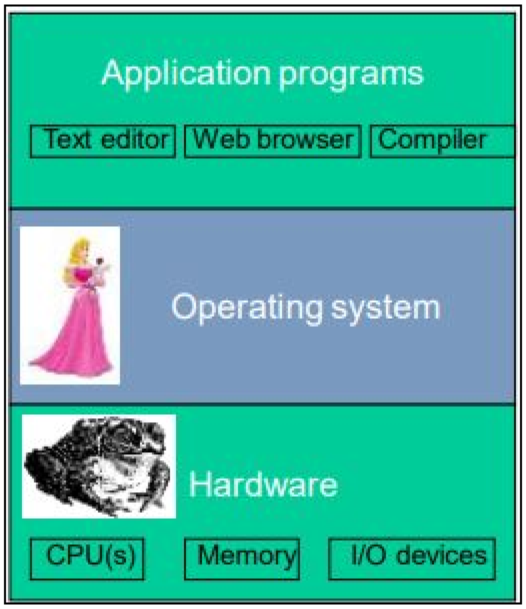
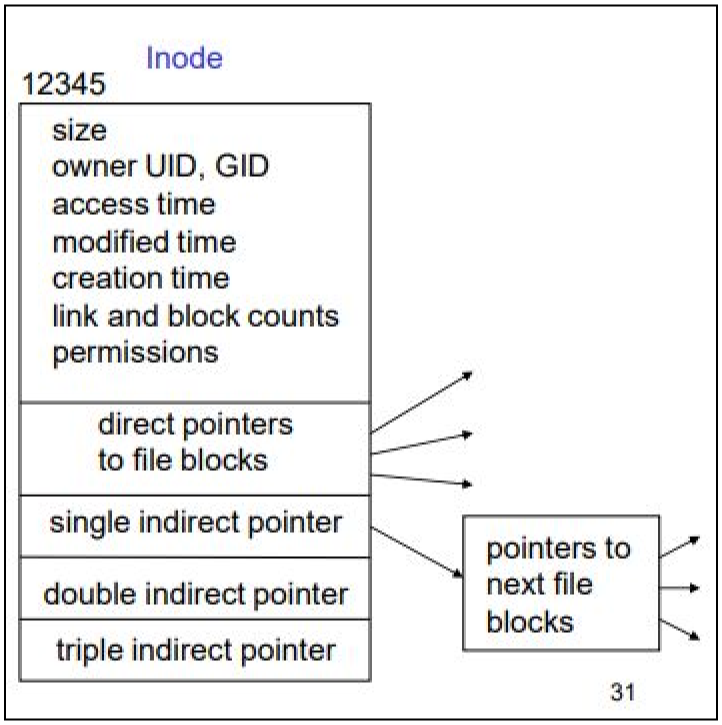
Week 1 Notes

1. **<u>Operating Systems (OS):</u>**
    - An **operating system** is the software layer between user applications and hardware.
    - It serves as a resource manager.
      I.e. It allows the proper use of resources like hardware, software and data.
    - It also serves as a control program (protection).
      I.e. It controls execution of user programs to prevent errors and improper use of the computer.
    - Turns ugly hardware into beautiful abstractions (file and directories).



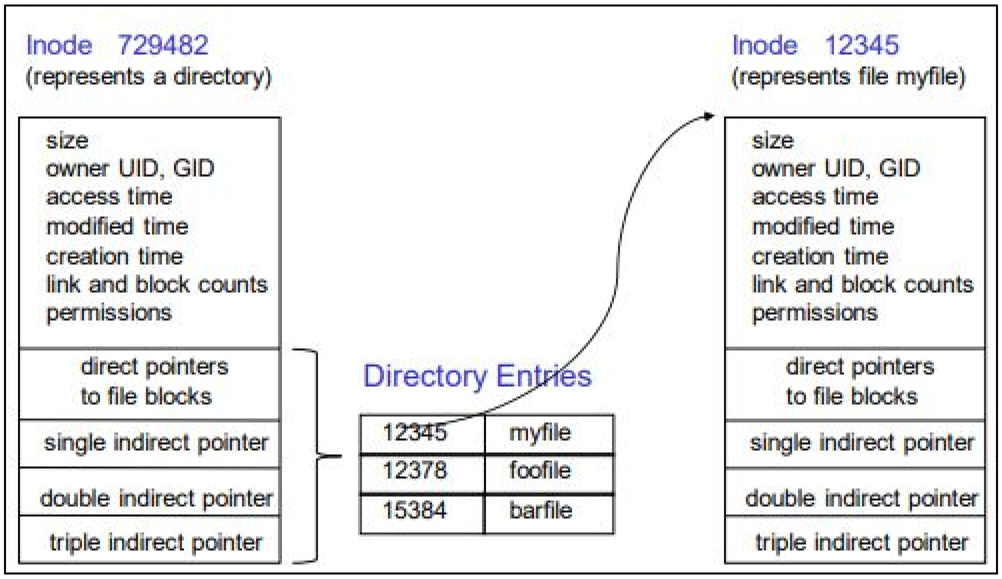
    - You can use Unix commands like cd, mv, cat, ls, etc to navigate directories and files.
    - You can use the man command to see what a command does.
2. **<u>Files:</u>**
    - A **file** is a name collection of data with some attributes:
        - Name
        - Owner (User and Group)
        - Size
        - Permissions
        - Time of creation
        - Last Access
        - Last Modification
        - Location on disk
    - To get info on a file in Unix, we can use:
        1. ls -l
        2. stat

- A data structure called **inode** stores the info of a file, including disk blocks which contain the file's data.



