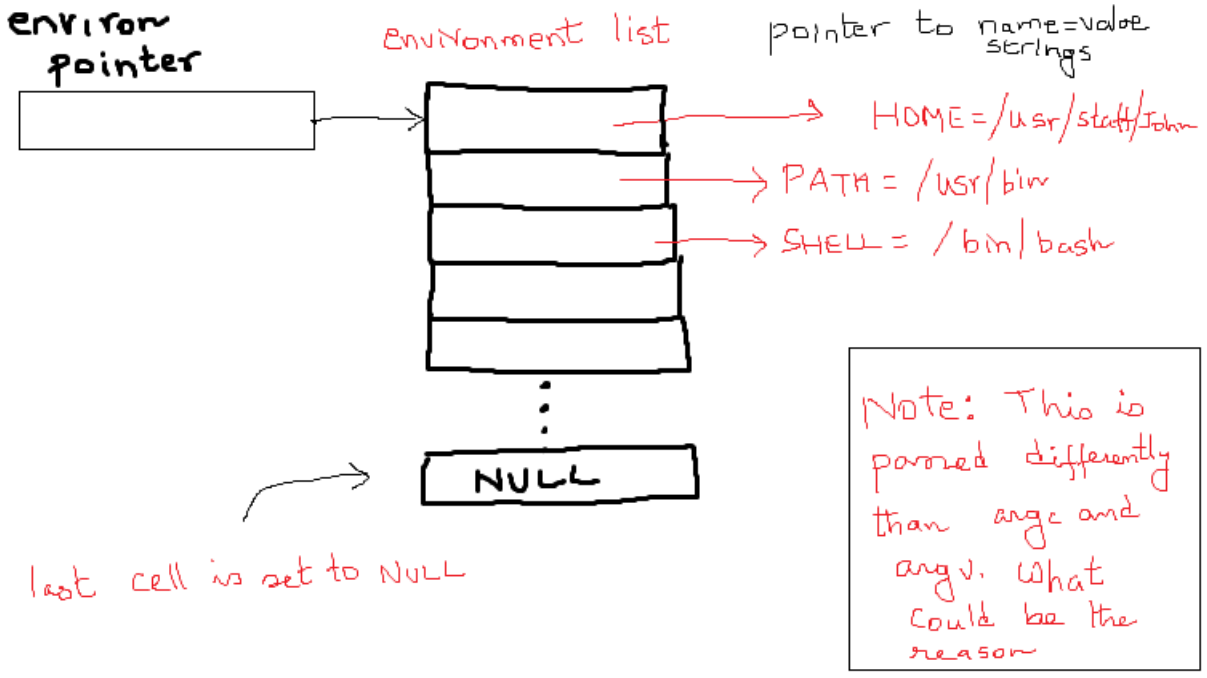


Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

environ variable



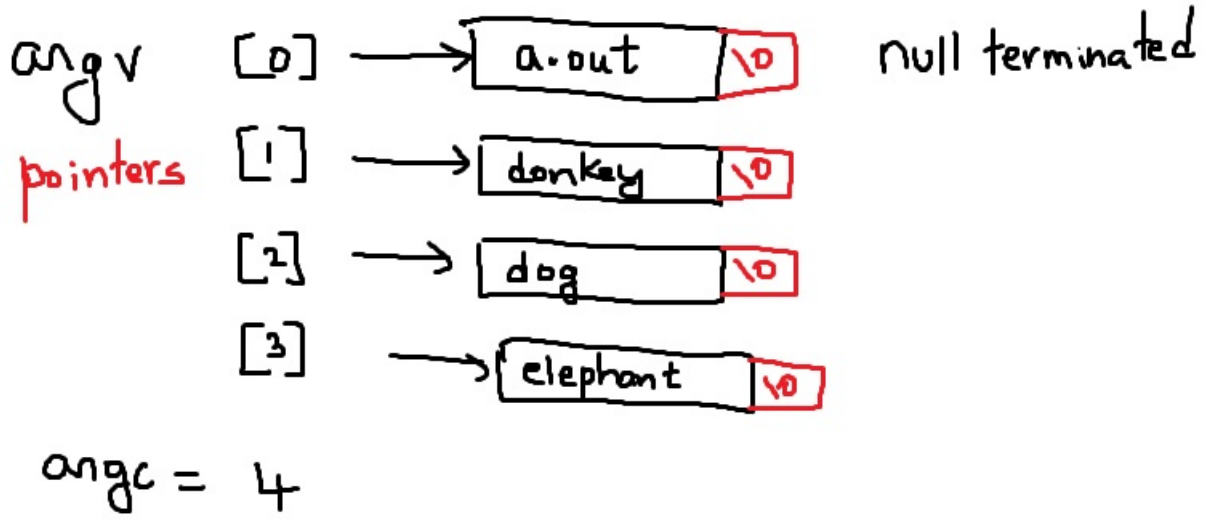
Sample program is:

```c
#include <stdio.h>
#include <stdlib.h>
extern char **environ; // pointer to pointer global variable
                       // copied to our memory address by the OS

int main( int argc, char *argv [ ] )
{

    int i = 0;

     for ( i = 0 ; environ[i] != NULL  ; i++ )
       printf ("value at %d is %s \n",  i, environ[i] );

}
```

environ global variable is checked for not NULL

passing arguments to C program (argc argv)

a.out donkey dog elephant



To print the arguments:

```
for ( i = 1 ; i < argc ; i++ )
  printf ( "  The argument are %s \n", argv[i]);
```

Wait functions

Why do we need wait function:

here is the video:

https://www.youtube.com/watch?v=Tt5qpp3141U (https://www.youtube.com/watch?v=Tt5qpp3141U)

(https://www.youtube.com/watch?v=Tt5qpp3141U)

Sample Program using wait function

Some documentation about wait function in FORK

int status ;

wait (&status) ;

waitpid (pid_t pid, &status, options);

wait (&status) is same as waitpid (-1 , &status, 0) ;

pid = -1 means do wait for any of the children

option = 0 means do not wait if any child has not exited.

The two system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose
state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state (see NOTES below)

Sample Code using wait ( ), please note the highlighted calls are macros. You can actually see the expansion of the macro using -E option. The sample code creates a child. The child sleeps for 60 seconds. Parent process waits to check the status of the child.

```
int main( void )
{
```

```
    int status;
    pid_t pid;

   pid = fork();


   if (pid == 0) {
       printf(" *** Child process is %d ***\n", getpid( ) );
      sleep ( 60) ;
  }  else      {
                wait ( &status);
                if ( WIFEXITED ( status) )
                     printf ( "Child exited normally %d \n", WEXITSTATUS (status) );
                else if ( WIFSIGNALED ( status) )
                    printf ( "Child exited by a Signal #%d \n", WTERMSIG (status) );
               }
}
```

1. Try to run this program in the background. wait for 60 seconds.

2. Again, try to run the program in the background. terminate the client process using the kill command. Kill is the command to send a signal to another process in the same group. There are various signals Linux supports. We will discuss them soon

Sample code using waitpid Function

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

void main( void )
{
       int status;
       pid_t pid;

     pid = fork();
     if (pid == 0) {
       printf("Child process is %d \n", getpid( ) );
       sleep ( 2 );
    }  else       {
       waitpid (pid,  &status, WUNTRACED | WCONTINUED );
       if ( WIFEXITED ( status) )
           printf ( "Child exited normally %d \n", WEXITSTATUS (status) );
       else if ( WIFSIGNALED ( status) )
           printf ( "Child exited by a Signal #%d \n", WTERMSIG (status) );
    }
}
```

vfork ( ) functon

```
#include  <stdio.h>
#include  <unistd.h>
#include  <stdlib.h>
void  main( void )
{
    int count = 10;
    srand ( time(NULL));
    pid_t pid = vfork();   //  try calling fork instead and check out the values of count printed

    if (pid == 0)  {
      count = rand( ) ;
      printf ( "child:count=%d \n" , count);
     _exit(0);
    }
    else
        {  // THIS IS PARENT
          printf ( "parent: count=%d \n" , count);
        }

}
```

Process Creation: System ( ) function call

system ("ls") ;

This will execute the command ls

system ("pwd"); This will execute the command pwd

How system works : it forks, executes exec functions in the child process. While the child executes the function, the parent waits.

Exec Family of Functions (video attached)

There is a system function called exec (6 of them). exec functions will execute any program you pass it to it. You also need to pass parameters if any.

When you launch the new program mentioned in the exec, the new program will replace the caller in memory, though the process ID remains the same. It is as if you launched a new program.

Here is the document :

exec-family-functions.docx (https://csus.instructure.com/courses/94454/files/14915326/download?wrap=1)

Here is the video :

linux operating system fork execl family of functions execl execlp execle (https://www.youtube.com/watch?v=duYojgFuT3s)

(https://www.youtube.com/watch?v=duYojgFuT3s)

EXEC FAMILY OF FUNCTIONS

A parent program can call a exec family of functions to launch a new program. The new program will replace the parent program, but not the process. The new program will get the process ID of the parent program. The segments such as text, initialized data, uninitialized data (bss), and stack of the calling process are overwritten according to the contents of the newly loaded program.

The parent program will not be able to check the status of the exec function on success. Only on error, the parent program can check and take remedial actions. We will discuss six functions, the difference between these functions is just in the format of the parameters passed. We also provided an example for each function.

execl - this function takes the command path, name and optional parameters and NULL parameters. execl ("/bin/ls", "ls", "-l", NULL)	execv – Very identical to execl, except the name and optional parameters and NULL parameters are passed separately in a array char *args [ ] = { "ls", "-l", NULL }; execv ("/bin/ls", args);
execle - In addition to execl, this also takes the environment variables in a NULL terminated string array.	execve - In addition to execv, this also takes the environment variables in a NULL terminated string array.
execlp – Same as execl , except it recognizes the path of the command using the $PATH variable. Absolute path is optional	execvp - Same as execv , except it recognizes the path of the command using the $PATH variable Absolute path is optional

The functions listed on the left side differ from the right side on one thing: function on the right side take optional parameters of the command in the array of strings. Whereas, you have to list them individually for functions on the left hand side, See the highlighted text in row 1. Functions that end in ‘e’ take the environment variables , and functions that end ‘p’ recognizes the $PATH variables for you to omit the absolute path of the command.

EXECL

```
1.  int execl       (   const char *file, const char *arg0, ..., NULL);
```

file : is the filename of the file that contains the executable image of the new process.
a**rg0, ..., NULL** : is a variable length list of arguments that are passed to the new process image. Each argument is specified as a null-terminated string, and the list must end with a NULL pointer. The first argument, arg0, is required and must contain the name of the executable file for the new process image. If the new process image is a normal SAS/C main program, the list of arguments will be passed to argv as a pointer to an array of strings.

Example 1:

```
main  ( )

{
  execl ( "/usr/bin/cal", "cal", "2017", NULL );
}

another example, echo prints SYSTEM PROGRAMMING

execl ( "/bin/echo", "echo", "SYSTEM PROGRAMMING", NULL );
```

EXECLE

```
2. int execle  ( const char *file,
                 const char *arg0, ..., NULL,
                 char *const envp [ ]  ) ;
```

file : is the filename of the program that is to be launched.

arg0, ..., NULL : is a variable length list of arguments that are passed to the new program. Each argument is specified as a null-terminated string, and the list must end with a NULL pointer. The first argument, arg0, is required and must contain the name of the executable file for the new process image. The number of strings in the array is passed to the main() function as argc.

envp is a pointer to an array of pointers to null-terminated character strings. A NULL pointer is used to mark the end of the array. Each character string pointed to by the array is used to pass an environment variable to the new process image. Each string should have the following form:

" *var* = *value* "

Example 1:

```
main  ( )
{
  char *env[ ] =
      { "SYSTEM PROGRAMMING", NULL };

   execle ( "/bin/echo", "echo", env[0], NULL, env );
}

another example
execle ( "/bin/echo", "echo", environ[3], NULL, environ );
```

EXECLP

```
3. int execlp(const char *path, const char *arg0, ..., NULL);
```

path : identifies the location of the new process in the system. If the path argument contains a slash (/), it is assumed that either an absolute or a relative pathname has been specified. If the path argument does not contain a slash, the directories specified by the PATH environment variable are searched in an attempt to locate the file.

arg0, ..., NULL : is a variable length list of arguments that are passed to the function. Each argument is specified as a null-terminated string, and the list must end with a NULL pointer. The first argument, arg0, is required and must contain the name of the executable file.

Example 3:

```
main  ( )
{
  execlp ( "ls", "ls", "-lF", NULL );
}

another example
execlp ( "./echo_1.sh", "./echo_1.sh", "Jack", "Sam", "Pam", NULL );
```

EXECV

```
4. int execv(const char *file, char *const argv[] ) ;
```

file : is the filename of the file that contains the executable image of the new process.
argv : is a pointer to an array of pointers to null-terminated character strings. A NULL pointer is used to mark the end of the array. Each character string pointed to by the array is used to pass an argument to the new process image. The first argument, argv[0], is required

Example:

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

EXECVE

5. **int execve (const char **filename*, char *const *argv [ ] ,**
char *const *envp [ ]) ;

filename : is the filename of the program to be launched by the function
argv : is a pointer to an array of pointers to null-terminated character strings. A NULL pointer is used to mark the end of the array. Each character string pointed to by the array is used to pass an argument to the new process image. The first argument, argv[0], is required and must contain the name of the executable file for the new process image.
envp : is a pointer to an array of pointers to null-terminated character strings. A NULL pointer is used to mark the end of the array. Each character string pointed to by the array is used to pass an environment variable to the new process image.

Example :

```
main  ( )
{
 char *paramList[ ] = { "echo", environ[4], NULL };
 execve ( "/bin/echo", paramList, environ );
}
```

EXECVP

```
6. int execvp ( const char *path, char *const argv [ ] ) ;
```

path: identifies the location of the new program. If the path argument contains a slash (/), it is assumed that either an absolute or a relative pathname has been specified. If the path argument does not contain a slash, the directories specified by the PATH environment variable are searched in an attempt to locate the file.
argv : is a pointer to an array of pointers to null-terminated character strings. A NULL pointer is used to mark the end of the array. Each character string pointed to by the array is used to pass an argument to the new process image. The first argument, argv[0], is required and must contain the name of the executable file for the new process image.

Example:

```
main  ( )
{
 char *paramList[ ] =  { "ls", "-l", NULL} ;
  execvp ( "ls", paramList );
}
```

Sample code for fork - versions

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

void  main( void )
{

     pid_t  pid;

     pid = fork();
     if (pid == 0)  {
          printf("   *** Child process is %d ***\n", getpid(  )  );
     }
     else
          {
             printf("   *** Parent process is %d ***\n", getpid(  )  );
          }
}
```

Version 2 with wait function. In the previous example, the parent prints first and doesn't wait for the child process. In this second version, the parent waits and checks the various status of the child

```
void  main( void )
{
     int status;
     pid_t  pid;

     pid = fork();
     if (pid == 0)  {
          printf("   *** Child process is %d ***\n", getpid(  )  );
          sleep ( 60);
     }
     else
          {
            wait ( &status);
            if ( WIFEXITED ( status)  )
                printf ( "Child exited normally %d \n", WEXITSTATUS (status) );
            else if ( WIFSIGNALED ( status)  )
               printf ( "Child exited by a Signal #%d \n", WTERMSIG (status) );
          }
}
```

BRIEF INTRODUCTION TO INTERPROCESS COMMUNICATIONS

We will read about inter-process communication between processes. There are many mechanisms that are available for a system programmer to use. The mechanisms range from data transfer from one process to another (via pipes, FIFOs, message queues, shared memory), signaling a process to handle or ignore an event, and lastly synchronization (using semaphores, mutexes, condition variables). Pipes are simple, but are less desirable because they are half-duplex (you can read or write) compared to shared memory. Shared memory is fast because a process can create a shared memory for all processes involved to read or write.

PIPES

We use a pipe to allow communication between two processes. A pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child process. A pipe is half duplex, i.e. data flows in only one direction from parent process to child process or child to parent process.

Every time you type a sequence of commands in a pipeline for the shell to execute, the shell creates a separate process for each command and links the standard output of one to the standard input of the next using a pipe. For example, consider this shell command

```
ls –l | wc –l
```

would create two processes : one for the ls –l and the other for wc –l . A pipe for the output of the first process is created and the data is piped to the second. The pipe in the second is created for reading the input data. When both are done completing the data transfer, the pipes are destroyed and then the processes.

General characteristics of pipe are:

A pipe is a byte stream - there is no concept of messages or message boundaries when using a pipe. The process reading from a pipe can read blocks of data of any size, regardless of the size of blocks written by the writing process. Furthermore, the data passes through the pipe sequentially— bytes are read from a pipe in exactly the order they were written. It is not possible to randomly access the data

Reading from a pipe - Attempts to read from a pipe that is currently empty block until at least one byte has been written to the pipe. If the write end of a pipe is closed, then a process reading from the pipe will see end-of-file (i.e., read ( ) returns 0) once it has read all remaining data in the pipe.

Half-Duplex- Pipes are unidirectional Data can travel only in one direction through a pipe. One end of the pipe is used for writing, and the other end is used for reading.

Pipes have a limited capacity - Amount of Data can be written - PIPE_BUF is a constant that varies across UNIX implementations; for example, it is 512 bytes on FreeBSD 6.0, 4096 bytes on Tru64 5.1, and 5120 bytes on Solaris 8. On Linux, PIPE_BUF has the value 4096. When a process is writing data to a pipe, another process should be reading it because there is a limit of

size. If the size is exceeded, the process that is writing is blocked. When there is no data on the pipe, if a process is reading it, it will get end of file.

Creating and Using Pipes programmatically :

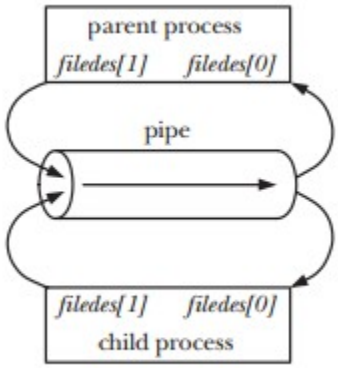
The pipe ( ) system call creates a new pipe

```
#include <unistd.h>
int   pipefd [ 2 ] ;  // step1 : create an array of two cells
int  pipe (  pipefd  )  ; // step2 :  pass the array to the pipe function , which returns
```

A successful call to pipe ( ) returns two open file descriptors in the array pipefd : one for the read end of the pipe (pipefd [0]) and one for the write end (pipefd [1]) .

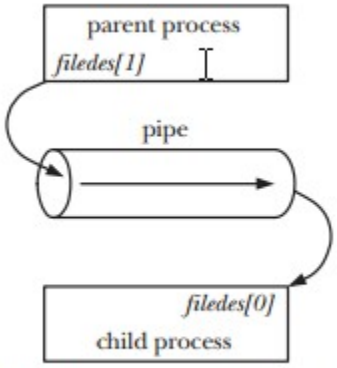
As with any file descriptor, we can use the read ( ) and write ( ) system calls to perform I/O on the pipe. Once written to the write end of a pipe, data is immediately available to be read from the read end.

To connect two processes using a pipe, we follow the pipe() call with a call to fork(). During a fork(), the child process inherits copies of its parent's file descriptors . See the figure .



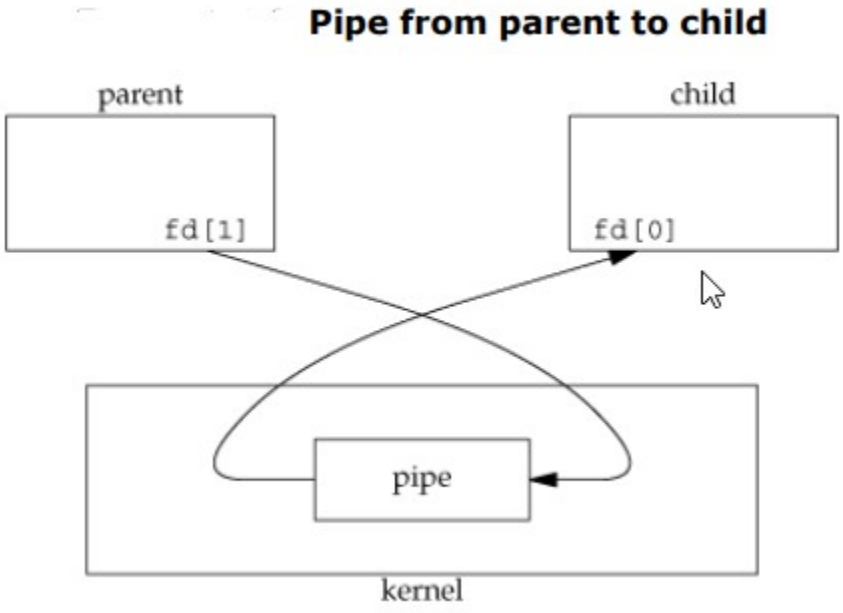
a) After *fork()*

To start communication between two processes (say parent and child), immediately after the fork ( ) by a parent, one process closes its descriptor for the write end of the pipe, and the other closes its descriptor for the read end. For example, if the parent is to send data to the child, then it would close its read descriptor for the pipe, fd [0], while the child would close its write descriptor for the pipe, fd [1] . Otherwise, if both parent and child write into the pipe or read from it, we may not know which process wrote into the pipe or read from the pipe, causing race conditions.



b) After closing unused descriptors

Pipes are generally done in kernel space, so reading or writing involves system call. So we have this diagram



Here is the sample program

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

#define MAX_LINE  32
int main(int argc, char *argv[])
{
    int pipefd[2];
    pid_t cpid;
    char buf[ MAX_LINE ];

    if (pipe(pipefd) == -1) {
       perror("pipe");
       exit(EXIT_FAILURE);
    }
    cpid = fork();
    if (cpid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
     }
    if (cpid == 0) {    /* Child reads from pipe */
      close(pipefd[1]);          /* Close unused write end */
      int num = read(pipefd[0], &buf, MAX_LINE) ;
      printf ( " Reading in child process\n");
      printf( "%s \n", buf );
```

```
    close(pipefd[0]);
  } else {          /* Parent writes argv[1] to pipe */
          close(pipefd[0]);         /* Close unused read end */
          write(pipefd[1], "Hello World", 12);
          close(pipefd[1]);         /* Reader will see EOF */
          wait(NULL);               /* Wait for child */
       }
  exit( EXIT_SUCCESS);
}
```

In the previous example, we called read and write directly on the pipe descriptors. The parent process closes the file descriptor pipefd [0] , and then writes “hello world” into the file descriptor pipefd [1]. The child process closes the file descriptor pipefd [1] and reads the data from pipefd [0] and then writes the data to the standard out .

dup and dup2 functions

What is dup ( ) ?

int fd2 = dup (fd1)

The dup() system call creates a copy of the file descriptor old fd1, using the lowest-numbered unused file descriptor for the new descriptor fd2.

After a successful return, the old and new file descriptors may be used interchangeably.

They refer to the same open file description and thus share file offset and file status flags; for example, if the file offset is modified by using lseek on one of the file descriptors, the offset is also changed for the other.

int fd2 ;

dup2 (fd2, fd1);

The dup2() system call performs the same task as dup(), but instead of using the lowest-numbered unused file descriptor, it uses the file descriptor number specified in newfd. If the file descriptor newfd was previously open, it is silently closed before being reused.

Note the following points:

- 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.

Here is the sample code for dup ( ) . I have given two versions. Compile them individually and check them out.

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

int ver1()
{
  int ofd, nfd;
  char buffer[13] = "Hello World\n";

  write(STDOUT_FILENO,buffer,strlen  ( buffer)); // write on the stdout

  nfd=dup(STDOUT_FILENO);   // duplicate
  write(nfd,buffer,strlen  ( buffer)); // write on the stdout
```

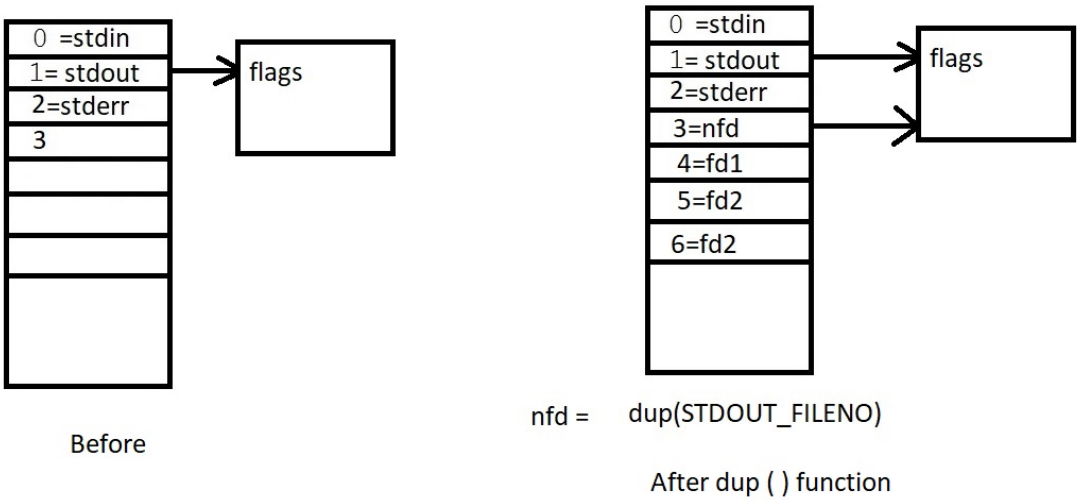
```
}
```

```
//ver2:
#include<unistd.h>
#include<stdio.h>
#include<fcntl.h>

int ver2()
{
   int fd1, fd2;
   char buffer[13] ;
   char ch='\n';

   fd1  =  open("alpha.txt",O_RDWR|O_CREAT);
   read(fd1,buffer,10); // read from the file
   write(STDOUT_FILENO,buffer,10); // write on the stdout
   write(STDOUT_FILENO,&ch,1); // write end of line

   fd2  =  dup(fd1);   // duplicate
   read(fd2,buffer,10); // read another 10 characters from the file
   write(STDOUT_FILENO,buffer,10); // write on the stdout
   write(STDOUT_FILENO,&ch,1); // write end of line
   close( fd1);
}
```

Here is the dup2 function:

```
#include<unistd.h>
#include<stdio.h>
#include<fcntl.h>
int main()
{

   int ofd, nfd;
   ofd = open("number.txt", O_RDWR | O_CREAT);

   printf ( "Hello World 1\n");

   dup2(ofd, STDOUT_FILENO);
   close(ofd);

   printf ( "Hello World 2\n");  // should write to the file , not display
}
```

Here is the before and after screenshot of the file descriptor of dup ( )



here is the video:

https://www.youtube.com/watch?v=mUX7TB1ZSX8 (https://www.youtube.com/watch?v=mUX7TB1ZSX8)

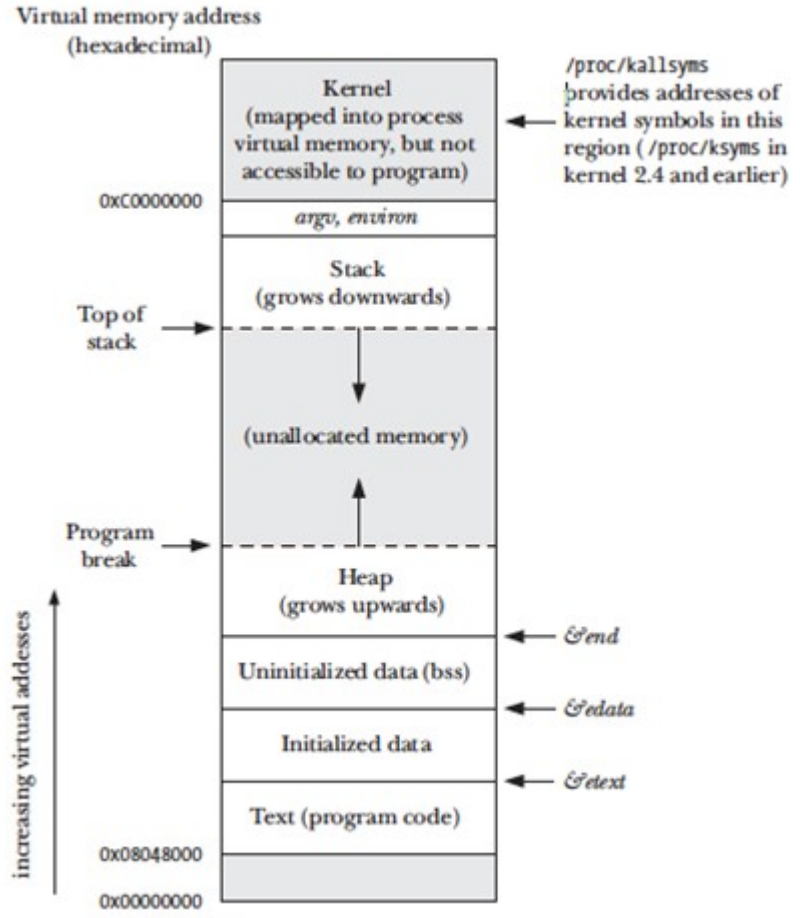
(https://www.youtube.com/watch?v=mUX7TB1ZSX8)

How to chain bash commands

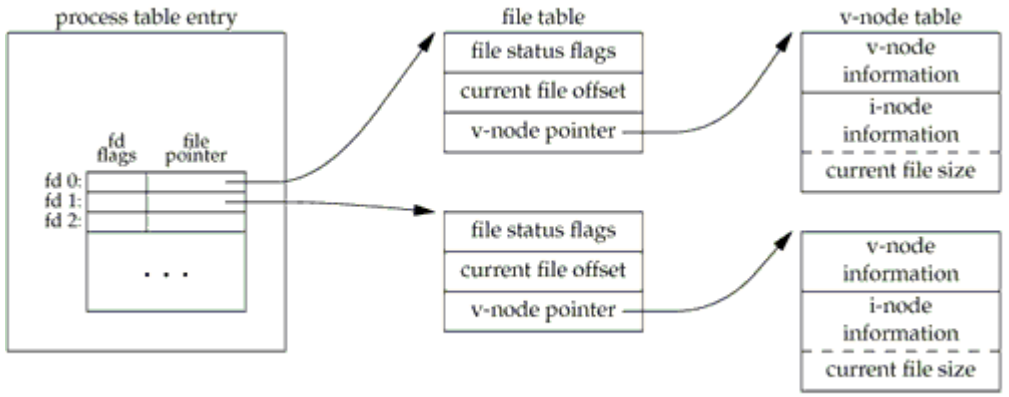
You have to review

- file descriptor table
- dup
- dup2
- pipe
- fork (both memory and page descriptor tables will be duplicated)
- execl

Memory layout of a Process



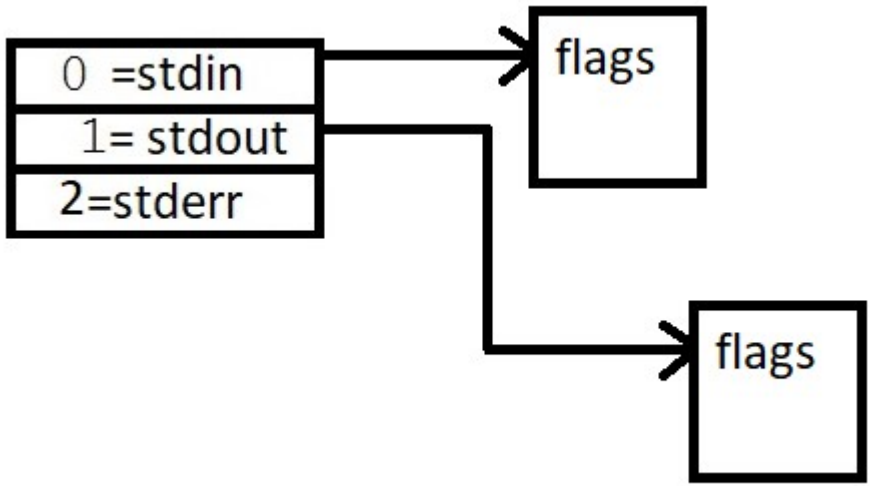
file descriptor



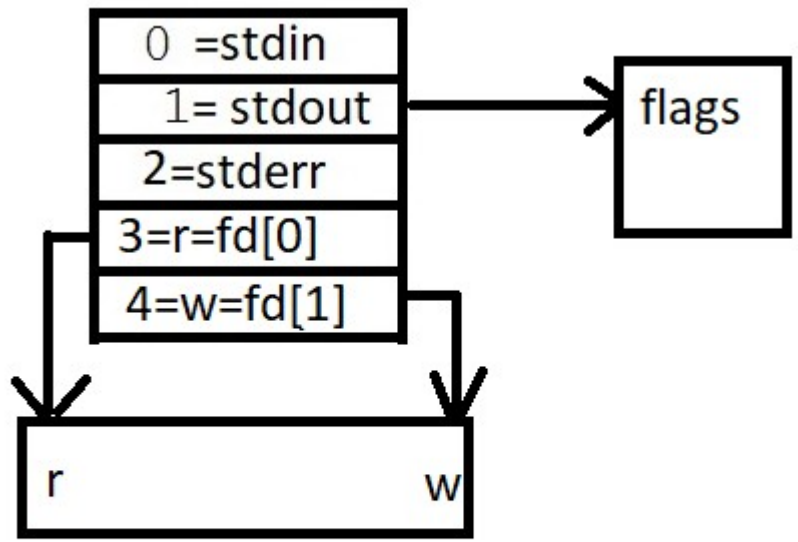
Standard Input output error streams

- STDIN_FILENO
- STDOUT_FILENO
- STDERR_FILENO

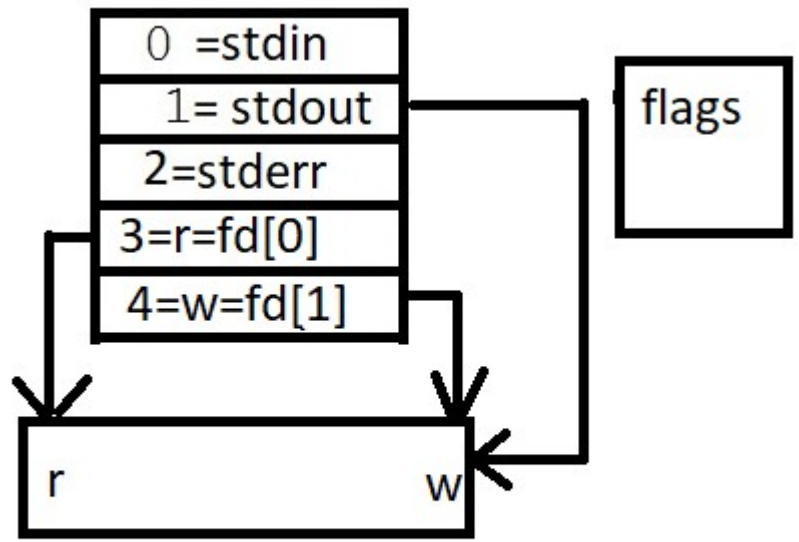
When you launch a program, the page descriptor table looks like this



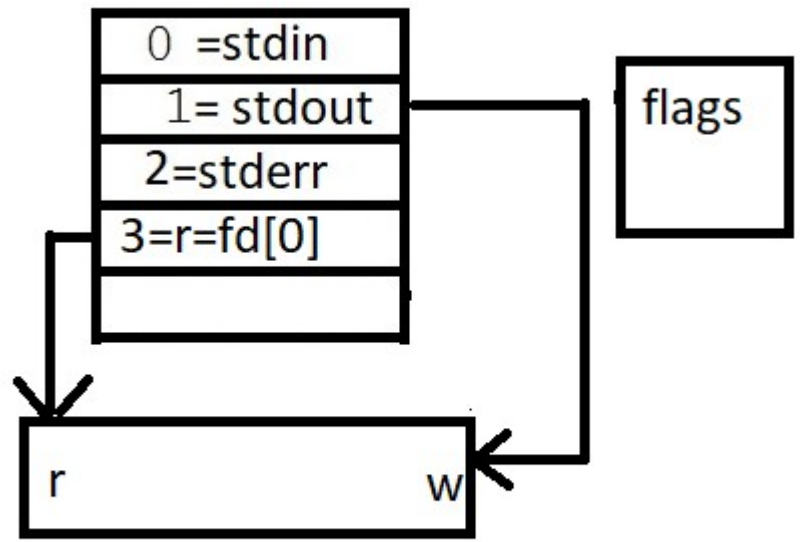
Let us see the example : ls | wc -l



int fd[2]
pipe(fd)

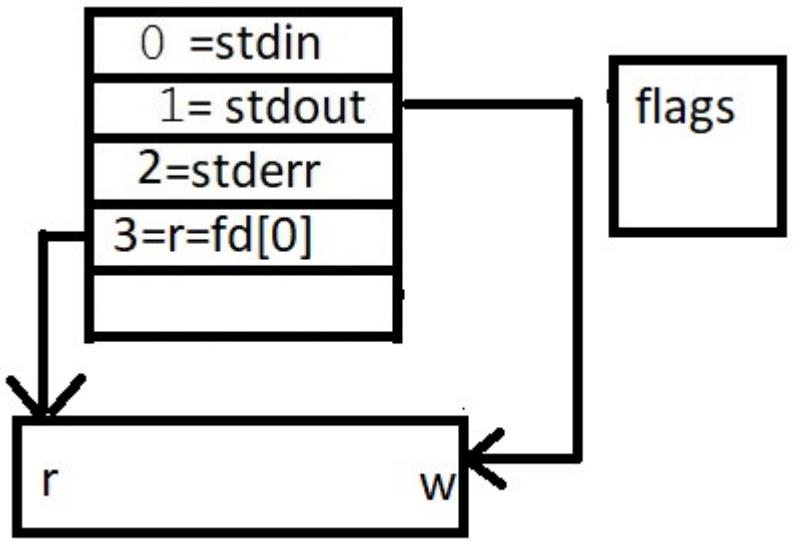


dup2(fd[1], 1)