- A file is identified by its **inode number**.

3. **Directories:**
    - A **directory** is a collection of files and sub-directories.
    - In Unix, every directory is a file.
    - The **root**, denoted by /, is a special directory.
    - A directory entry maps a file name to an inode.



4. **Directory Hierarchy:**
   - The directory hierarchy is an **acyclic graph** because of **links**.
5. **Links:**
   - Sharing of files can be implemented by creating a new directory entry called a **link**, which is a **pointer** to another file or directory.
   - There are 2 types of links:
     1. **Hard Link:**
        - Created with ln <target> <name of link>
        - The second directory entry is identical to the first and shares the same inode number. Furthermore, both are files.
        - If the first directory entry is deleted, the second one is still there.
     2. **Soft Link:**
        - Created with ln -s <target> <name of link>
        - The second directory entry points to a small file containing the path of the first directory entry.
        - The second directory entry is a link, and has a different inode number than the first.
        - If the first directory entry is deleted, then we have a **dangling reference**.
6. **Permissions:**
   - **File permissions** have the follow setup: -----------
   - After the first dash, the next 3 dashes are for the owner. Then, the following 3 dashes are for the groups. Lastly, the final 3 dashes are for others.
     I.e. -(---)(---)(---)
     The red dashes are for the owner.
     The blue dashes are for the groups.
     The orange dashes are for others.
   - **Directory permissions** have the follow setup: d----------
   - After the d, the first 3 dashes are for the owner. Then, the following 3 dashes are for the groups. Lastly, the final 3 dashes are for others.
     I.e. d(---)(---)(---)
     The red dashes are for the owner.
     The blue dashes are for the groups.
     The orange dashes are for others.
   - Each entries specify 3 permissions, **read**, **write** and **execute**.

	read	write	execute
Denoted by	r	w	x
File Permission	Gives authority to open and read a file.	Gives authority to modify the contents of a file.	Gives authority to execute a file.
Directory Permission	Gives authority to run ls on a directory.	Gives authority to add, remove and rename files in a directory.	Gives authority to cd into a directory.

- Example:
  drw-rw-r-- means that:
    - The owner has read and write permissions, but not execute permissions on the directory
    - The group has read and write permissions, but not execute permissions on the directory.
    - Others have read permissions on the directory.
- You can use the command chmod to change permissions.
- Examples:
    1. chmod u+x fname will give the user who owns fname execute permissions.
    2. chmod g+r fname will give all users in group read permissions.
    3. chmod a+rwx will give all users all permissions.

7. **<u>Shell:</u>**
    - A **shell** is a commandline interpreter.
    - It is the interface between the user and an OS.
    - The shell is a program that:
        1. Waits for input commands.
        2. Analyzes commands.
        3. Determines what actions are to be performed.
        4. Performs the actions.
    - Shells can execute all the Unix commands, do I/O redirection, pipelining of commands, filtering output of commands, job control, shell programming and more.
8. **<u>Input and Output Redirection:</u>**
    - Programs read from standard input (keyboard), write the results on standard output (screen) and write errors to standard error (screen).
    - You can redirect input, output and errors using the following:
        1. > filename redirects the output to the file.
           I.e. It replaces the file's original text with the output.
        2. >> filename appends the output to the file.

    3. < input file redirects input
- **Standard output (stdout)** is denoted by 1.
- **Standard error (stderr)** is denoted by 2.
- Examples
    1. ls > output.txt will overwrite the contents of output.txt with the output of ls.
    2. ls >> output.txt will append the output of ls to the contents of output.txt.
    3. ls -z 2>output.txt will overwrite the contents of output.txt with an error message, because ls -z is not a valid command.

9. **Pipelines:**
- Use | to send the output of one command to the input of another command.

10. **Filters:**
- A **filter** reads from standard input, processes the input and writes to standard output.
- Some useful filters
    - wc: count words, lines, characters
    - grep: filter lines that do or do not match a pattern
    - uniq: remove repeated lines
    - sort: sorts input
    - head: output only the first lines of the provided input
    - tail: output only the last lines of the provided input
    - sed: a stream editor to perform text transformations

11. **Job Control:**
- A **job/process** is a program in execution. Use ps to view processes.
- **Foreground job** has control of the terminal.
- **Background job** runs concurrently with the shell in the background.
- To run a program in the background, append & to the name of the program.
- At any point, a program can be suspended. Hit <ctrl> z to suspend the current foreground job.
- The command jobs gives you a list of jobs and each is associated with a number.
    - fg [num] puts job num in the foreground.
    - bg [num] puts job num in the background.
    - kill %num kills job num.

CSCB09 Week 2 Notes

1. **Pipe:**
   - Denoted by "|".
   - Allows the user to run 2 or more commands consecutively.
   - The syntax is: command1|command2|...|commandn. Note that data flows left to right. This means that the output of command1 is the input for command2, and so on.
2. **Filter:**
   - A filter is an Unix command that reads from standard input, processes the input and writes on standard output.
     I.e. It takes some input, processes the input and produces some output based on the input.
   - Some Unix filter commands are:
     a. Grep
     b. Sort
     c. Uniq
     d. Cat
     e. More
     f. Head
     g. Tail
     h. Wc
3. **Shell Scripting:**
   - We use shell scripting because it helps us automate things.

   1) **File Expansion:**
      - ***** means 0 or more characters.
      - **?** means exactly 1 character.
      - **[x-y]** means one character in the range x to y, inclusive.
      - **[^oa]** means any character except o or a.
      - **~** means home directory.
      - **~u** means home directory of user u.
      - E.g.
        a. ls *.txt will list out all the text files in that directory.
        b. rm * will remove everything in that directory.
        c. cp ?? ~ will copy everything with 2 characters in that directory and put it into the home directory.
   2) **Creating a shell script:**
      a. Create a file using any editor and name the file with the extension **.sh**.
      b. Start the script with **#!/bin/bash**, or any of its variant. This is called the **shebang**, and is the path to your bash interpreter. It tells the shell how to interpret/execute the commands in this file.
      Note: The shebang is not the same on every machine. Use the which command to find the right one for your computer.
      c. Write some code. You can put any Unix commands in the script file.
      d. Save the file as filename.sh and execute the script using
         i. bash filename.sh

ii. ./filename.sh ← No Spaces!!!

Note: The file must be executable. Use the chmod command to change execute permission, if needed.

3) Variables and Comments:

a) A comment is denoted by #.

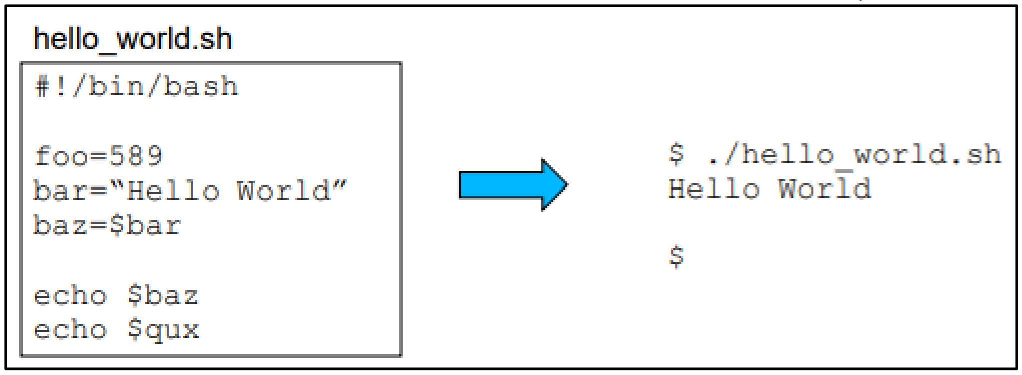
b) In any Unix system, including shell, there are 2 types of variables:

i) System Variables:

- These are the built-in variables.
- One of the most useful built-in variables is PATH.

ii) User-defined Variables:

- To assign a variable to a value, you do **var=value**.
  Note: There are no spaces before or after the equal sign.
- To get the value of a variable, you do **$var**.
- Variables are not declared, just assign a value.
- Variables have no type they can hold any type of data.
- If you access a variable without a value, you will get an empty string instead of an error.
- Variable names can only start with a letter or an underscore sign. However, after the first letter, variable names can have numbers, too.

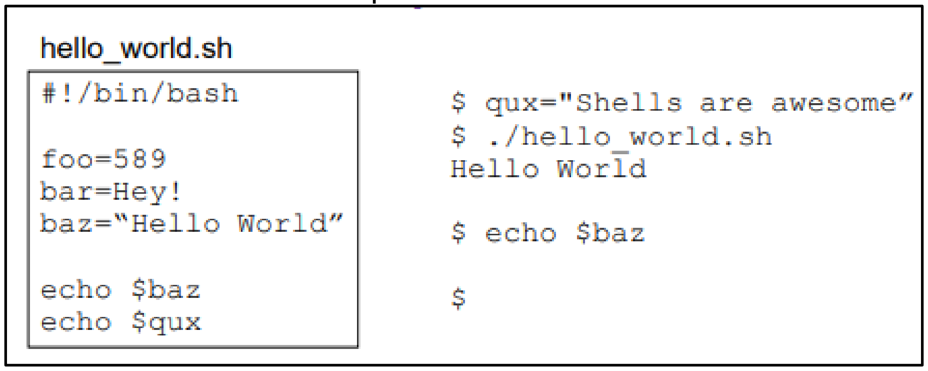


hello_world.sh

```
#!/bin/bash

foo=589
bar="Hello World"
baz=$bar

echo $baz
echo $qux
```

```
$ ./hello_world.sh
Hello World

$
```

- Variables defined in a script are lost when the script ends, unless you use the source command to run the script.