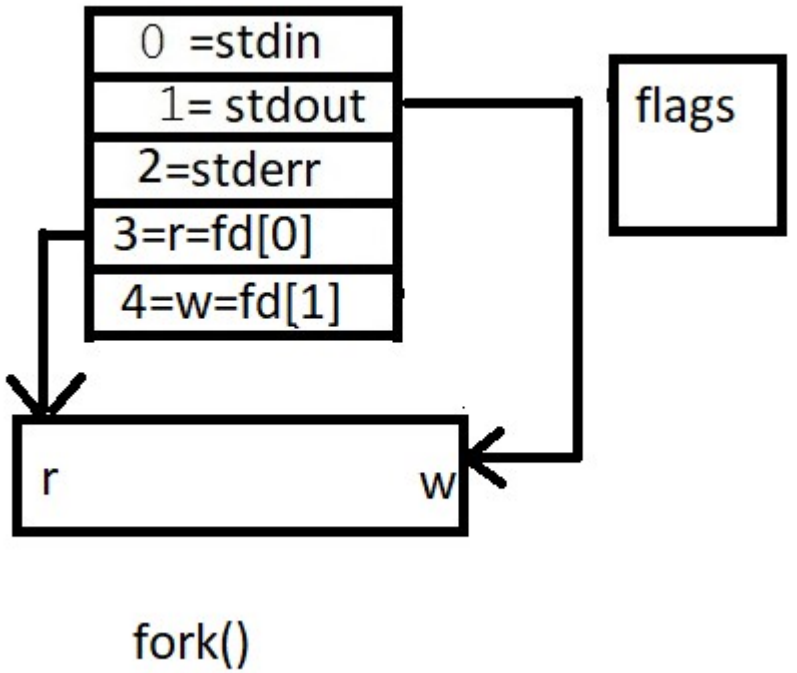


close(fd[1)

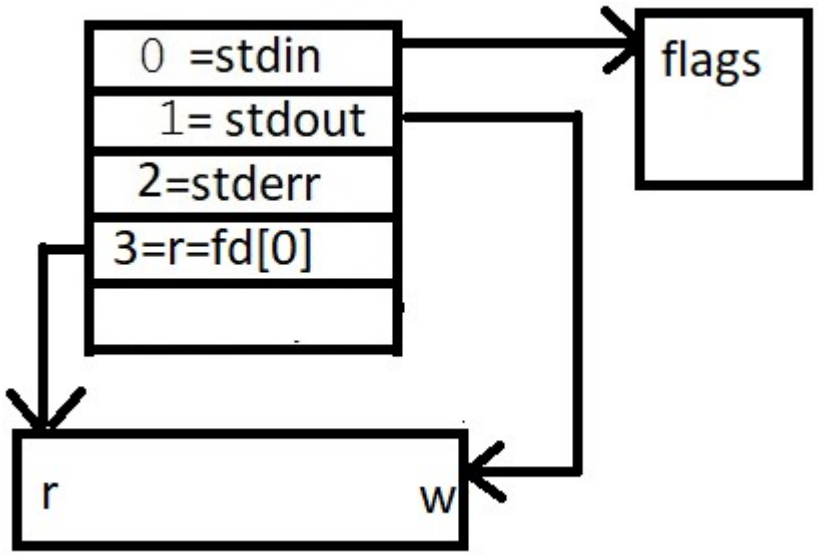
restoreFd = fd[0]

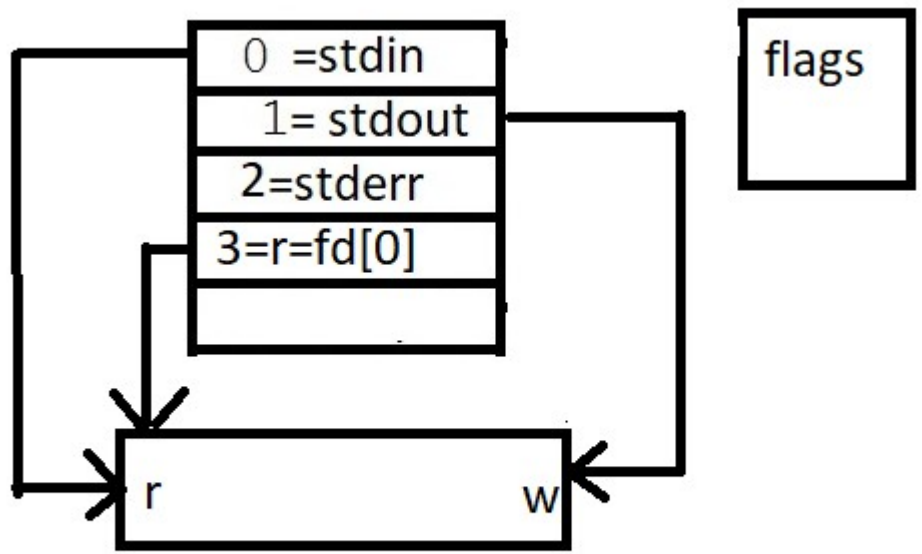


child will execute (say ls printing in the pipe)

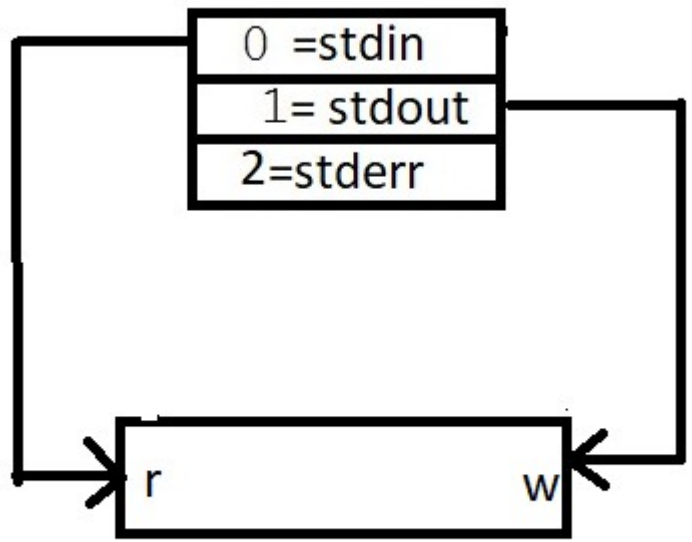


back to parent

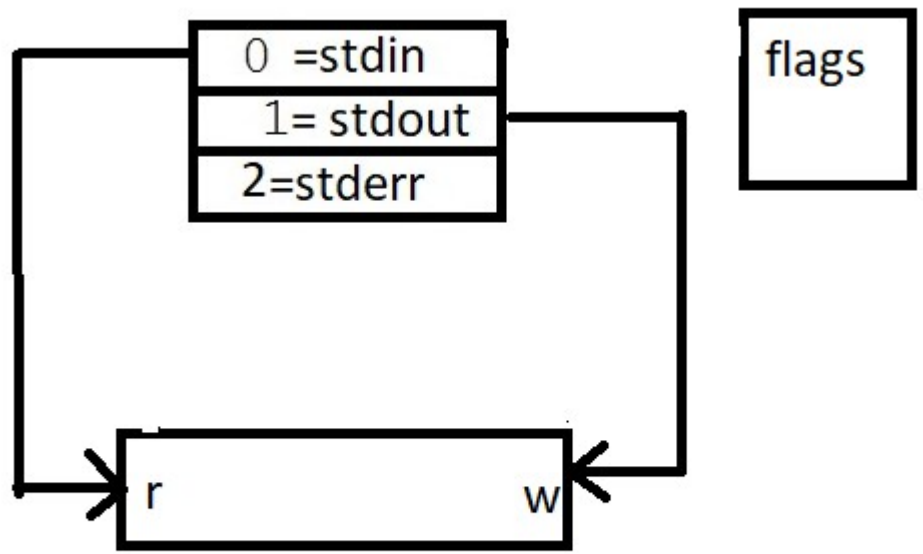




dup2(restoreFd, 0)

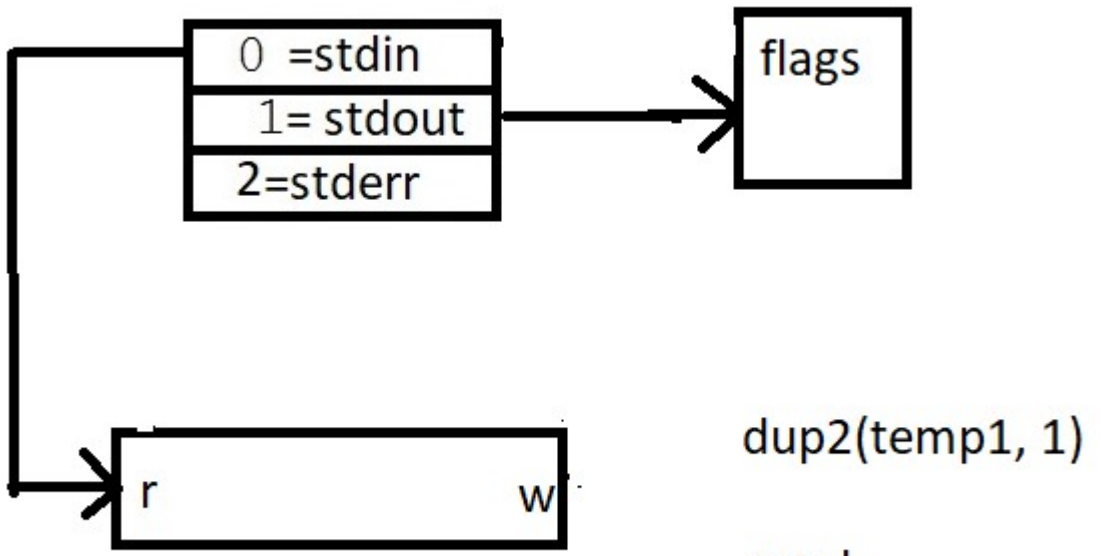


close(restoreFd)



fork

Will execute WC



dup2(temp1, 1)

execl

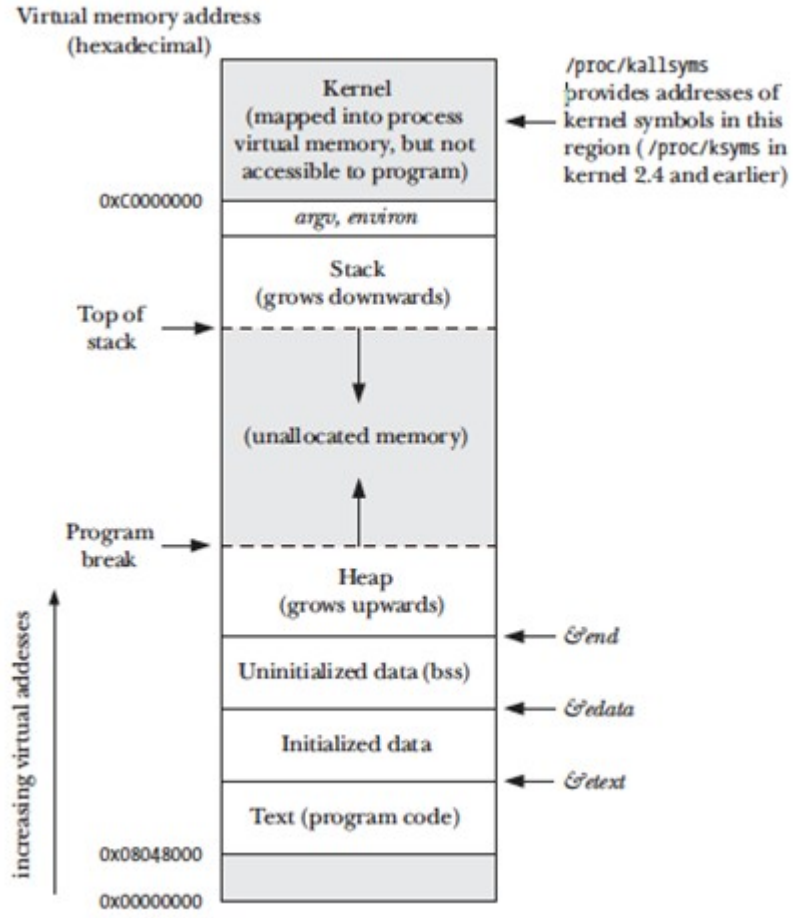
How to chain bash commands

ls –l | grep sankar | wc -l

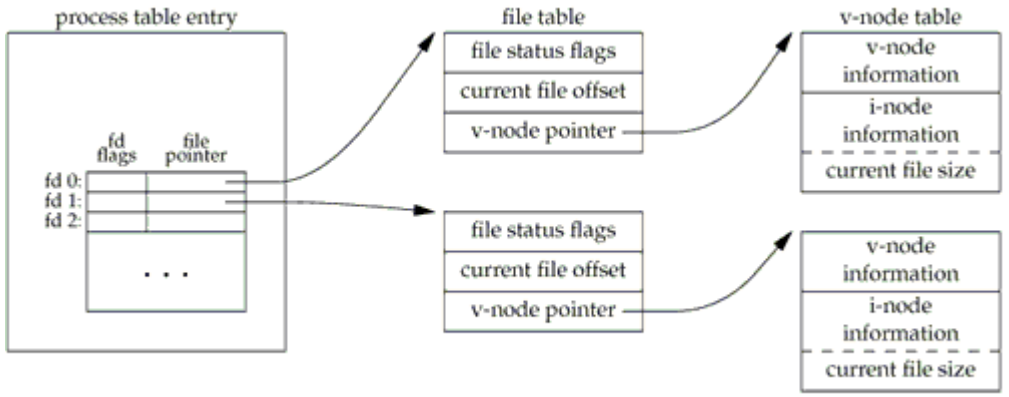
You have to review

- file descriptor table
- dup
- dup2
- pipe
- fork (both memory and page descriptor tables will be duplicated)
- execl

Memory layout of a Process



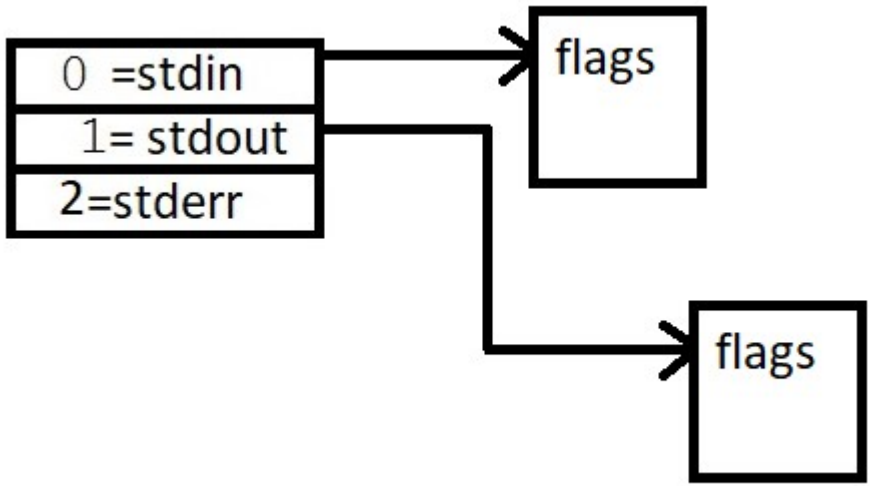
file descriptor



Standard Input output error streams

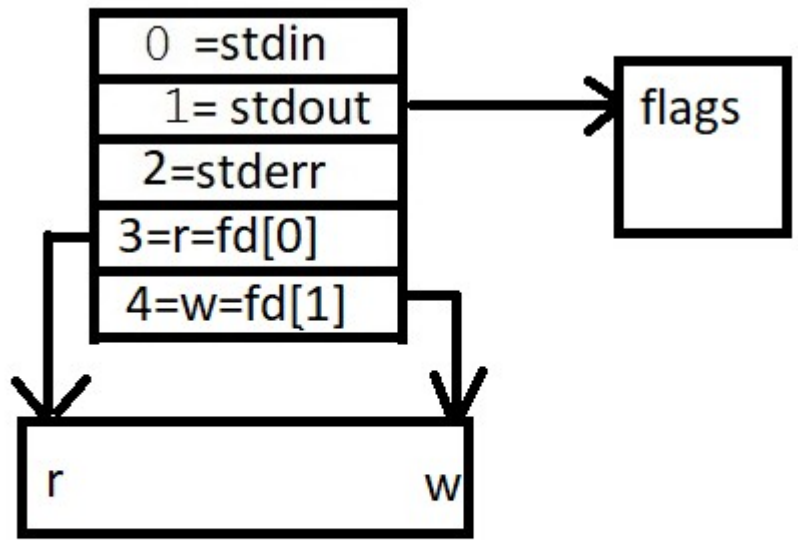
- STDIN_FILENO
- STDOUT_FILENO
- STDERR_FILENO

When you launch a program, the page descriptor table looks like this

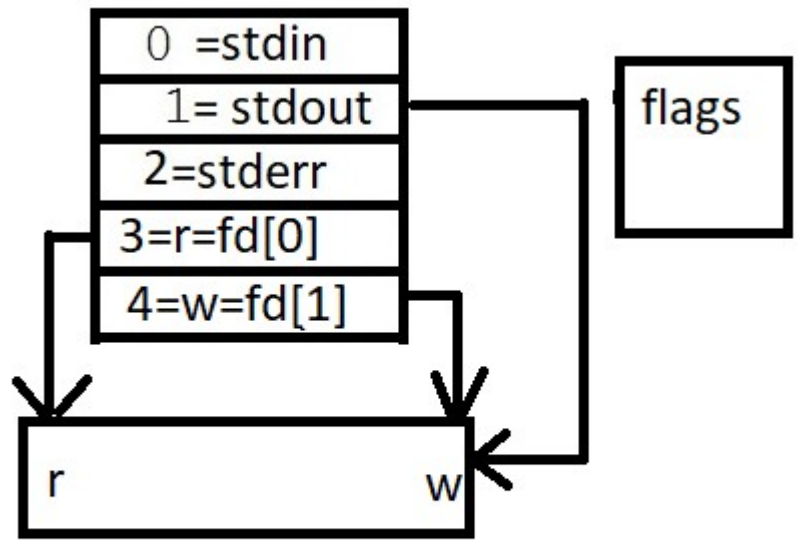


Another example: ls | grep sankar | wc -l

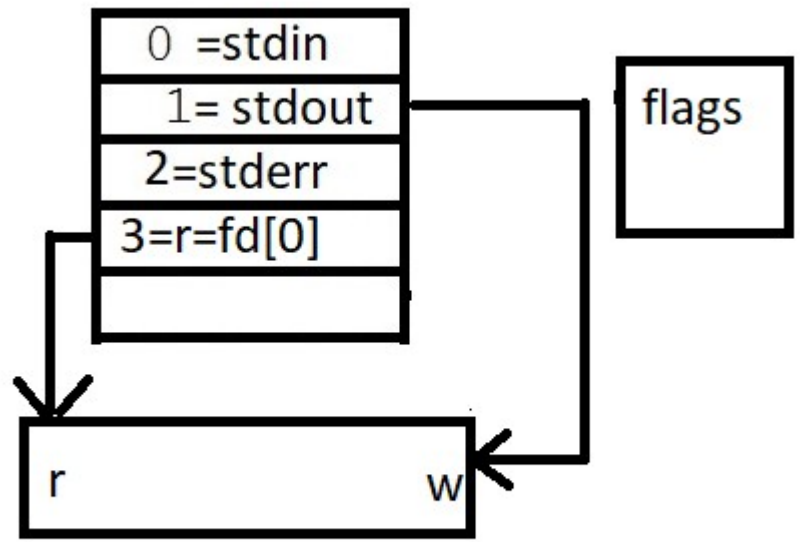
- Three or more pipes



int fd[2]
pipe(fd)

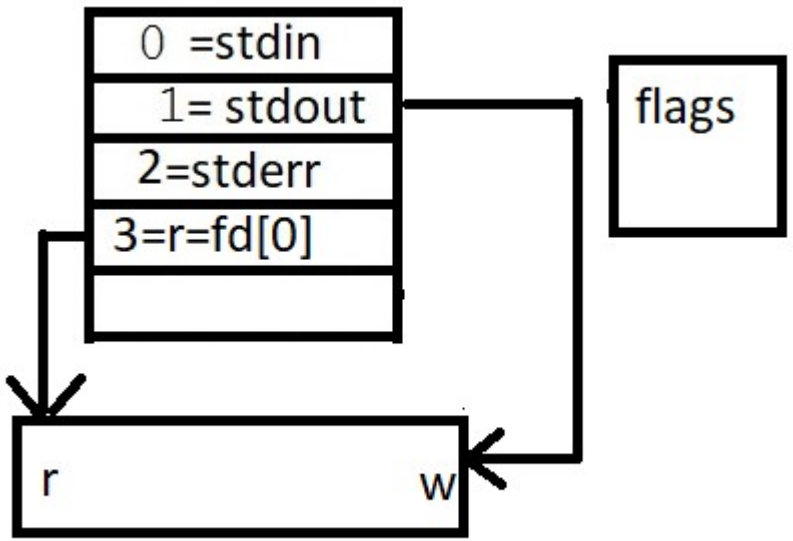


dup2(fd[1], 1)

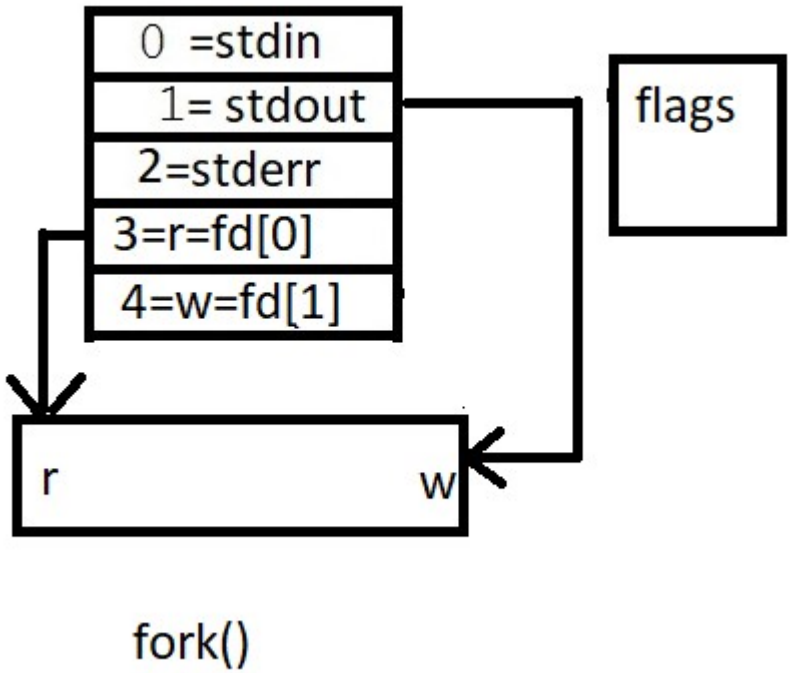


close(fd[1)

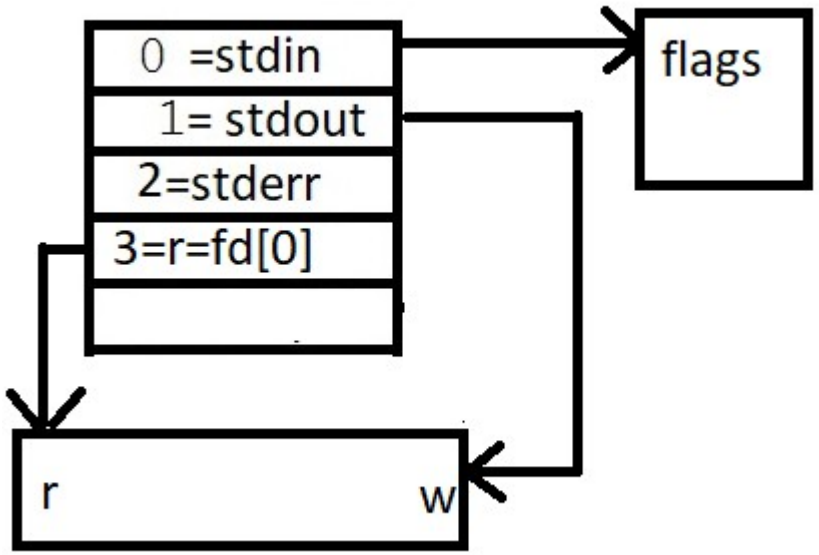
restoreFd = fd[0]



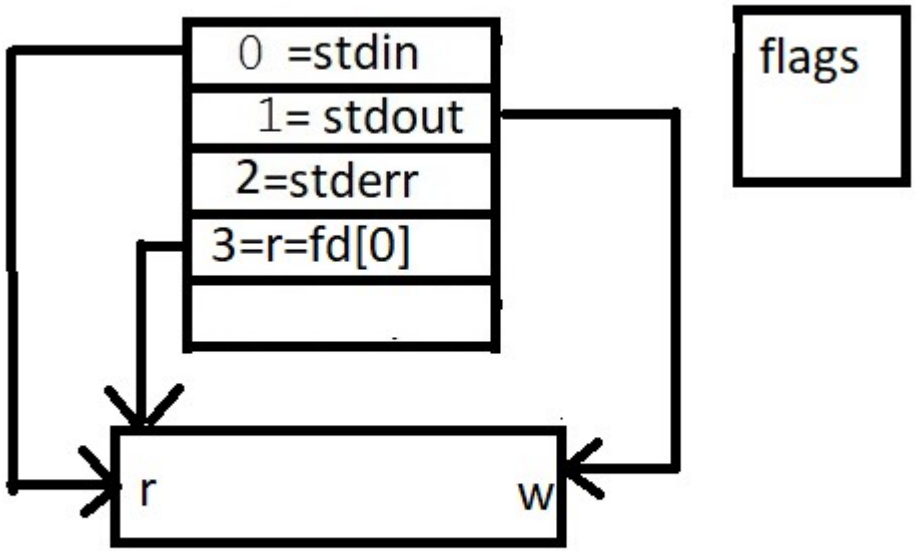
child will execute (say ls printing in the pipe)



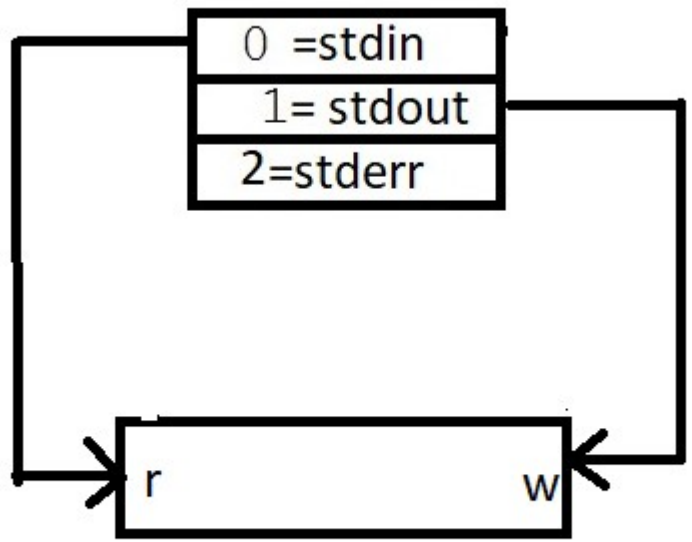
back to parent



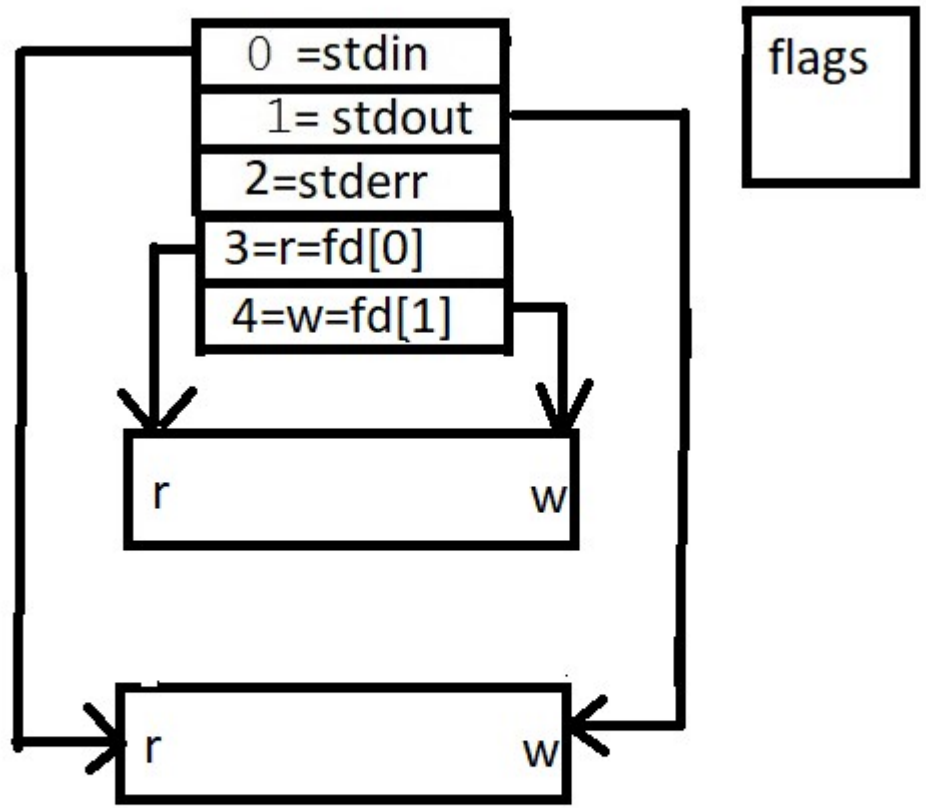
Will setup to execute the grep command



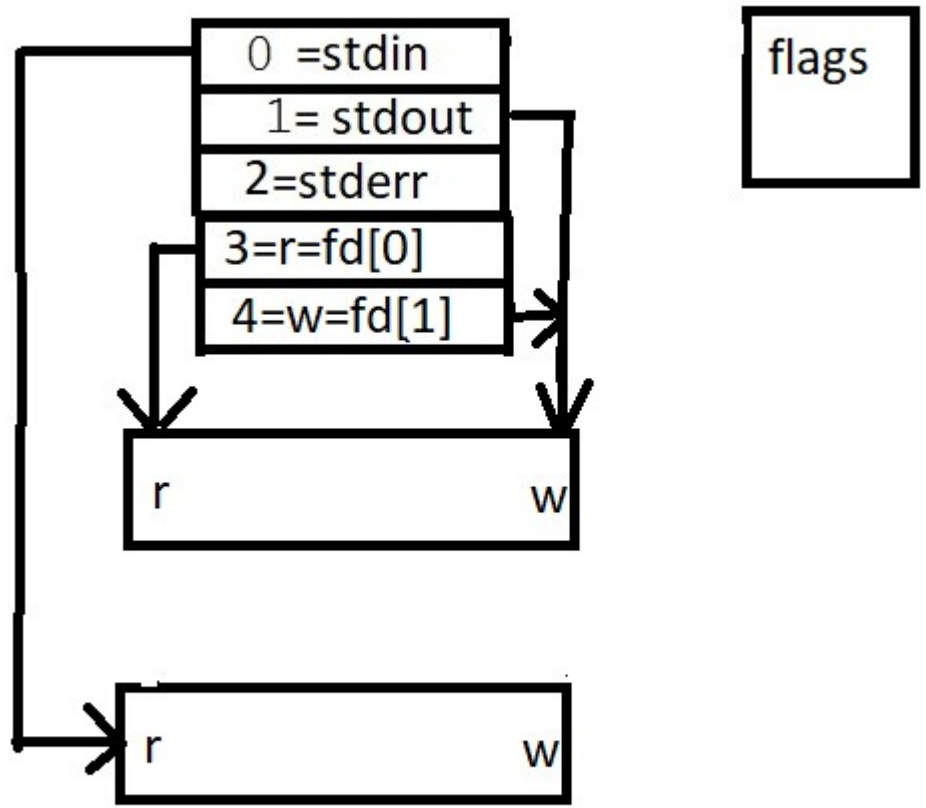
dup2(restoreFd, 0)



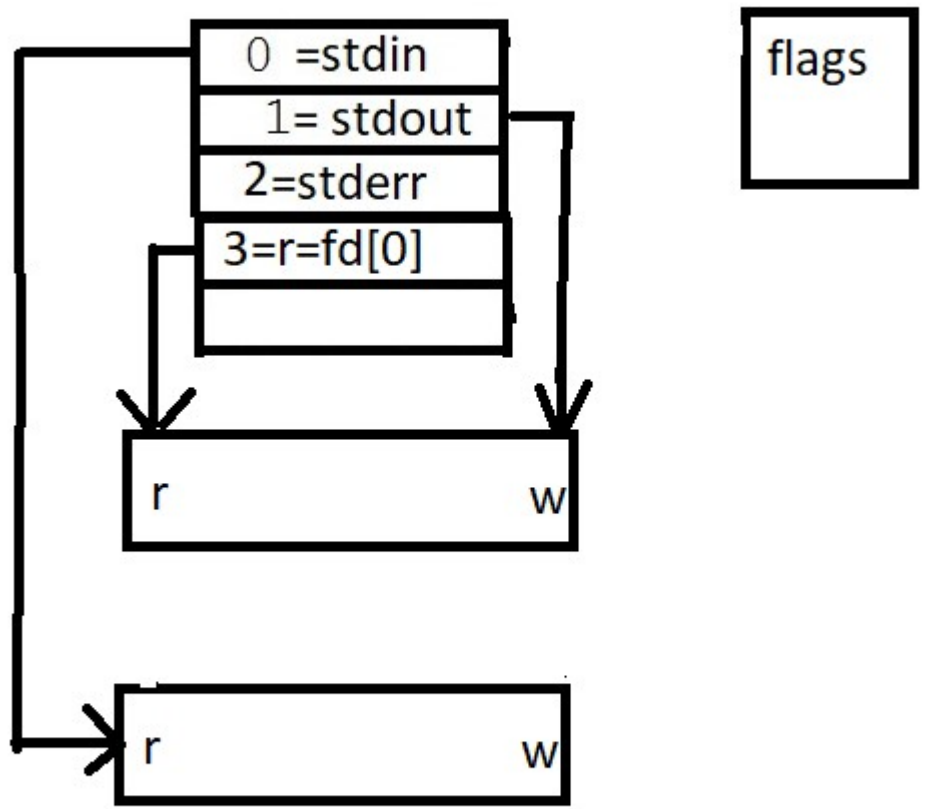
close(restoreFd)



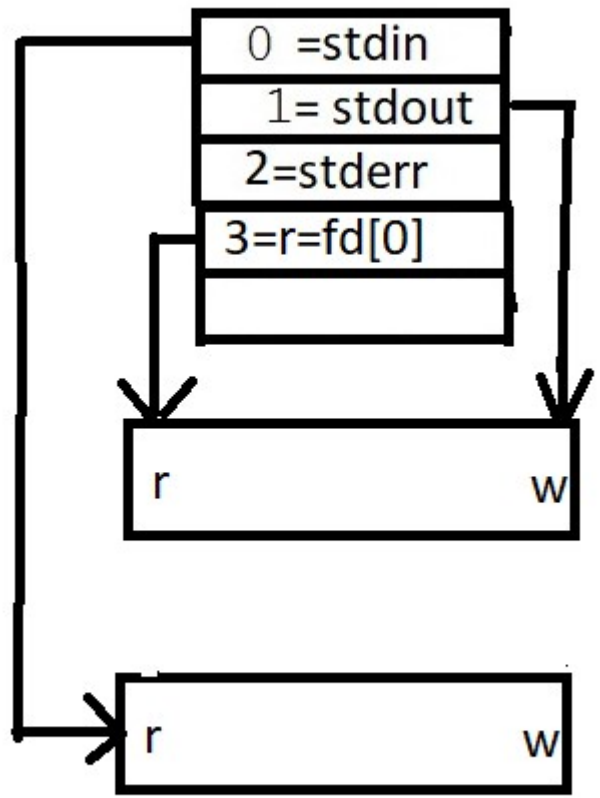
int fd[2]
pipe(fd)



dup2(fd[1], 1);

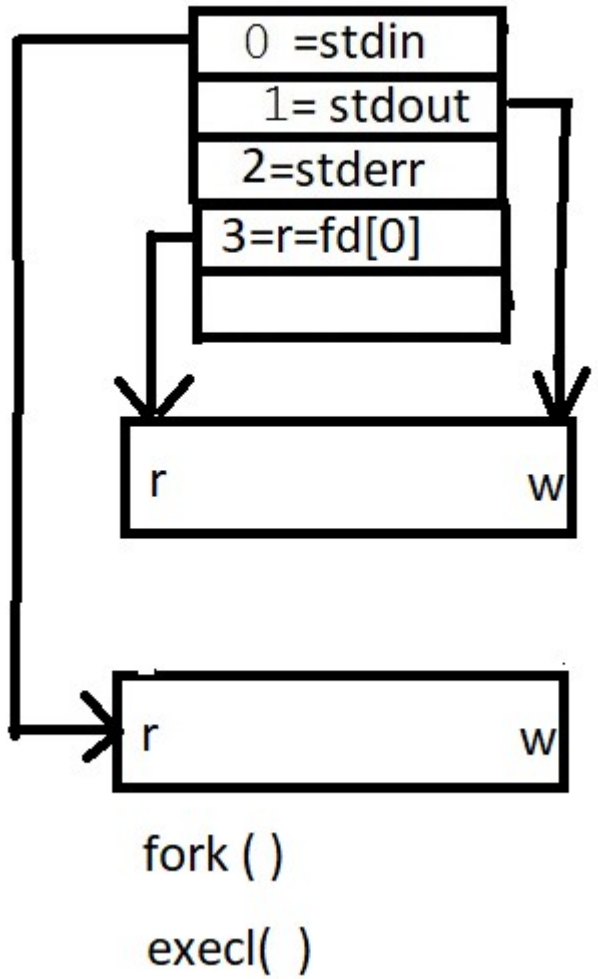


close(fd[1])