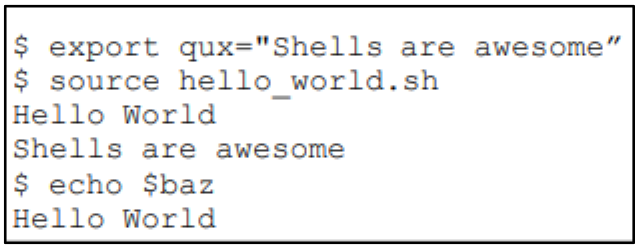


hello_world.sh

```
#!/bin/bash

foo=589
bar=Hey!
baz="Hello World"

echo $baz
echo $qux
```

```
$ qux="Shells are awesome"
$ ./hello_world.sh
Hello World

$ echo $baz

$
```

- The subshell does not have access to the variables of the parent shell, unless you export the variable.



```
$ export qux="Shells are awesome"
$ source hello_world.sh
Hello World
Shells are awesome
$ echo $baz
Hello World
```

4) <u>Quotes in Shell:</u>

- Back quotes `` : Anything in between back quotes would be treated as a command and would be executed.
  I.e. Back quotes does command substitution.
- Single quotes ‘’ : All special characters between these quotes lose their special meaning.
  I.e. Single quotes force shell to take string literally.
- Double quotes “” :Expand variables and do command substitution.
  Most special characters between these quotes lose their special meaning with these exceptions:
  i) $
  ii) `
  iii) \$
  iv) \'
  v) \"
  vi) \\
- E.g. Suppose we have 3 directories, a, b and c in our current working directory.
  $ echo * will print <u>a b c</u>
  $ echo ls * will print <u>ls a b c</u>
  $ echo `ls *` will print <u>a b c</u>
  $ echo “ls *” will print <u>ls *</u>
  $ echo ‘ls *’ will print <u>ls *</u>
  $ echo `*` will create an error.
- E.g. Consider the shell script below:
  #!/bin/sh
  $ date
  $ echo Today is `date`
  $ echo "Today is `date`"
  $ echo ’Today is `date`’
  The output is:
  <u>Thu Jan 17 10:58:13 STD 2019</u>
  <u>Today is Thu Jan 17 10:58:13 STD 2019</u>
  <u>Today is Thu Jan 17 10:58:13 STD 2019</u>
  <u>Today is `date`</u>

5) **<u>Read:</u>**
   - Read one line from standard input and assigns successive words to the specified variables.
   - Leftover words are assigned to the last variable.
   - E.g. Suppose we have this shell script:

```
#!/bin/sh
echo "Enter your name:"
read firstName lastName
echo $firstName
echo $lastName
```

     If the user runs the script and enters Alexander Graham Bell for read, then the output is:

```
Alexander
Graham Bell
```

6) **<u>Commandline Arguments:</u>**
   - Commandline arguments are placed in positional parameters. The commandline arguments are denoted by $1, $2, … . After $9, we use ${10}.
   - Command line arguments allows the users to either control the flow of the command or to specify the input data for the command.
   - Positional parameters in a shell script are the command line arguments passed to a shell script.
   - $0 is the name of the script
   - $# is the number of commandline arguments
   - $* and $@ list all commandline arguments
   - E.g. Consider the shell script below:

```
#!/bin/sh
echo firstname: $1
echo lastname: $2
echo Firstname is $1 and lastname is $2
echo $#
echo $@
echo $*
```

     If I run the script and type, **<u>$ bash test.sh rick lan</u>**, the output is:

```
firstname: rick
lastname: lan
Firstname is rick and lastname is lan
2
rick lan
rick lan
```

7) **<u>Positional parameters and set and shift:</u>**
   - The set command assigns its parameters to the positional parameters and all previous positional parameters are thrown away.

- E.g.
  $ set pizza spaghetti rice
  $ echo $1 $2 $3 will prints pizza spaghetti rice
  The set command stored pizza in $1, spaghetti in $2 and rice in $3.
- Shift moves all positional parameters to the left.
  I.e. $1 becomes the old $2, etc.
- The syntax for the shift command is **shift number**, which will shift the positional parameters that many times to the left.
  E.g. shift 1 will shift the positional parameters one to the left.
  E.g. shift 2 will shift the positional parameters two to the left.
  E.g. shift 0 will not do anything.
- The shift command is useful for iterating over all positional parameters.
- E.g. Consider the shell script below:

```
#!/bin/sh
set x y z
shift 1
echo $@
```

  This will print y z
- E.g. Consider the shell script below:

```
#!/bin/sh
set x y z
shift 0
echo $@
```

  This will print x y z

8) if statement and test command:

- The test command takes an expression and returns 0 if its true and 1 if its false.
- E.g.

```
if test $str1 = $str2;
then
        echo "The strings are identical"
else
        echo "The strings are different"
fi
```

- A short form of test is [ ].
- E.g.
  if [$str1 = $str2] ← The spaces are mandatory.
  then …

Test commands for String	**Description**
-z string	True if empty string
str1 = str2	True if str1 equals str2
str1 != str2	True if str1 not equal to str2
Test commands for Integer	**Description**
int1 -eq int2	True if int1 equals int2
int1 -ge int2	True if int1 >= int2
int1 -gt int2	True if int1 > int2
int1 -lt int2	True if int1 < int2
int1 -le int2	True if int1 <= int2
-a	and
-o	or
Test commands for Files and Directories	**Description**
-d filename	Exists as a directory
-f filename	Exists as a regular file
-r filename	Exists as a readable file
-w filename	Exists as a writable file
-x filename	Exists as an executable file

- The syntax for the if statement is:
```
if [ condition1 ]
then
    Statement(s) to be executed if condition1 is true
elif [ condition2 ]
then
   Statement(s) to be executed if condition2 is true
else
   Statement(s) to be executed if no condition is true
fi
```
- In bash the if statement checks the return value of a condition and proceeds to “then” if the return value is 0, to “elif” if the return value is not 0, and to “else” if none of the return values are 0.

- Unix requires that programs return 0 for success and some other number for failure. You can use $? to check for the return value.
- E.g. Consider the shell script below:

  ```
  #!/bin/sh
  a=10
  b=20
  if [ a != b ] ← The spaces are mandatory.
  then
          echo "Correct: $?"
  else
          echo "False: $?"
  fi
  ```

  The output of this is: Correct: 0
- E.g. Consider the shell script below:

  ```
  #!/bin/sh
  a=10
  b=20
  if [ a == b ] ← The spaces are mandatory.
  then
          echo "Correct: $?"
  else
          echo "False: $?"
  fi
  ```

  The output of this is: False: 1

9) expr:

- Since shell scripts work by text replacement, we need a special function for arithmetic.
- expr works only for integer arithmetic.
- E.g. Consider the shell script below:

  ```
  #!/bin/sh
  x=1+3
  y=`expr 1 + 3` ← The spaces are mandatory.
  z=`expr 1 \* 5` ← Need to escape the * sign first.
  echo $x
  echo $y
  echo $z
  ```

  The output is:

  ```
  1+3
  4
  5
  ```

10)<u>**while loop:**</u>

- The syntax is:
```
while [ condition ]
do
        some code
done
```

- E.g. Consider the shell script below:
```
#!/bin/sh
x=10

while [ $x -gt 0 ]
do
      echo $x
      x=`expr $x - 1`
done
```
The output is:
```
10
9
8
7
6
5
4
3
2
1
```
- We can also use the while loop to read from a file, one line at a time.
- E.g.
```
#!/bin/bash
file="my_file.txt"
while read line
do
        echo $line
done < $file
```
- We can also use the while loop to iterate over arguments.
- E.g.
```
#!/bin/sh
while test "$1" != ""
do
        echo $1
        shift
done
```
- Don’t use this one unless you know that the argument list will always be short.

11) **for loop:**

- The general syntax is:

```
for var in something
do
	some code
done
```

- E.g.

  a.

```bash
#!/bin/bash
for i in 1 2 3 4 5
do
	echo $i
done
```

  b.

```bash
#!/bin/bash
for i in `ls`
do
	mv $i $i.txt
done
```

  This appends all filenames in the current directory with the extension .txt. A file called x would now be called x.txt.

  c.

```bash
#!/bin/bash
for i in $*
do
	mv $i $i.txt
done
```

  This appends all files whose names are given to the script as command-line arguments with the extension .txt.

Week 3 Notes

1. **<u>Functions in Shell Scripts:</u>**
    - Also called subroutines.
    - Arguments are passed through positional parameters.
    - Variables defined outside the function are visible within as long as they are declared before the function calling.
    - Variables defined inside the function are visible outside.
    - Return value is stored in “$?”.
    - Must return a number.
    - E.g. Consider the shell script below:
      ```
      #!/bin/sh
      myfunc() {
              arg1=$1
              arg2=$2
              echo $globalvar $arg1
              return 100
      }
      globalvar="I like to eat”
      myfunc pizza spaghetti
      echo $?
      ```
      The output is:
      <u>I like to eat pizza.</u>
      <u>100</u>
    - E.g. Consider the shell script below:
      ```
      #!/bin/sh
      myfunc() {
              arg1=$1
              arg2=$2
              echo $globalvar $arg1
              return 100
      }
      globalvar="I like to eat”
      myfunc pizza spaghetti
      echo $arg2
      echo $?
      ```
      The output is:
      <u>I like to eat pizza.</u>
      <u>spaghetti</u>
      <u>0</u>