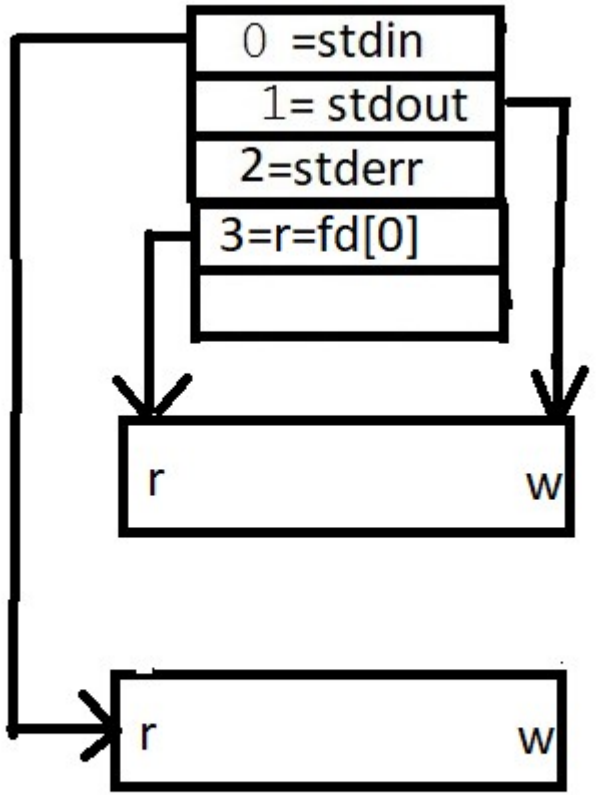


restoreFd(fd[0])

child will execute the grep command

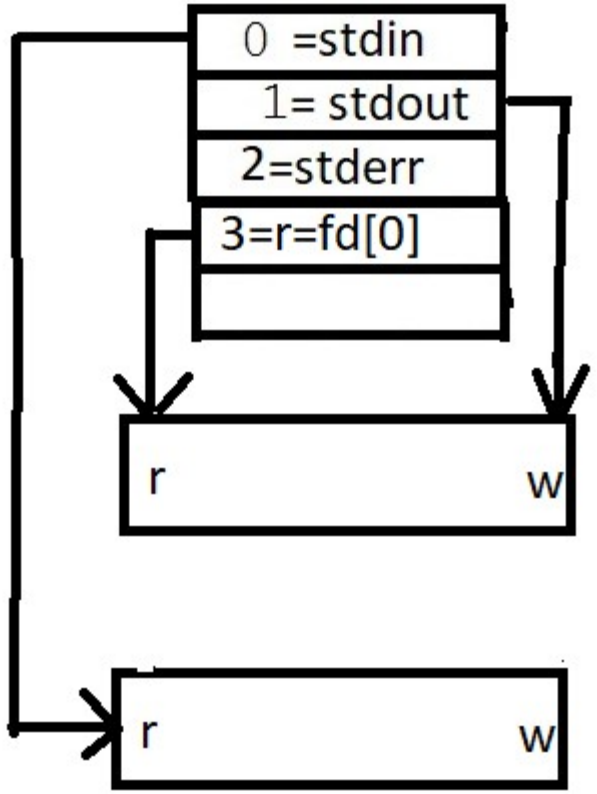


Back to the parent

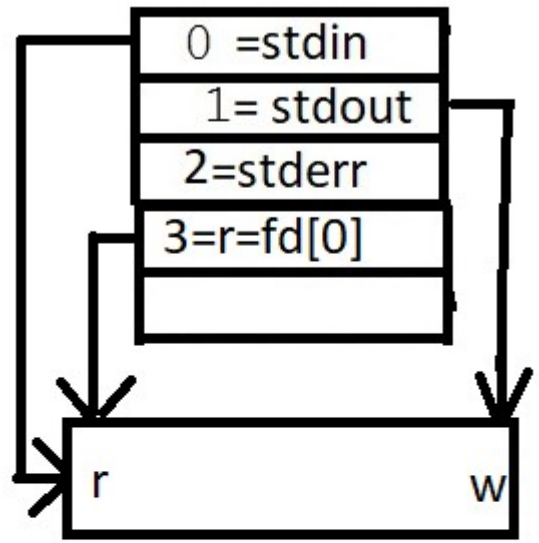


back to parent

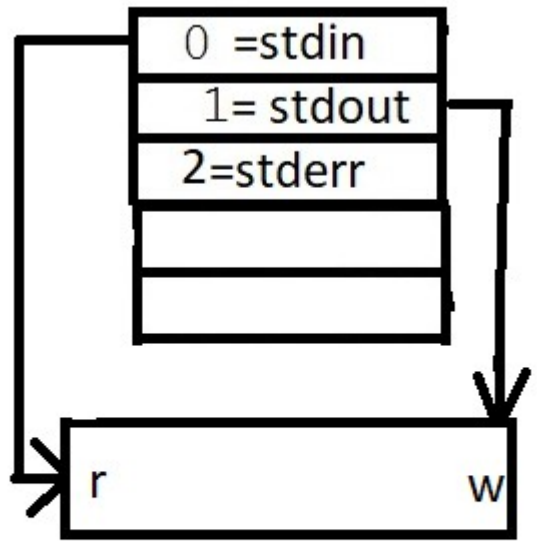
will setup to execute the wc command



back to parent

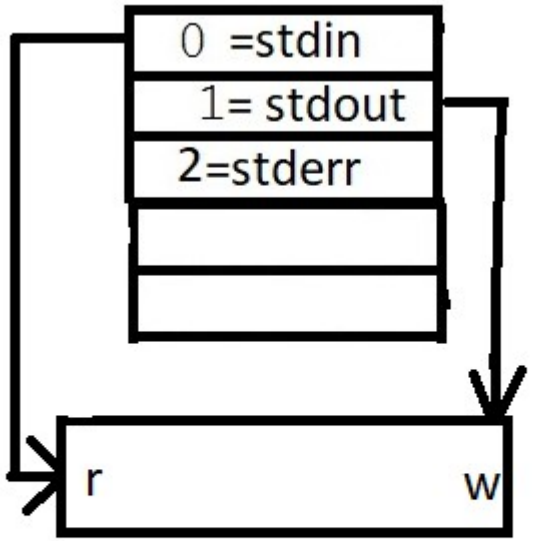


dup2(restoreFd, 0) ;



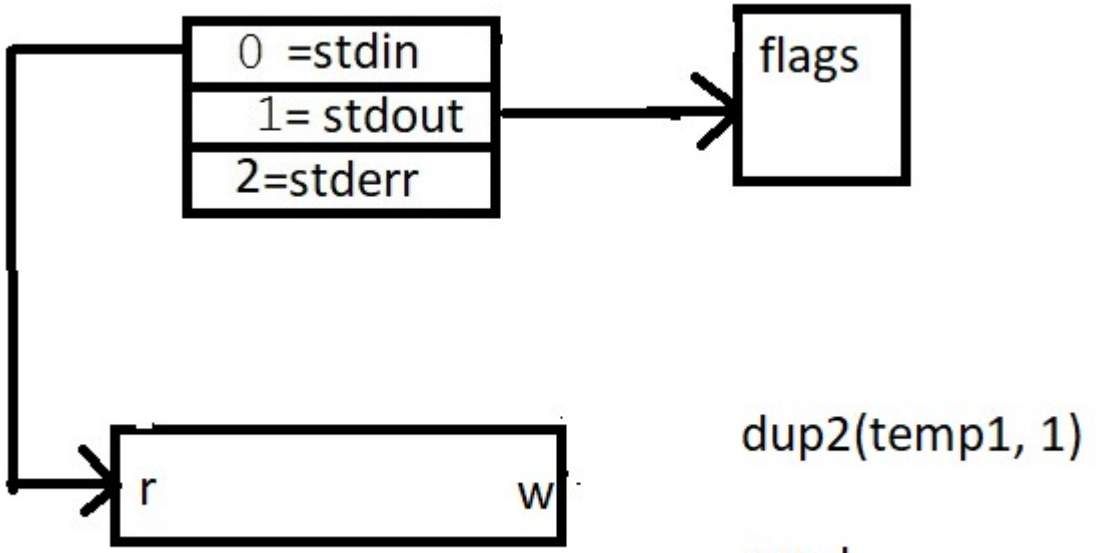
close(restoreFd)

will execute wc



fork ( )
execl

make sure the last command prints to stdout
we use dup2(temp1, 1) and then execl



Compile multiple files using Makefiles

Most of the time in commercial applications, the programs are written in many files. Many developers create or manage programs written in various files.

How do we then compile all these programs ? One method is compiling them individually and linking them together. This process is not efficient as you will see in the videos. If a developer changes a file, then the entire list of program files have to be compiled. Can you imagine 1000s of files need to be compiled again even though those files have not changed over time. This will take days to compile especially if you are developing big products such as operating systems.

To reduce the time, we use Makefiles. Makefile compiles only files that have changed since compiled last time. If the file has not changed, compiler won't touch them and reuses them during the linking process.

Say have files a.c, b.c, d.c and we compiled all files using Makefiles on Thursday. A developer changed one file say a.c on Friday. If we run the makefile, it will only compile the file a.c, not the other files b.c and d.c . This is because the only file that was edited since Thursday is a.c. Makefile recognizes this. During the linking process, it will reuse the compiled code of b.c and d.c to generate the executable file.

PROGRAM DEMO :

We will be writing a simple functions to add two numbers, subtract two numbers, multiply two numbers.

The prototype of these functions are:

int add (int, int);

int subtract (int, int) ;

int multiply (int, int) ;

and these are defined/written in different files, namely add.c, subtract.c and multiply.c

The make file will compile individual files and link them together.

We compile multiple files using makefiles.

here is the document : **Compiling Files independently and linking them together.docx (https://csus.instructure.com/courses/94454/files/14915323/download?wrap=1)**

Compiling Files independently and linking them together

In a real world situation, several developers work on a product creating and modifying functions and files they created or someone created for them. The normal compilation and linking we have so far being doing with the command gcc should be changed so developer can just compile , not link to the main function. When everyone completed their developing their function, we link them together to generate the main product or program.

Let us take a calculator example program having several functions such as add, subtract, and multiply. The add function

```
// add.c : function add
#include <stdio.h>
int add ( int x, int y )
{
   printf ( " Inside the function add \n") ;
   return x + y ;
}
```

and we will code this a file add.c .

As you can see, there is no main function within this file. We can compile this piece of code using the command line :

gcc -c add.c -o add

the command will look for compilation errors and generates an object file without linking to any other file. The –o option specifies the output file and –c option as per the man page

-c Compile or assemble the source files, but do not link. The linking stage simply is not done. The ultimate output is in the form of an object file for each source file.

We will add similar piece of code for subtract in a file subtract.c

```
// subtract.c : function subtract
#include <stdio.h>
int subtract ( int x, int y )
{
   printf ( " Inside the function subtract \n") ;
```

```
   return x - y ;
}
```

gcc –c subtract.c -o sub

and similarly for multiply

```
//  multiply.c  :  function multiply
#include <stdio.h>
int multiply ( int x, int y )
{
   printf ( " Inside the function multiply  \n") ;
   return x * y ;
}
```

gcc –c multiply.c –o multiply.o

We have three object files with no main function. Let us create the main function

```
//  main.c  :  main function
#include <stdio.h>
int main (  )
{
   // let us make this very simple
   int value = add ( 5, 4 );
   printf  ( " 5 + 4 = %d \n",  value  );

   value = subtract ( 5, 4 );
   printf  ( " 5 - 4 = %d \n",  value  );

   value = multiply ( 5, 4 );
   printf  ( " 5 x 4 = %d \n",  value  );

}
```

gcc main.c –o calculator

But you will get compilation errors because the compiler is trying to figure the definition of add, multiply and subtract. Because it couldn't find, it will end with linking errors.

How do we link those functions ? There are two ways

```
gcc main.c add.c subtract.c multiply.c -o calculator
```

in this all source codes will be compiled together taking long time
or

gcc main.c add.o subtract.o multiply.o –o calculator

main.c is compiled and linked with already compiled code of add.o sub.o and multiply.o .

Now the compiler find the add function being defined in the object file add.o, subtract function defined in subtract.o and multiply function in multiply.o , generating the final product calculator.

So, whenever you develop a function(s) in a separate file(s), just make sure it compiles, and generate the object file. Then, you will link all of them together.

Advantage with this process of developing programs is multiple programmers will be able to develop programs simultaneously without waiting for the other to finish.

> But the problem with this approach is the all source code should be compiled from the scratch and/or linking with old files when programmers have new code available.

The solution to both method is makefiles.

Makefile monitor for changes in the source code. If the timestamp of the source code of a function in a file is newer than the timestamp of the executable, it will compile only the source code of that function, it will not compile the entire source code saving considerable amount of time.

For instance, say our executable is compiled on Tuesday 5PM and the source code of add.c was modified at 6PM, makefiles compile only the add.c code and link with the rest of the code. Can you imagine the time it saves when you are amongst 10000 developers. You change one file and the entire process will take only your file, not files created by 10000 developers. What a nice piece of creativity ?

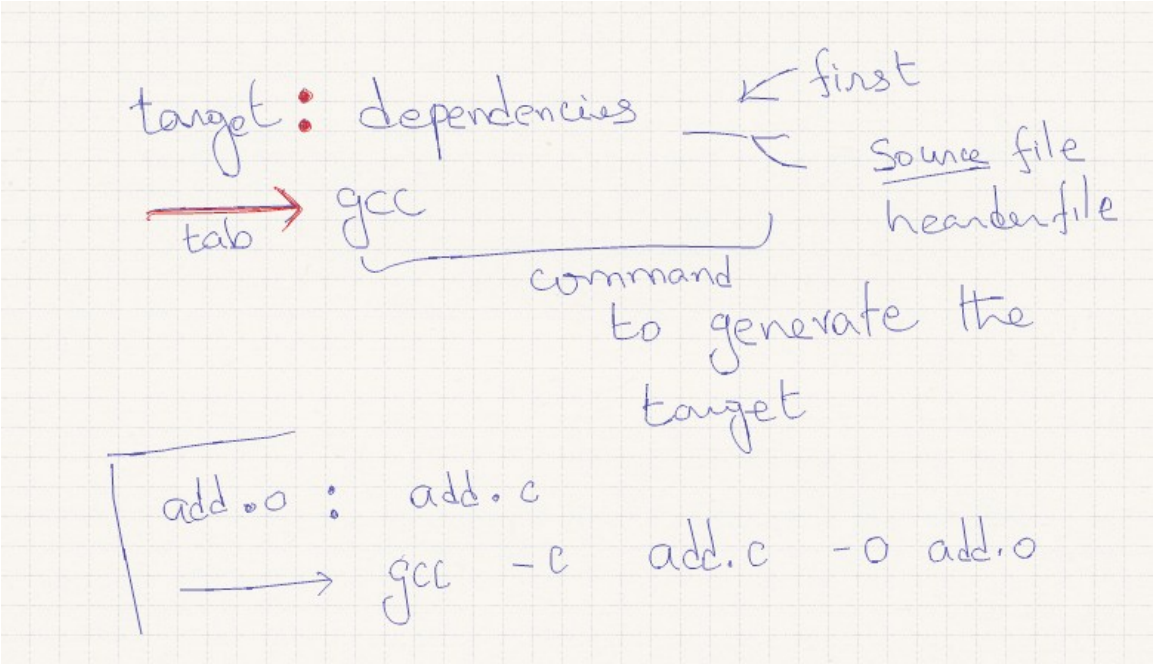
Makefiles

Makefile contains pair of two lines : first line specifies object file (or target) to be created followed by <colon> and followed by the dependencies the target depends. In the next line starting with a tab, then specify the commands to compile to generate a target. For instance,

add.o : add.c
<tab>gcc –c add.c

In the above lines, add.o is the target we are tryig to generate, followed by <colon> followed by dependency. In the second line, we enter tab first followed by the command.

Here is the screen shot we did in the video



The entire makefile is

```
c-prog>cat mymake
# add file
add.o : add.c
        gcc -c add.c -o add.o

#subtract
sub.o : subtract.c
        gcc -c subtract.c -o sub.o

#multiply
multiply.o : multiply.c
        gcc -c multiply.c -o multiply.o

# executable
calculator: add.o sub.o multiply.o main.c
        gcc main.c add.o sub.o multiply.o -o calculator

clean :
        rm -f add.o sub.o multiply.o calculator
c-prog>
```

To clean and remove all object file, we run the command
make –f mymake clean

the target is clean, it run the shell command rm –f , f stands for forcefully remove without prompting or any errors.

to generate the executable calculator (the target here) , we run the command
make –f mymake calculator

to generate only an object file say add (the target here) , we run the command
make –f mymake add

Makefile example

```
// add.c


int add ( int x, int y )
{

return ( x + y) ;

}

// sub.c
int sub ( int x, int y )
{
return x - y;
}


// main.c
#include <stdio.h>
int main ( )
{
int a = add ( 3, 4);
int b = sub ( 4, 2);

printf ( "add= %d \n", a);
printf ( "sub= %d \n", b);
}


// REAL MAKE FILE

// mymake


add.o: add.c
<tab>gcc -c add.c -o add.o

sub.o: sub.c
<tab> gcc -c sub.c -o sub.o

main.o: main.c
<tab>gcc -c main.c -o main.o

mycalc: add.o sub.o main.o
<tab>gcc add.o sub.o main.o -o calculator




// shell command to do make file

make -f mymake mycalc
```

Why to Debug a Program?

When you write small programs (say 3-4 lines of code), and your program is not behaving correctly, you can debug your program inserting printf statements to print value of variables, manually trace the variables and values, find the statement that is written incorrectly, rectify and finally run the program successfully.

But, when you write larger programs, you should not debug your programs using printf statements. Instead use a debug tool, we use gdb tool in Linux environment.

What is gdb ?

You can use GDB to debug programs written in C, C++. The purpose of a debugger such as GDB is to allow you to see what is going on ``inside'' a program while it executes.