    - E.g. Consider the shell script below:
      ```
      #!/bin/bash
      func(){
      var1=3
      echo $var2
      return 100
      ```

```
}
func ← Function calling
var2=5
echo $var1
```
The output is:

```
3
```

- E.g. Consider the shell script below:

```
#!/bin/bash
func(){
var1=3
echo $var2
return 100
}
var2=5
func ← Function calling
echo $var1
```
The output is:
```
5
3
```

2. **Introduction to C:**
   - C is a low-level language: machine access
   - C is a high-level language: structured
   - C is a small language, extendable with libraries
   - C is permissive: assumes you know what you're doing
   - Good: efficient, powerful, portable, flexible
   - Bad: easy to make errors
3. **Basic Data Types:**
   - char: Integer; 8 bits **Note:** 8 bits = 1 byte
   - int: Integer; 32 bits
   - long: Integer; 64 bits
   - float: Real number; 32 bits
   - double: Real number; 64 bits
   - In an assignment statement make sure that the variable on the left is at least as wide as the expression on the right.
   - **Note:** In C, there is no string type. We have to use char arrays.
4. **Variables:**
   - Variable names cannot start with a number or an underscore.
   - Variables need to be declared.
   - Variables have no default value. If a variable is not initialized, there's no guarantees about what its content is.
   - One function's variables are not visible to other functions.
     (I.e. they are local)
   - Variables are only visible inside the block they were declared in, with the exception of global variables declared outside of the main function.

- Examples of variable declaration:
    - int a;
    - int a = 10;
    - char a, b, c;
    - char a = “a”, b = “b”, c = “c”;
- The type of a variable must be a valid C type.

5. **If statement:**
    - You must have an if, but the else if and else are optional. Furthermore, you can have as many else if’s and at most, one else. The if is followed by the else if’s, if there are any, followed by an else, if there is any. Lastly, once a condition is met, the rest of the else if’s and else won’t be tested.
    - If there is any variables created in the if, else if, else block, then once you exit the block, the variable is no longer accessible.
    - We can’t mix declarations and non-declaration in a block.
    - Syntax:

```
if ( condition 1 ){
        some code …
}
else if ( condition 2 ){
        some code …
}
else{
        some code …
}
```

- Example:

```
int a = 10;
if ( a < 20 ){
        printf(“a is less than 20.”);
}
else if ( a == 20){
        printf(“a is equal to 20.”);
}
else{
        printf(“a is greater than 20.”);
}
```

6. **While loop:**
    - If there is any variables created in the while loop, then once you exit the loop, the variable is no longer accessible.
    - We can’t mix declarations and non-declaration in a block.
    - Syntax:

```
while ( condition ){
        some code ...
}
```

- Example:
```
int a = 10;
while (a < 20){
        printf("%d", a);
        a++;
}
```

7. **For loop:**
   - If there is any variables created in the for loop, then once you exit the loop, the variable is no longer accessible.
   - Syntax:
     ```
     for ( init; condition; increment ) {
       statement(s);
     }
     ```
     Note: In older versions of C, you cannot have the initialization in the for loop. I.e. You can’t do for (int i = 1; …)
     - Here is the flow of control in a 'for' loop:
       1. The init step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
       2. Next, the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the 'for' loop.
       3. After the body of the 'for' loop executes, the flow of control jumps back up to the increment statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
       4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the 'for' loop terminates.
   - Example:
     ```
     for (int i = 1; i < 20; i++){
             printf("%d \n", i);
     }
     ```
8. **Functions:**
   - Every C program has the main function, main().
   - A **function declaration** tells the compiler about a function's name, return type, and parameters.
     The syntax is **return_type function_name(parameter list);**
   - A **function definition** provides the actual body of the function.
   - Functions must be declared before first use then defined.

- **return_type function_name (parameters) {**
  some code …
  }
  Return Type: A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value.

  Function Name: This is the actual name of the function. The function name and the parameter list together constitute the function signature.

  Parameters: A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

  Function Body: The function body contains a collection of statements that define what the function does.

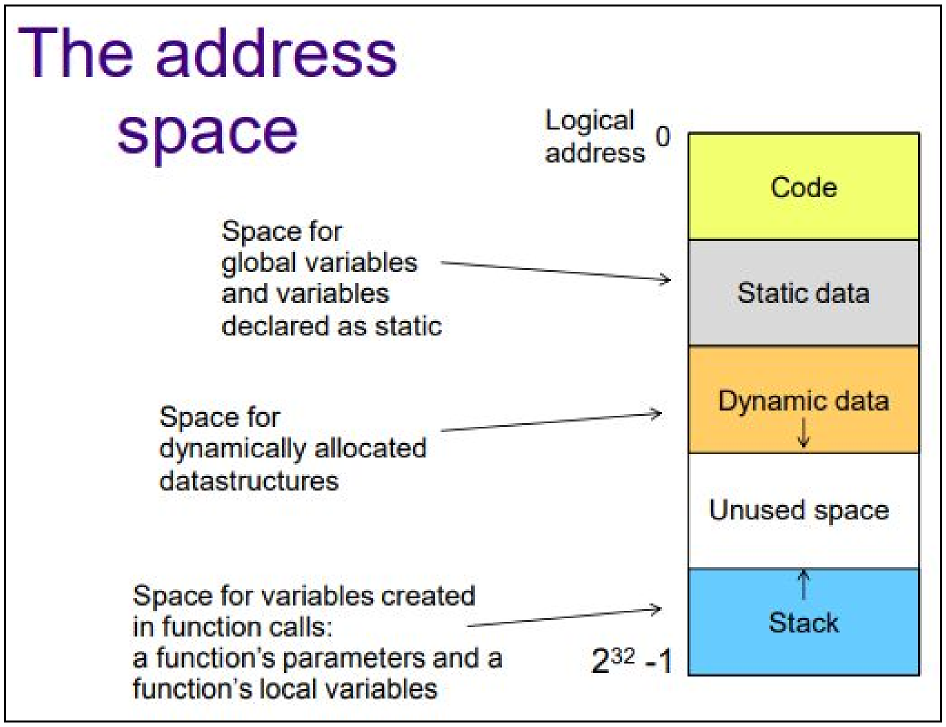
- If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function. Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.
- While calling a function, there are two ways in which arguments can be passed to a function:
  1. **Call by value:** This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
  2. **Call by reference:** This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

     By default, C uses call by value to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

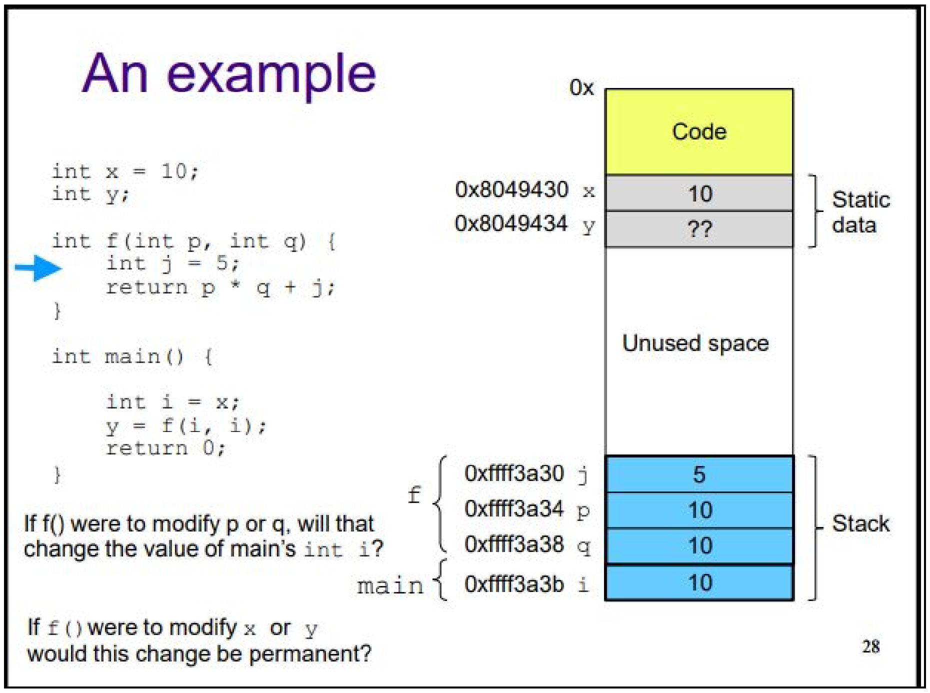
- Example:

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

int min(int num1, int num2); /* Method declaration */

int main (){
   int a = 0;
   int b = 100;
   int result;
   result = min(a, b);
   printf("The minimum value is %d", result);
   return 0;
}

/* Method Definition */
int min (int num1, int num2){

   int result;

   if (num1 < num2){
     result = num1;
   }

   else {
     result = num2;
   }

   return result;
}
```

9. **<u>Memory Model:</u>**
    - The memory for a **process** (a running program) is called it's **address space**.
    - Memory is just a sequence of bytes.
    - A memory location (a byte) is identified by an address.


The address space
Logical address 0
Code
Space for global variables and variables declared as static
Static data
Space for dynamically allocated datastructures
Dynamic data
Unused space
Space for variables created in function calls: a function's parameters and a function's local variables
Stack
$2^{32}-1$

    - E.g.



An example