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

GDB is invoked with the shell command gdb. Once started, it reads commands from the terminal until you tell it to exit with the GDB command- quit.

First, let us take a simple program, say swap.c that swaps the values of two variables.

```c
#include <stdio.h>

main (  )

{
  int i = 3;
  int j = 5;
  int k;

  printf ("Before Swapping \n");
  printf ("i=%d j=%d \n", i, j);

  k = i;
  i = j;
  j = k;

  printf ("After Swapping \n");
  printf ("i=%d j=%d \n", i, j);

}
```

We compile the program using –g option.

gcc swap.c –g –o swap

The –g option compiles swap.c, embeds debugging information for the gdb tool to support debugging.

To start debugging our program swap , we type

gdb swap

Then gdb loads our program and prompts us to issue more commands. Now we are under debugger mode. Now we can stop the program at a particular line or function, step through each and every line if we want to, print the variables and much more.

Here is the list of commands to use for debugging:

run	run the program.
break N	put a break point at line number N
break main	put a break point at function main
break function_name	put a break at the start of the function
break N if I == 5	break if the value of I == 5 at line number N
step	execute next line only and stop
step 3	execute three lines and stop
continue	continue until next break point
info break	print the list of break points
finish	finish the current function
next	almost same as step, but won't stop at

	functions unless a break point is set
watch variable	stops whenever the variable changes value,
delete	delete all breakpoints
delete N	delete breakpoint number N
clear	delete breakpoints
disable N	do not delete breakpoint N, just disable now
enable N	enable the breakpoint
backtrace	Show trace, functions, and print the stack
set variable variable= value	sets the value to the variable
set logging on	enable logs
set logging off	disable logs, Default name of file is gdb.txt
Set logging file filename	change the name of the current logfile
save breakpoints break_filename	saves all breakpoints in a file
source break_filename	loads all breakpoints
list line-number	print the source code around the line number
list function	print the source code around the function
quit	logout
gdb a.out core	in case a.out crashes with segmentation fault
print variable	prints the value of variable
print array@5	will print the addresses of 5 cells
print array[0]@5	will print the values the five cells
whatis variable	prints the type of the variable
where	

Passing functions using pointers

Passing functions to functions

Now we will see how functions could be passed to other functions. in other words, how function name and parameters could be passed as one of the parameters.

WHY Do we have to pass function?

Now you may ask why do have to pass functions to other functions. This is predominantly used in system programming and handle events that arrive at random. When events arrive at random, we need a function to handle or process that event. But the event may happen outside of our process:

Now take this example: Our program is running in a process, reading and writing data to files. The user types Control-C to terminate the program. The OS receives this Control-C key stroke as en event, sees the process ID, and checks if there is a function to handle this event in our program. Because we didn't write such a function, it executes a default function to terminate this program. But if you prefer to handle this control-C event on your own, you could write a function - aka signal handler - to handle this event. In that case, you register this function with the OS. Next time, when you type Control-C while your program is running, the OS launches this function, but in User space, not in kernel space, under the same process. When this signal handler is executing, reading and writing from files would commence . Without registering this signal handlers, reading and writing would stop and terminate the process.

Here, we will just learn about passing functions. Later we will learn about signal handlers.

Let us take up the same add and minus function

```
#include <stdio.h>

int add ( int x, int y) // add is a function name, as well as pointer to a function

{

   return x + y;

}

int minus ( int x, int y)

{

   return x -y ;

}
```

Let us add a new function called print that will take a function and its arguments. Then, it will execute the function.

```
void print (    int  ( * ptr ) ( int, int ) ,  int x, int y )
```

```
{
    int value =  ptr ( x, y ) ;
    printf ( " The value printed is %d \n",  value ) ;
}

main  (  )
{
    print (   add,  4, 5 ) ;
    print ( minus, 5, 1 ) ;
}
```

Sample code using pthreads

```
void * print_greeting( void *ptr )
{
     printf("%s \n", (char *) ptr);
}

main()
{
     pthread_t thread1, thread2 ;
     char *msg1 = "Hello from 1";
     char *msg2 = "Hello from 2";
     int  threadOne ;
     int  threadTwo ;

     threadOne = pthread_create( &thread1, NULL, print_greeting, (void*) msg1);
     printf("Thread One return : %d\n",threadOne);

     threadTwo = pthread_create( &thread2, NULL, print_greeting, (void*) msg2);
     printf("Thread Two return : %d\n",threadTwo);

     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);

}
```

The above code is a simple thread program. Two threads are simply printing the string passed to them. PAY attention to the pthread_join arguments.

Now the next version creates five threads and prints the process ID and thread ID

```
#include <pthread.h>
#include <stdlib.h>
#define NUM_THREADS     5

void *PrintHello( )
{
   pid_t pid = pthread_self ( ) ;
   printf("My process #%x, my thread ID #%x\n", getpid( ) , pid );
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long x;
   for(x=0; x<NUM_THREADS; x++){
      rc = pthread_create(&threads[x], NULL, PrintHello, NULL );
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }

   for(x=0; x < NUM_THREADS; x++)
      pthread_join ( threads[x] , NULL );

}
```

In the next version , we print the number of odd numbers counted by two threads , but the result will be incorrect as they overwrite the count value.

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

int count = 0;  // SHARED VARIABLE AMONG THREADS

void *countOddNumbers( void *ptr )  // FUNCTION PASSED TO THREADS
{
  int low, high;
  low =  * ( (int *)ptr) ;
  high = low + 500000 ;

  int i ;
  for (i = low ; i < high ; i++)  {
        if  ( ( i % 2 ) == 0 ) {
           count++; // increment if odd.  THIS IS THE SHARED VARIABLE
        }
  }
   pthread_exit((void*) 0);
}

main()
{
     pthread_t T1, T2;
     int start_t1 = 1;
     int start_t2 = 500001;

     int id1 = pthread_create( &T1, NULL, countOddNumbers, (void*)&start_t1);
     int id2 = pthread_create( &T2, NULL, countOddNumbers, (void*)&start_t2);

     if ( id1 ) perror ( "thread creation error ");
     if ( id2 ) perror ( "thread creation error ");

     pthread_join( T1, NULL);
     pthread_join( T2, NULL);

     printf ( "The number of odd numbers is:  %d  \n", count );
}
```

The above code will result in incorrect value , it should be 1/2 million, but we saw it is not.

We solve the problem using Mutex:

pthread-count-odd-numbers-ver2.mp4 (https://csus.instructure.com/courses/94454/files/14915301/download?wrap=1) (https://csus.instructure.com/courses/94454/files/14915301/download?download_frd=1)

Here is the code that fixes the above program using Mutex:

```c
#include <stdlib.h>
#include <pthread.h>
int count = 0;  // SHARED VARIABLE AMONG THREADS
pthread_mutex_t myMutex = PTHREAD_MUTEX_INITIALIZER;

void *countOddNumbers( void *ptr )  // FUNCTION PASSED TO THREADS
{
  int low, high;
  low =  * ( (int *)ptr) ;
  high = low + 500000 ;

  int i ;
  for (i = low ; i < high ; i++)  {
        if  ( ( i % 2 ) == 0 ) {
           pthread_mutex_lock ( &myMutex);
           count++; // increment if odd.  THIS IS THE SHARED VARIABLE
           pthread_mutex_unlock ( &myMutex);
        }
  }
   pthread_exit((void*) 0);
}

main()
{
     pthread_t T1, T2;
     int start_t1 = 1;
     int start_t2 = 500001;

     int id1 = pthread_create( &T1, NULL, countOddNumbers, (void*)&start_t1);
     int id2 = pthread_create( &T2, NULL, countOddNumbers, (void*)&start_t2);

     if ( id1 ) perror ( "thread creation error ");
     if ( id2 ) perror ( "thread creation error ");

     pthread_join( T1, NULL);
     pthread_join( T2, NULL);

     printf ( "The number of odd numbers is:  %d  \n", count );
}
```

pthreads

According to Lawrence Livermore National Laboratory, I am quoting the text from their website

- Technically, a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system. But what does this mean?
- To the software developer, the concept of a "procedure" that runs independently from its main program may best describe a thread.
- To go one step further, imagine a main program (a.out) that contains a number of procedures. Then imagine all of these procedures being able to be scheduled to run simultaneously and/or independently by the operating system. That would describe a "multi-threaded" program

While fork is a costly process because it clones the entire program, pthreads are light weight because only a function is made to run in a separate thread.

Because thread have to be portable across machines, there is a library called pthreads (named after posix threads). Threaded Programs written using pthreads are portable across Linux platforms and any system that support pthreads.

Fork function creates child process while threads do not create new child processes.

Pthread Create Function

So how do we create a thread ? You have to revisit passing functions to functions.

The library function that creates threads is pthread_create. This pthread_create function takes another user defined function as a parameter (remember passing functions to functions), and the arguments also.

The man pages of pthread_create ( ) is

NAME

pthread_create - create a new thread

SYNOPSIS

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);

Compile and link with -pthread.
```

thread handle

can be NULL for now

name of the function to pass and arguments passed separately

DESCRIPTION

The **pthread_create**() function starts a new thread in the calling process. The new thread starts execution by invoking start_routine(); arg is passed as the sole argument of start_routine().

couple of observations:

the first argument pthread_t * thread is a pointer we pass.

second argument pthread_attr_t - we could ignore this and we will keep it NULL

third argument void * (*start_routine) (void *) -

A very trick argument. But don't despair. We will figure it out. What this argument says - *start_routine should be pointer to a function, or function name itself.

(void *) is a pointer to the argument to the function. Unfortunately, we could pass only one variable, that type could be structure, array, int, char, float

and void * on the left says the function returns a pointer

Fourth argument is *arg - we pass the starting numbers for each thread.

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
    long tid;
    tid = (long)threadid;
    pid_t pid = pthread_self ( ) ;
```

```
    printf("In the function! my process #%x\n", getpid( ) ) ;
    printf("In the function! my thread #%x\n", pid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
     pthread_t threads[NUM_THREADS];
     int rc;
     long t;
     for(t=0; t<NUM_THREADS; t++){
     rc = pthread_create(&threads[t], NULL, PrintHello, (void *)&t);
       if (rc) {
          printf("ERROR; return code from pthread_create() is %d\n", rc);
          exit(-1);
       }
}

/* Last thing that main() should do */
pthread_exit(NULL);
}
```

Second Program listed here :

This program counts the number of odd numbers from 1 to 1M using two threads. Thread1 counts from 1 to half a million and the other thread counts from half a million to 1M. Check the results

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>


int count = 0;  // SHARED VARIABLE AMONG THREADS


void *  countOddNumbers  ( void *ptr )  // FUNCTION PASSED TO THREADS

{

  int low, high ;

  low =  * (  ( int * ) ptr  )  ;  //  low is 1 for Thread1,  500000 for thread 2.

  high = low + 500000  ;  // add 500000 to each thread to distribute the work.


  int i  ;
```

```
  for (  i = low ;  i < high  ;   i++   ) {
        if  (    ( i % 2 )   == 1    ) {   // CHECK FOR ODD NUMBERS
           count++  ;    //  shared variable.  This is the problem we will discuss later.
         }
  }
   pthread_exit (   ( void  * ) 0  ) ;
}

main()
{
    pthread_t T1, T2;
    int start_t1 = 1;
    int start_t2 = 500000;
    int  id1, id2 ;
    id1 = pthread_create(  &T1, NULL,   countOddNumbers, (void*) &start_t1);
    id2 = pthread_create(  &T2,  NULL,  countOddNumbers, (void*) &start_t2);

    if ( id1 ) perror ( "thread creation error ");
    if ( id2 ) perror ( "thread creation error ");
    pthread_join( T1, NULL);
    pthread_join( T2, NULL);
    int i ;
    printf ( "The number of odd numbers is:  %d  \n", count );

pthread_exit(NULL);
}
```

All compile programs with link -pthread

See how this program works in this video:

pthread_count_odd_numbers.mp4 (https://csus.instructure.com/courses/94454/files/14915280/download?wrap=1) ↓
(https://csus.instructure.com/courses/94454/files/14915280/download?download_frd=1)

Semaphores

Consider this: You want to meet the chairwomen of the department, but the secretary in guarding the door. If another student is already inside, you wait. As soon as the student exits, the secretary lets you in. Only one student can be visiting the chair. We could say the chair of the department is the Critical Resource/Section and the secretary is the semaphore. In systems, a binary semaphore is a kernel-maintained integer variable whose value is restricted to being 0 or 1 . When a process (Say A) arrives to access the critical section (CS) when no other process inside, the process requests the kernel for access. The kernel decrements this variable (now it is 0) and the process goes ahead to access the CS. In the mean time, another process (say B) arrives to access the CS. Because the value is zero, the kernel makes it to wait until the value goes back to 1. This semaphore is called binary semaphore. There are semaphores that are not binary. Say, a restaurant has a capacity to serve only 20 people. If people inside exceeds 20, the new incoming people have to wait until there is a space. In this situation, you initialize the semaphore variable to 20. This is how mutual exclusion is implemented. Because kernel manages this variable, synchronization is easily implemented.

There are two types of POSIX semaphores : Named and Unnamed Semaphores.

Named semaphores: This type of semaphore has a name. By calling sem_open() with the same name, unrelated processes can access the same semaphore.

Unnamed semaphores: This type of semaphore doesn't have a name; instead, it resides at an agreed-upon location in memory. Unnamed semaphores can be shared between processes or between a group of threads. We will not discuss unnamed semaphores as the p

POSIX semaphores is an integer whose value is not permitted to fall below 0. If a process attempts to decrease the value of a semaphore below 0, then, depending on the function used, the call either blocks or fails with an error indicating that the operation was not currently possible.

Named Semaphore:
To work with a named semaphore, we use various functions :

- The sem_open() function opens or creates a new semaphore, initializes the semaphore if it is created by the call, and returns a handle for use in later calls.
- The sem_post(sem) and sem_wait(sem) functions respectively increment and decrement a semaphore's value.
- The sem_getvalue() function retrieves a semaphore's current value.
- The sem_close() function removes the calling process's association with a semaphore that it previously opened. We generally use the sem_unlink instead.
- The sem_unlink() function removes a semaphore name and marks the semaphore for deletion when all processes have closed it.

On Linux, they are created as small POSIX shared memory objects with names of the form sem.name under the directory /dev/shm. This file system has kernel persistence—the semaphore objects that it contains will persist, even if no process currently has them open, but they will be lost if the system is shut down.

Creating and/or Opening a Named Semaphore

The sem_open() function creates and opens a new named semaphore or opens an existing semaphore. If sem_open() is being used to open an existing semaphore, the call requires only two arguments: name and oflag.

```
#include <fcntl.h> /* Defines O_* constants */
#include <sys/stat.h> /* Defines mode constants */
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag, ... /* mode_t mode, unsigned int value */ );
```

Returns pointer to semaphore on success, or SEM_FAILED on error

- The name argument identifies the semaphore.
- The oflag argument is a bit mask that determines whether we are opening an existing semaphore or creating and opening a new semaphore. If oflag is 0, we are accessing an existing semaphore and mode and value arguments are ignored and not needed.

If O_CREAT is specified in flags, then two further arguments are required: mode and value.

- Mode: The mode argument is a bit mask that specifies the permissions to be placed on the new semaphore. We should ensure that both read and write permissions are granted to each category of user—owner, group, and other—that needs to access the semaphore. Typical value would be 666

- Value: The value argument is an unsigned integer that specifies the initial value to be assigned to the new semaphore. The creation and initialization of the semaphore are performed atomically.

If O_CREAT is specified in oflag, then a new semaphore is created if one with the given name doesn't already exist. If oflag specifies both O_CREAT and O_EXCL, and a semaphore with the given name already exists, then sem_open() fails.

Regardless of whether we are creating a new semaphore or opening an existing semaphore, sem_open() returns a pointer to a sem_t value, and we use this pointer in subsequent calls to functions that operate on the semaphore. On error, sem_open() returns the value SEM_FAILED.

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

#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */
#include <semaphore.h>

int main(int argc, char *argv[])
{
    int flags, opt;
    mode_t perms;
    unsigned int value;
    sem_t *sem;
    flags = 0;

    flags |= O_CREAT;
    flags |= O_EXCL;

    sem = sem_open("/demo", flags, 0600, 0);
    if (sem == SEM_FAILED) {
       perror("sem_open");
       exit(EXIT_FAILURE);
    }

    exit(EXIT_SUCCESS);
}
```

Removing a Named Semaphore

The sem_unlink() function removes the semaphore identified by name and marks the semaphore to be destroyed once all processes cease using it (this may mean immediately, if all processes that had the semaphore open have already closed it).

SAMPLE CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */
#include <semaphore.h>

int main(int argc, char *argv[])
{
 //sem_t *sem;

 //sem_close( "demo1");
 if (sem_unlink("/demo") == -1) {
     perror("sem_unlink");
     exit(EXIT_FAILURE);
  }

exit(EXIT_SUCCESS);
}
```

Wait Operation on a Semaphore

The sem_wait() function decrements (decreases by 1) the value of the semaphore referred to by sem.

#include <semaphore.h>
int **sem_wait**(sem_t *sem);

If the semaphore currently has a value greater than 0, sem_wait() returns immediately. If the value of the semaphore is currently 0, sem_wait() blocks until the semaphore value rises above 0; at that time, the semaphore is then decremented and sem_wait() returns

```
#include <stdlib.h>
#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */
#include <semaphore.h>

int main(int argc, char *argv[])
{
 sem_t *sem;
 sem = sem_open("/demo", 0);
   if (sem == SEM_FAILED) {
       perror("sem_open");
       exit(EXIT_FAILURE);
    }

   if (sem_wait(sem) == -1) {
       perror("sem_wait");
       exit(EXIT_FAILURE);
    }


    printf("%ld sem_wait() succeeded\n", (long) getpid());

   exit(EXIT_SUCCESS);

}
```

Posting a Semaphore

The sem_post() function increments (increases by 1) the value of the semaphore referred to by sem.

int **sem_post**(sem_t *sem);

Posting is incrementing a POSIX semaphore. If there are processes waiting to access the CS, they are released. They all have to do a wait operation and one of them (depending on scheduling algorithms) will be given access to the CS.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */
#include <semaphore.h>

int main(int argc, char *argv[])
{
 sem_t *sem;
 sem = sem_open("/demo", 0);
   if (sem == SEM_FAILED) {
       perror("sem_open");
       exit(EXIT_FAILURE);
    }


     if (sem_post(sem) == -1){
       perror("sem_post");
       exit(EXIT_FAILURE);
    }


   exit(EXIT_SUCCESS);

}
```

Getting Current Value of the Semaphore

The sem_getvalue() function returns the current value of the semaphore referred to by sem in the int pointed to by sval.

int **sem_getvalue**(sem_t *sem, int *sval);
Returns 0 on success, or –1 on error

If one or more processes (or threads) are currently blocked waiting to decrement the semaphore's value, then the value returned in sval depends on the implementation. The Unix standard permits two possibilities: 0 or a negative number whose absolute value is the number of waiters blocked in sem_wait(). Linux and several other implementations adopt the former behavior; a few other implementations adopt the latter behavior. Note that by the time sem_getvalue() returns, the value returned in sval may already be out of date.

```
#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */
#include <semaphore.h>

int main(int argc, char *argv[])
{
 sem_t *sem;
 sem = sem_open("/demo", 0);
   if (sem == SEM_FAILED) {
       perror("sem_open");
       exit(EXIT_FAILURE);
    }
  int value;
   if (sem_getvalue(sem, &value) ) {
       perror("sem_getvalue");
       exit(EXIT_FAILURE);
    }
   printf("%d\n", value);

   exit(EXIT_SUCCESS);

}
```

Closing a Semaphore

When a process opens a named semaphore, the system records the association between the process and the semaphore. The sem_close() function terminates this association (i.e., closes the semaphore), releases any resources that the system has associated with the semaphore for this process, and decreases the count of processes referencing the semaphore. Open named semaphores are also automatically closed on process termination or if the process performs an exec(). Closing a semaphore does not delete it. For that purpose, we need to use sem_unlink().

int **sem_close**(sem_t *sem);
Returns 0 on success, or –1 on error

We will not discuss unnamed semaphores because they are little identical and can be easily used in applications.

SIGNALS AND ALARM

Concepts and Overview

A signal is a notification to a process that an event has occurred. Signals are software interrupts. Most nontrivial application programs need to deal with signals. Signals are analogous to hardware interrupts in that they interrupt the normal flow of execution of a program; in most cases, it is not possible to predict exactly when a signal will arrive. Signals provide a way of handling asynchronous events—for example, a user at a terminal typing the interrupt key to stop a program or the next program in a pipeline terminating prematurely. when a signal is delivered to a process, the process will stop what its doing, either handle or ignore the signal, or in some cases terminate, depending on the signal.

Signals also are delivered in an unpredictable way out of sequence with the program because signals usually originate outside of the current executing process. Another way to view signals is a mechanism for handling **asynchronous events**. As opposed to **synchronous events**, which is when a standard program executing iteratively, one line of code following another, **asynchronous events** is when portions of the program may execute out of order, or not immediately in a iterative style. Asynchronous events are typically due to external events at the interaction layer between the hardware and the operating system; the signal, itself, is the way for the operating system to communicate these events to the processes

One process can (if it has suitable permissions) send a signal to another process. In this use, signals can be employed as a synchronization technique, or even as a primitive form of interprocess communication (IPC). It is also possible for a process to send a signal to itself. However, the usual source of many signals sent to a process is the kernel. Among the types of events that cause the kernel to generate a signal for a process are the following:

- A hardware exception occurred, meaning that the hardware detected a fault condition that was notified to the kernel, which in turn sent a corresponding signal to the process concerned. Examples of hardware exceptions include executing a malformed machine-language instruction, dividing by 0, or referencing a part of memory that is inaccessible. See sample programs at the end of these document.

- The user typed one of the terminal special characters that generate signals. These characters include the interrupt character (usually Control-C) and the suspend character (usually Control-Z).

- A software event occurred. For example, input became available on a file descriptor, the terminal window was resized, a timer went off, the process's CPU time limit was exceeded, or a child of this process terminated.

First, every signal has an unique name and all begin with the three characters SIG. For example, SIGABRT is the abort signal that is generated when a process calls the abort function. SIGALRM is the alarm signal that is generated when the timer set by the alarm function goes off. When the user types the interrupt character, SIGINT (signal number 2) is delivered to a process. Each signal name is defined as a unique (small) integer, starting sequentially from 1. These integers are defined in <signal.h> with symbolic names of the form SIGxxxx. Since the actual numbers used for each signal vary across implementations, it is these symbolic names that are always used in programs.

Signals fall into two broad categories. The first set constitutes the traditional or standard signals, which are used by the kernel to notify processes of events. On Linux, the standard signals are numbered from 1 to 31. We describe the standard signals in this document.

DO NOT DISTURB : Can a running process block signals ?

Between the time a signal is generated externally and the time it is delivered to the process, a signal is said to be pending. Normally, a pending signal is delivered to a process as soon as it is next scheduled to run, or immediately if the process is already running (e.g., if the process sent a signal to itself). Sometimes, however, we need to ensure that a segment of code is not interrupted by the delivery of a signal. To do this, we can add a signal to the process's signal mask—a set of signals whose delivery is currently blocked. If a signal is generated while it is blocked, it remains pending until it is later unblocked (removed from the signal mask). Various system calls allow a process to add and remove signals from its signal mask. This is much like "DO NOT DISTURB" sign you see in hotel rooms.

SIGNAL DELIVERED:
Upon delivery of a signal, a process carries out one of the following default actions, **depending on the signal**:

- The signal is ignored; that is, it is discarded by the kernel and has no effect on the process. (The process never even knows that it occurred.)
- The process is terminated (killed). This is sometimes referred to as abnormal process termination, as opposed to the normal process termination that occurs when a process terminates using exit().
- A core dump file is generated, and the process is terminated. A core dump file contains an image of the virtual memory of the process, which can be loaded into a debugger in order to inspect the state of the process at the time that it terminated.
- The process is stopped—execution of the process is suspended.
- Execution of the process is resumed after previously being stopped.

Instead of accepting the default for a particular signal, a program can change the action that occurs when the signal is delivered. This is known as setting the disposition of the signal. A program can set one of the following dispositions for a signal:

- The default action should occur. This is useful to undo an earlier change of the disposition of the signal to something other than its default.
- The signal is ignored. This is useful for a signal whose default action would be to terminate the process.
- A signal handler is executed.

What is a signal handler ? A signal handler is a function, written by the programmer, that performs appropriate tasks in response to the delivery of a signal. For example, the shell has a handler for the SIGINT signal (generated by the interrupt character, Control-C) that causes it to stop what it is currently doing and return control to the main input loop, so that the user is once more presented with the shell prompt. Notifying the kernel that a handler function should be invoked is usually referred to as installing or establishing a signal handler. When a signal handler is invoked in response to the delivery of a signal, we say that the signal has been handled or, synonymously, caught.

Signal Types and Default Actions

Earlier, we mentioned that the standard signals are numbered from 1 to 31 on Linux. However, the Linux signal(7) manual page lists more than 31 signal names. The excess names can be accounted for in a variety of ways. Some of the names are simply synonyms for other names, and are defined for source compatibility with other UNIX implementations. Other names are defined but unused. The following list describes the various signals:

SIGABRT	A process is sent this signal when it calls the abort() function. By default, this signal terminates the process with a core dump. This achieves the intended purpose of the abort() call: to produce a core dump for debugging.
SIGALRM	The kernel generates this signal upon the expiration of a real-time timer set by a call to alarm()or setitimer(). A real-time timer is one that counts according to wall clock time (i.e., the human notion of elapsed time).
SIGBUS	This signal (“bus error”) is generated to indicate certain kinds of memory access errors. One such error can occur when using memory mappings created with mmap(), if we attempt to access an address that lies beyond the end of the underlying memory-mapped file
SIGCHLD	This signal is sent (by the kernel) to a parent process when one of its children terminates (either by calling exit() or as a result of being killed by a signal). It may also be sent to a process when one of its children is stopped or resumed by a signal.
SIGCONT	When sent to a stopped process, this signal causes the process to resume (i.e., to be rescheduled to run at some later time). When received by a process that is not currently stopped, this signal is ignored by default. A process may catch this signal, so that it carries out some action when it resumes

SIGFPE	This signal is generated for certain types of arithmetic errors, such as divide-by-zero. The suffixFPE is an abbreviation for floating-point exception, although this signal can also be generated for integer arithmetic errors. The precise details of when this signal is generated depend on the hardware architecture and the settings of CPU control registers. For example, on x86-32, integer divide-by-zero always yields a SIGFPE,
SIGINT	When the user types the terminal interrupt character (usually Control-C), the terminal driver sends this signal to the foreground process group. The default action for this signal is to terminate the process.
SIGKILL	This is the sure kill signal. It can't be blocked, ignored, or caught by a handler, and thus always terminates a process.
SIGPIPE	This signal is generated when a process tries to write to a pipe, a FIFO, or a socket for which there is no corresponding reader process. This normally occurs because the reading process has closed its file descriptor for the IPC channel
SIGQUIT	When the user types the quit character (usually Control-\) on the keyboard, this signal is sent to the foreground process group. By default, this signal terminates a process and causes it to produce a core dump, which can then be used for debugging. Using SIGQUIT in this manner is useful with a program that is stuck in an infinite loop or is otherwise not responding. By typingControl-\ and then loading the resulting core dump with the gdb debugger and using thebacktrace command to obtain a stack trace, we can find out which part of the program code was executing.
SIGTERM	This is the standard signal used for terminating a process and is the default signal sent by thekill and killall commands. Users sometimes explicitly send the SIGKILL signal to a process usingkill –KILL or kill –9. However, this is generally a mistake. A well-designed application will have a handler for SIGTERM that causes the application to exit gracefully, cleaning up temporary files and releasing other resources beforehand. Killing a process with SIGKILL bypasses theSIGTERM handler. Thus, we should always first attempt to terminate a process using SIGTERM, and reserve SIGKILL as a last resort for killing runaway processes that don't respond toSIGTERM
SIGTSTP	This is the job-control stop signal, sent to stop the foreground process group when the user types the suspend character (usually Control-Z) on the keyboard.

The #define values are coded in <sys/signal.h> and some of the values are

```
#define SIGHUP  1       /* hangup */
#define SIGINT  2       /* interrupt */
#define SIGQUIT 3       /* quit */
#define SIGILL  4       /* illegal instruction */
#define SIGABRT 6       /* abort() */
#define SIGFPE  8       /* floating point exception */
#define SIGKILL 9       /* kill (cannot be caught or ignored) */
#define SIGBUS  10      /* bus error */
```

```
#define SIGSEGV 11      /* segmentation violation */
#define SIGSYS  12      /* bad argument to system call */
#define SIGPIPE 13      /* write on a pipe with no one to read it */
#define SIGALRM 14      /* alarm clock */
#define SIGTERM 15      /* software termination signal from kill */
#define SIGURG  16      /* urgent condition on IO channel */
#define SIGSTOP 17      /* sendable stop signal not from tty */
#define SIGCHLD 20      /* to parent on child stop or exit */
#define SIGWINCH 28     /* window size changes */
#define SIGUSR1 30      /* user defined signal 1 */
#define SIGUSR2 31      /* user defined signal 2 */
```

Implementation of Signal Handlers

A signal handler (also called a signal catcher) is a function that is called when a specified signal is delivered to a process. Invocation of a signal handler may interrupt the main program flow at any time; the kernel calls the handler on the process's behalf, and when the handler returns, execution of the program resumes at the point where the handler interrupted it. Although signal handlers can do virtually anything, they should, in general, be designed to be as simple as possible.

SIGINT

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

In the main function, we have registered our signal handler controlC with our operating system using the signal function. This main program will be in an infinite loop. In the signal handler, we are printing a statement and returning immediately. Generally, we don't print in the signal handlers, but for our purposes, we will use a printf statement. The parameter signum is not utilized though the OS will pass us the value.

Some key points to note from this program is that the second argument to signal() is a function pointer, a reference to a function to call. This tells the operating system that whenever this signal is sent to this process, run this function as the signal handler.

Also, the execution of the signal handler is asynchronous, which means the current state of the program will be paused while the signalhandler executes, and then execution will resume from the pause point, much like context switching.

```
void controlC ( int signum)
{
 printf ( "Thank you for pressing control-C\n");
}

 int  main ( )
{
 signal ( SIGINT, controlC)  ;  // registering our signal handler function with OS

  while ( 1 ) ;  // infinite loop
  exit ( 0 ) ;
}
```

Compile the above program and run. This program will be in an infinite loop. To see this signal in action, press control-c . The shell passes the control-C key stroke to the Operating system, OS will call the signal handler. Remember, before the function will be executed, our program is executing the while loop. The while loop will be stopped, the signal handler will be executing. When the handler finishes the printf statement, the control goes back to the while loop.

Now, if you prefer to exit the program in the handler, you can insert exit statement.

```
void controlC ( int signum)
{
 printf ( "Thank you for pressing control-C\n");
 exit ( 0 ) ;
}
```

SIGQUIT

```
void controlQuit ( int signum)
{
 printf ( "Thank you for pressing control-backslash \n");
 exit ( 0 ) ;
}
int main ( )
{
 signal( SIGQUIT, controlQuit);
 while ( 1 ) ;
 exit ( 0 ) ;
}
```

In the above program, we are registering a signal handler for SIGQUIT signal. The main program will be infinite loop. To generate the signal, press control \ (backslash) . You will see the program printing the statement and quitting the program.

SIGWINCH

As the name implies, this signal is sent whenever the size of the WINdow is CHanged.

```
void windowChange ( int signum)
{
  printf ( "Window size changed\n");
}
int main ( )
{
 signal( SIGWINCH, windowChange ) ;
 while ( 1 ) ;
 exit ( 0 ) ;
}
```

In the above program, we are registering a signal handler to receive signals whenever the size of the Window Changes. When you run the program, try to change the size of the window by resizing it. Note: We are exiting out of the handler in this program.

SIGCHD

```
void childExit ( int signum )
{
  printf ( "Child process Terminated \n");
  exit ( 0 ) ;
}
int main ( )
{
 signal ( SIGCHLD, childExit ) ;

 int child = fork () ;
 if ( child == 0 ) // child
   {
     sleep (2);  // when the child dies after 2 seconds
                 // parent will be notified.
   }
  else {
    while ( 1 ) ;
  }

 exit ( 0 ) ;
}
```

In the above program, we are registering a signal for the parent process to receive a signal when the child process terminates. In the main program, we are forking a child process. The child process sleep for 2 seconds and dies. When it dies, the parent process gets a signal. Just compile and run. In 2 seconds, you should see the statement in the handler function.

SIGSEGV

```
void sigsegv ( int signum )
{
   printf ( "illegal memory access \n");
   exit ( 1 );
}
int main ( )
{
  signal ( SIGSEGV, sigsegv ) ;
  int x = 10;
  scanf ( "%d", x); // it should &x

  exit ( 0 ) ;
}
```

In the above program, we are registering a signal handler to handle any illegal memory access. In the scanf function, we intentionally removed the address & for x and we are passing the address = 10. When you run program, the system call will place the value entered by the user in address 10 creating an illegal memory access. The kernel will generate a signal and our handler will print and exit.

SIGFPE

```
void sigfpe ( int signum)
{
   printf ( "Floating Point Exception \n");
   exit ( 1 );
}
int main ( )
{
  signal( SIGFPE, sigfpe);
  int x = 1/0;

  exit ( 0 ) ;
}
```

SIGFPE is a signal for floating point exception. When you divide a number by zero, floating point exception signal is generated.

SIGPIPE

```
void sigpipe ( int signum )
{
  printf ( "child may have died, and Pipe broken \n");
  exit ( 0 ) ;
}

int main(void)
{
        int     fd[2] ;
        pipe(fd);
        signal ( SIGPIPE, sigpipe);

    switch ( fork() )
    {
        case 0 : /* child */
                close(fd[1]);
                close(fd[0]);  /* closing read end of pipe */
                exit(0);

        default: /* parent */
                sleep ( 1);
                close(fd[0]);
                write(fd[1], "ABCD\n", 5);
    }
        return(0);
}
```

In the above program, we create a pipe and we close the pipes on the child process. When the parent writes on the pipe, because the child is already terminated, a SIGPIPE signal is generated. You need sleep on the parent process to give enough time for child to terminate .

SIGSTOP The SIGSTOP signal stops the process. It cannot be handled, ignored, or blocked.

The SIGKILL signal is used to cause immediate program termination. It cannot be handled or ignored, and is therefore always fatal. It is also not possible to block this signal.

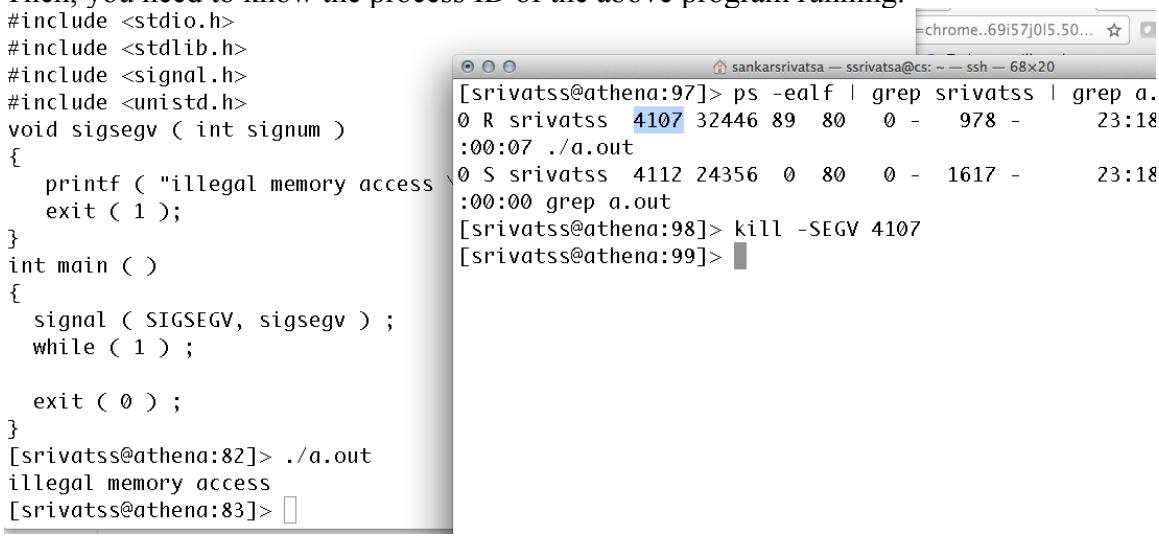
GENERATING SIGNALS FROM COMMAND LINE

So far we have seen signals were triggered in our program. Now we will try to trigger from command line of another shell .

In one shell window, try to run this program from the command line.

```
void sigsegv ( int signum )
{
  printf ( "illegal memory access \n");
  exit ( 1 );
}
int main ( )
{
 signal ( SIGSEGV, sigsegv ) ;
 while ( 1 ) ;

 exit ( 0 ) ;
}
```

Then, you need to know the process ID of the above program running.



In the above diagram, the process ID for the program 4835 .

In another shell terminal, send the signal to the process running 4835 using the command

```
kill -SEGV 4835
or
kill -11 4835
```

The 11 above is the constant value defined for SIGSEGV in the <sys/signal.h>

You can also use killall command , instead of kill, but use the name of the program.

```
killall -SIGKILL a.out
```

Alarms :

A `SIGALRM` signal is an alarm set by the function alarm (int sec) after sec seconds have elapsed since calling the function, it is delivered by the Operating System much like signals we read so far.

```
void buzzMe( int signum )
{
   printf ( "Wake up...\n");
}

int main()
{
  signal ( SIGALRM, buzzMe);
  alarm ( 2 );
  while ( 1  ) ;
}
```

The above looks very similar to the program we wrote earlier. But here, we use SIGALRM signal value to register our alarm with the operating system and the alarm handler is defined. Then, we have to request an alarm , here we use 2 seconds. After the request has been put, we wait using a while loop. You can instead use a function pause or sleep , but these are implementation specific , in general avoid using signals and sleep.

Because only one alarm can used in a program, we cannot set another alarm. In case you plan to use another alarm, the previous alarm will be replaced.

SIGNALS AND ALARM

Concepts and Overview

A signal is a notification to a process that an event has occurred. Signals are software interrupts. Most nontrivial application programs need to deal with signals. Signals are analogous to hardware interrupts in that they interrupt the normal flow of execution of a program; in most cases, it is not possible to predict exactly when a signal will arrive. Signals provide a way of handling asynchronous events—for example, a user at a terminal typing the interrupt key to stop a program or the next program in a pipeline terminating prematurely. when a signal is delivered to a process, the process will stop what its doing, either handle or ignore the signal, or in some cases terminate, depending on the signal.

Signals also are delivered in an unpredictable way out of sequence with the program because signals usually originate outside of the current executing process. Another way to view signals is a mechanism for handling **asynchronous events**. As opposed to **synchronous events**, which is when a standard program executing iteratively, one line of code following another, **asynchronous events** is when portions of the program may execute out of order, or not immediately in a iterative style. Asynchronous events are typically due to external events at the interaction layer between the hardware and the operating system; the signal, itself, is the way for the operating system to communicate these events to the processes

One process can (if it has suitable permissions) send a signal to another process. In this use, signals can be employed as a synchronization technique, or even as a primitive form of interprocess communication (IPC). It is also possible for a process to send a signal to itself. However, the usual source of many signals sent to a process is the kernel. Among the types of events that cause the kernel to generate a signal for a process are the following:

- A hardware exception occurred, meaning that the hardware detected a fault condition that was notified to the kernel, which in turn sent a corresponding signal to the process concerned. Examples of hardware exceptions include executing a malformed machine-language instruction, dividing by 0, or referencing a part of memory that is inaccessible. See sample programs at the end of these document.

- The user typed one of the terminal special characters that generate signals. These characters include the interrupt character (usually Control-C) and the suspend character (usually Control-Z).

- A software event occurred. For example, input became available on a file descriptor, the terminal window was resized, a timer went off, the process's CPU time limit was exceeded, or a child of this process terminated.

First, every signal has an unique name and all begin with the three characters SIG. For example, SIGABRT is the abort signal that is generated when a process calls the abort function. SIGALRM is the alarm signal that is generated when the timer set by the alarm function goes off. When the user types the interrupt character, SIGINT (signal number 2) is delivered to a process. Each signal name is defined as a unique (small) integer, starting sequentially from 1. These integers are defined in <signal.h> with symbolic names of the form SIGxxxx. Since the actual numbers used for each signal vary across implementations, it is these symbolic names that are always used in programs.

Signals fall into two broad categories. The first set constitutes the traditional or standard signals, which are used by the kernel to notify processes of events. On Linux, the standard signals are numbered from 1 to 31. We describe the standard signals in this document.

DO NOT DISTURB : Can a running process block signals ?

Between the time a signal is generated externally and the time it is delivered to the process, a signal is said to be pending. Normally, a pending signal is delivered to a process as soon as it is next scheduled to run, or immediately if the process is already running (e.g., if the process sent a signal to itself). Sometimes, however, we need to ensure that a segment of code is not interrupted by the delivery of a signal. To do this, we can add a signal to the process's signal mask—a set of signals whose delivery is currently blocked. If a signal is generated while it is blocked, it remains pending until it is later unblocked (removed from the signal mask). Various system calls allow a process to add and remove signals from its signal mask. This is much like "DO NOT DISTURB" sign you see in hotel rooms.

SIGNAL DELIVERED:
Upon delivery of a signal, a process carries out one of the following default actions, **depending on the signal**:

- The signal is ignored; that is, it is discarded by the kernel and has no effect on the process. (The process never even knows that it occurred.)
- The process is terminated (killed). This is sometimes referred to as abnormal process termination, as opposed to the normal process termination that occurs when a process terminates using exit().
- A core dump file is generated, and the process is terminated. A core dump file contains an image of the virtual memory of the process, which can be loaded into a debugger in order to inspect the state of the process at the time that it terminated.
- The process is stopped—execution of the process is suspended.
- Execution of the process is resumed after previously being stopped.

Instead of accepting the default for a particular signal, a program can change the action that occurs when the signal is delivered. This is known as setting the disposition of the signal. A program can set one of the following dispositions for a signal:

- The default action should occur. This is useful to undo an earlier change of the disposition of the signal to something other than its default.
- The signal is ignored. This is useful for a signal whose default action would be to terminate the process.
- A signal handler is executed.

What is a signal handler ? A signal handler is a function, written by the programmer, that performs appropriate tasks in response to the delivery of a signal. For example, the shell has a handler for the SIGINT signal (generated by the interrupt character, Control-C) that causes it to stop what it is currently doing and return control to the main input loop, so that the user is once more presented with the shell prompt. Notifying the kernel that a handler function should be invoked is usually referred to as installing or establishing a signal handler. When a signal handler is invoked in response to the delivery of a signal, we say that the signal has been handled or, synonymously, caught.

Signal Types and Default Actions

Earlier, we mentioned that the standard signals are numbered from 1 to 31 on Linux. However, the Linux signal(7) manual page lists more than 31 signal names. The excess names can be accounted for in a variety of ways. Some of the names are simply synonyms for other names, and are defined for source compatibility with other UNIX implementations. Other names are defined but unused. The following list describes the various signals:

SIGABRT	A process is sent this signal when it calls the abort() function. By default, this signal terminates the process with a core dump. This achieves the intended purpose of the abort() call: to produce a core dump for debugging.
SIGALRM	The kernel generates this signal upon the expiration of a real-time timer set by a call to alarm()or setitimer(). A real-time timer is one that counts according to wall clock time (i.e., the human notion of elapsed time).
SIGBUS	This signal (“bus error”) is generated to indicate certain kinds of memory access errors. One such error can occur when using memory mappings created with mmap(), if we attempt to access an address that lies beyond the end of the underlying memory-mapped file
SIGCHLD	This signal is sent (by the kernel) to a parent process when one of its children terminates (either by calling exit() or as a result of being killed by a signal). It may also be sent to a process when one of its children is stopped or resumed by a signal.
SIGCONT	When sent to a stopped process, this signal causes the process to resume (i.e., to be rescheduled to run at some later time). When received by a process that is not currently stopped, this signal is ignored by default. A process may catch this signal, so that it carries out some action when it resumes

SIGFPE	This signal is generated for certain types of arithmetic errors, such as divide-by-zero. The suffixFPE is an abbreviation for floating-point exception, although this signal can also be generated for integer arithmetic errors. The precise details of when this signal is generated depend on the hardware architecture and the settings of CPU control registers. For example, on x86-32, integer divide-by-zero always yields a SIGFPE,
SIGINT	When the user types the terminal interrupt character (usually Control-C), the terminal driver sends this signal to the foreground process group. The default action for this signal is to terminate the process.
SIGKILL	This is the sure kill signal. It can't be blocked, ignored, or caught by a handler, and thus always terminates a process.
SIGPIPE	This signal is generated when a process tries to write to a pipe, a FIFO, or a socket for which there is no corresponding reader process. This normally occurs because the reading process has closed its file descriptor for the IPC channel
SIGQUIT	When the user types the quit character (usually Control-\) on the keyboard, this signal is sent to the foreground process group. By default, this signal terminates a process and causes it to produce a core dump, which can then be used for debugging. Using SIGQUIT in this manner is useful with a program that is stuck in an infinite loop or is otherwise not responding. By typingControl-\ and then loading the resulting core dump with the gdb debugger and using thebacktrace command to obtain a stack trace, we can find out which part of the program code was executing.
SIGTERM	This is the standard signal used for terminating a process and is the default signal sent by thekill and killall commands. Users sometimes explicitly send the SIGKILL signal to a process usingkill –KILL or kill –9. However, this is generally a mistake. A well-designed application will have a handler for SIGTERM that causes the application to exit gracefully, cleaning up temporary files and releasing other resources beforehand. Killing a process with SIGKILL bypasses theSIGTERM handler. Thus, we should always first attempt to terminate a process using SIGTERM, and reserve SIGKILL as a last resort for killing runaway processes that don't respond toSIGTERM
SIGTSTP	This is the job-control stop signal, sent to stop the foreground process group when the user types the suspend character (usually Control-Z) on the keyboard.

The #define values are coded in <sys/signal.h> and some of the values are

```
#define SIGHUP  1       /* hangup */
#define SIGINT  2       /* interrupt */
#define SIGQUIT 3       /* quit */
#define SIGILL  4       /* illegal instruction */
#define SIGABRT 6       /* abort() */
#define SIGFPE  8       /* floating point exception */
#define SIGKILL 9       /* kill (cannot be caught or ignored) */
#define SIGBUS  10      /* bus error */
```

```
#define SIGSEGV 11      /* segmentation violation */
#define SIGSYS  12      /* bad argument to system call */
#define SIGPIPE 13      /* write on a pipe with no one to read it */
#define SIGALRM 14      /* alarm clock */
#define SIGTERM 15      /* software termination signal from kill */
#define SIGURG  16      /* urgent condition on IO channel */
#define SIGSTOP 17      /* sendable stop signal not from tty */
#define SIGCHLD 20      /* to parent on child stop or exit */
#define SIGWINCH 28     /* window size changes */
#define SIGUSR1 30      /* user defined signal 1 */
#define SIGUSR2 31      /* user defined signal 2 */
```

Implementation of Signal Handlers

A signal handler (also called a signal catcher) is a function that is called when a specified signal is delivered to a process. Invocation of a signal handler may interrupt the main program flow at any time; the kernel calls the handler on the process's behalf, and when the handler returns, execution of the program resumes at the point where the handler interrupted it. Although signal handlers can do virtually anything, they should, in general, be designed to be as simple as possible.

SIGINT

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

In the main function, we have registered our signal handler controlC with our operating system using the signal function. This main program will be in an infinite loop. In the signal handler, we are printing a statement and returning immediately. Generally, we don't print in the signal handlers, but for our purposes, we will use a printf statement. The parameter signum is not utilized though the OS will pass us the value.

Some key points to note from this program is that the second argument to signal() is a function pointer, a reference to a function to call. This tells the operating system that whenever this signal is sent to this process, run this function as the signal handler.

Also, the execution of the signal handler is asynchronous, which means the current state of the program will be paused while the signalhandler executes, and then execution will resume from the pause point, much like context switching.

```
void controlC ( int signum)
{
 printf ( "Thank you for pressing control-C\n");
}

 int  main ( )
{
 signal ( SIGINT, controlC)  ;  // registering our signal handler function with OS

  while ( 1 ) ;  // infinite loop
  exit ( 0 ) ;
}
```

Compile the above program and run. This program will be in an infinite loop. To see this signal in action, press control-c . The shell passes the control-C key stroke to the Operating system, OS will call the signal handler. Remember, before the function will be executed, our program is executing the while loop. The while loop will be stopped, the signal handler will be executing. When the handler finishes the printf statement, the control goes back to the while loop.

Now, if you prefer to exit the program in the handler, you can insert exit statement.

```
void controlC ( int signum)
{
 printf ( "Thank you for pressing control-C\n");
 exit ( 0 ) ;
}
```

SIGQUIT

```
void controlQuit ( int signum)
{
 printf ( "Thank you for pressing control-backslash \n");
 exit ( 0 ) ;
}
int main ( )
{
 signal( SIGQUIT, controlQuit);
 while ( 1 ) ;
 exit ( 0 ) ;
}
```

In the above program, we are registering a signal handler for SIGQUIT signal. The main program will be infinite loop. To generate the signal, press control \ (backslash) . You will see the program printing the statement and quitting the program.

SIGWINCH

As the name implies, this signal is sent whenever the size of the WINdow is CHanged.

```
void windowChange ( int signum)
{
  printf ( "Window size changed\n");
}
int main ( )
{
 signal( SIGWINCH, windowChange ) ;
 while ( 1 ) ;
 exit ( 0 ) ;
}
```

In the above program, we are registering a signal handler to receive signals whenever the size of the Window Changes. When you run the program, try to change the size of the window by resizing it. Note: We are exiting out of the handler in this program.

SIGCHD

```
void childExit ( int signum )
{
  printf ( "Child process Terminated \n");
  exit ( 0 ) ;
}
int main ( )
{
 signal ( SIGCHLD, childExit ) ;

 int child = fork () ;
 if ( child == 0 ) // child
   {
     sleep (2);  // when the child dies after 2 seconds
                 // parent will be notified.
   }
  else {
    while ( 1 ) ;
  }

 exit ( 0 ) ;
}
```

In the above program, we are registering a signal for the parent process to receive a signal when the child process terminates. In the main program, we are forking a child process. The child process sleep for 2 seconds and dies. When it dies, the parent process gets a signal. Just compile and run. In 2 seconds, you should see the statement in the handler function.

SIGSEGV

```
void sigsegv ( int signum )
{
   printf ( "illegal memory access \n");
   exit ( 1 );
}
int main ( )
{
  signal ( SIGSEGV, sigsegv ) ;
  int x = 10;
  scanf ( "%d", x); // it should &x

  exit ( 0 ) ;
}
```

In the above program, we are registering a signal handler to handle any illegal memory access. In the scanf function, we intentionally removed the address & for x and we are passing the address = 10. When you run program, the system call will place the value entered by the user in address 10 creating an illegal memory access. The kernel will generate a signal and our handler will print and exit.

SIGFPE

```
void sigfpe ( int signum)
{
   printf ( "Floating Point Exception \n");
   exit ( 1 );
}
int main ( )
{
  signal( SIGFPE, sigfpe);
  int x = 1/0;

  exit ( 0 ) ;
}
```

SIGFPE is a signal for floating point exception. When you divide a number by zero, floating point exception signal is generated.

SIGPIPE

```
void sigpipe ( int signum )
{
  printf ( "child may have died, and Pipe broken \n");
  exit ( 0 ) ;
}

int main(void)
{
        int     fd[2] ;
        pipe(fd);
        signal ( SIGPIPE, sigpipe);

    switch ( fork() )
    {
        case 0 : /* child */
                close(fd[1]);
                close(fd[0]);  /* closing read end of pipe */
                exit(0);

        default: /* parent */
                sleep ( 1);
                close(fd[0]);
                write(fd[1], "ABCD\n", 5);
    }
        return(0);
}
```

In the above program, we create a pipe and we close the pipes on the child process. When the parent writes on the pipe, because the child is already terminated, a SIGPIPE signal is generated. You need sleep on the parent process to give enough time for child to terminate .

SIGSTOP The SIGSTOP signal stops the process. It cannot be handled, ignored, or blocked.

The SIGKILL signal is used to cause immediate program termination. It cannot be handled or ignored, and is therefore always fatal. It is also not possible to block this signal.

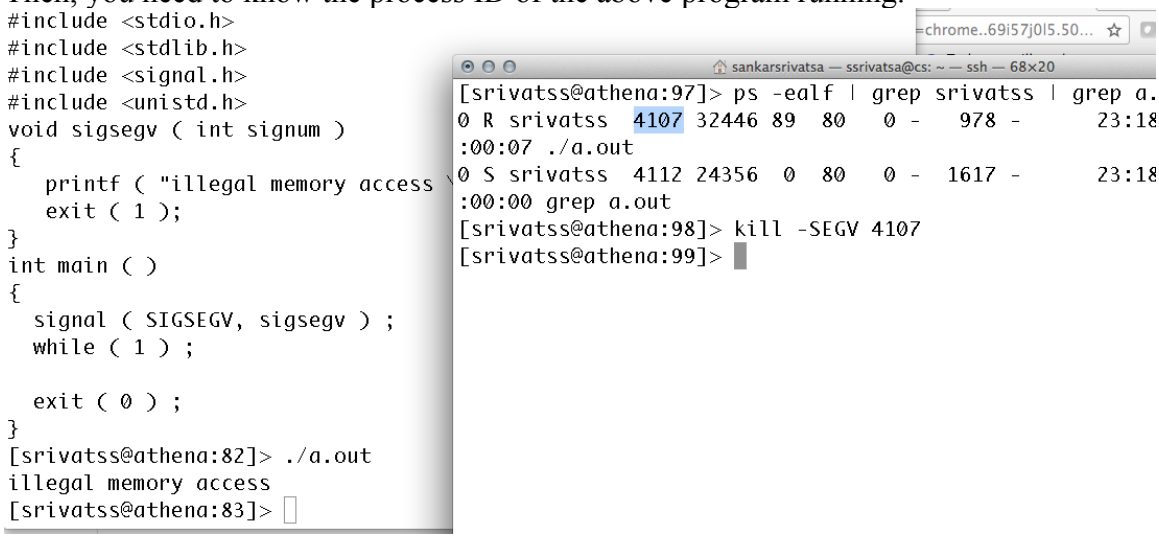
GENERATING SIGNALS FROM COMMAND LINE

So far we have seen signals were triggered in our program. Now we will try to trigger from command line of another shell .

In one shell window, try to run this program from the command line.

```
void sigsegv ( int signum )
{
  printf ( "illegal memory access \n");
  exit ( 1 );
}
int main ( )
{
 signal ( SIGSEGV, sigsegv ) ;
 while ( 1 ) ;

 exit ( 0 ) ;
}
```

Then, you need to know the process ID of the above program running.



In the above diagram, the process ID for the program 4835 .

In another shell terminal, send the signal to the process running 4835 using the command

```
kill -SEGV 4835
or
kill -11 4835
```

The 11 above is the constant value defined for SIGSEGV in the <sys/signal.h>

You can also use killall command , instead of kill, but use the name of the program.

```
killall -SIGKILL a.out
```

Alarms :

A `SIGALRM` signal is an alarm set by the function alarm (int sec) after sec seconds have elapsed since calling the function, it is delivered by the Operating System much like signals we read so far.

```
void buzzMe( int signum )
{
   printf ( "Wake up...\n");
}

int main()
{
  signal ( SIGALRM, buzzMe);
  alarm ( 2 );
  while ( 1  ) ;
}
```

The above looks very similar to the program we wrote earlier. But here, we use SIGALRM signal value to register our alarm with the operating system and the alarm handler is defined. Then, we have to request an alarm , here we use 2 seconds. After the request has been put, we wait using a while loop. You can instead use a function pause or sleep , but these are implementation specific , in general avoid using signals and sleep.

Because only one alarm can used in a program, we cannot set another alarm. In case you plan to use another alarm, the previous alarm will be replaced.