```
int x = 10;
int y;

int f(int p, int q) {
    int j = 5;
    return p * q + j;
}

int main() {

    int i = x;
    y = f(i, i);
    return 0;
}
```

If f() were to modify p or q, will that change the value of main's `int i`?

If `f()` were to modify `x` or `y` would this change be permanent?

The answer to both questions is no.

Week 3 Notes

10. Arrays:

- Arrays in C are a contiguous chunk of memory that contain a list of items of the same type.
- In particular, the size of the array is not stored with the array. There is no runtime checking.
- To declare a single-dimensional array, you do:
  type arrayName [arraySize]
  type can be any valid C type.
  arraySize must be an integer constant greater than zero.
  E.g.
  int a[10]; This creates an array of size 10 that holds ints.
- You can initialize an array by any of the following methods:
  1. **Initialization Loops:**
     ```
     for (int i = 0; i < N; i++){
             a[i] = i;
     }
     ```
  2. **Static Initialization:**
     int a[5] = {1,2,3,4,5};
     Note: The number of values in the curly brackets cannot exceed the size of the array.
     int a[] = {1,2,3,4,5}; also works.
- **Note:** The first element of an array is its 0th index.
  I.e. a[0] will get the first element of a.
- You can set or change values for a specific element, too.
  E.g.
  int a[4] = 50; will set 50 as the 5th element of array a.
- An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array.
  E.g.
  int x = a[9]; This will set x to a's 10th element.
- There are multi-dimensional arrays, too.
- The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size [x][y], you do type arrayName [x][y]. Here, x is the number of rows and y is the number of columns.
  E.g.
  int a[3][4]; This creates a 2-D array with 3 rows and 4 columns.

Week 3 Notes

- You can initialize an array by any of the following methods:
    1. <u>**Initialization Loops:**</u>
```
for (int i = 0; i < N; i++){
        for (int j = 0; j < M; j++){
                a[i][j] = i*j;
        }
}
```
    2. <u>**Static Initialization:**</u>
```
int a[3][4] = {
                {0, 1, 2, 3} ,
                {4, 5, 6, 7} ,
                {8, 9, 10, 11}
               };
```
       int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11}; also works.
- To access an element of a 2-D array, you do
  int x = a[i][j];

11. <u>**Strings:**</u>
- Strings are actually one-dimensional array of characters terminated by a null character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a null.
- Since there's the null character at the end, the array's size is one bigger than the length of the word. If a word has 3 letters, the size of the array is 4.
- E.g.
  char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
  char greeting[] = "Hello";
  Both ways are valid.

12. <u>**Pointers:**</u>
- A pointer is a higher-level version of an address.
  I.e. A pointer is a variable whose value is the address of another variable.
- The general form of a pointer declaration is: **type *var;**
  E.g.
  char *cptr;
- Assigning a value to a pointer:
  char c = 'a';
  cptr = &c;
    - cptr gets the value of the address of c.
    - The value stored in the variable cptr is the address of the memory location where variable c is stored.
- Dereferencing a pointer:
  ***cptr = 'b';**
    - Stores 'b' at the memory address that is stored in cptr.

  char d = *cptr;
    - Takes the contents of the memory address stored in cptr and stores them in the variable d.

Week 3 Notes

- Without an asterisk, pointer references an address.
- With an asterisk, pointer references the value at that address.
- Always use the same type of pointer as the variable it examines.
  (I.e. ints for ints, chars for chars)
- Initialize the pointer before using it. Set it to an address of some variable.
- When you declare an array, you are creating a pointer to the array's 0th element. (In C, pointers and arrays are very similar.)
  E.g.
  int a[3] = {1, 3, 5};
  int *p = a; // same as p = &a[0];
  Note: When pointing to an array, you do not need to use &.
- The array access operator [ ] is really only a shorthand for pointer arithmetic + dereference.
- These are equivalent in C: a[i] == *(a + i)
  E.g.
  a[0] == *p
  a[1] == *(p+1)
  a[2] == *(p+2)
- Pointer arithmetic respects the type of the pointer.
  E.g.
  int i[2] = {1, 2};
  int *ip;
  ip = i;
  ***(ip + 1) += 2; // This puts 4 in i[1].**
  E.g.
  char c[2] = {'a','z'};
  char *cp;
  cp = c;
  ***(cp + 1) = 'b'; // This puts b to c[1].**
- C knows the size of what is being pointed at from the type of the pointer.
- When you pass arrays as parameters, what is being passed to the function is the name of the array which decays to a pointer to the first element. Since C doesn't store the size of arrays, you should also input the size of the array.
- int sum(int *a, int size) and int sum(int a[], int size) are both legal, but it's bad form to use the latter. This is because you really are passing a pointer-to-int not an array, you still don't know how big the array is.
- **Note:** Use array notation rather than pointer arithmetic whenever you have an array.

Week 3 Notes

13. **Scope Rules:**
    - There are three places where variables can be declared in C programming language:
        - Inside a function or a block which is called **local variables**.
        - Outside of all functions which is called **global variables**.
        - In the definition of function parameters which are called **formal parameters**.
    - **Local Variables**: Variables that are declared inside a function or block. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own.
    - **Global Variables**: Variables that are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program. A global variable can be accessed by any function.
      I.e. A global variable is available for use throughout your entire program after its declaration.
    - **Formal Parameters**: Formal parameters, are treated as local variables with-in a function and they take precedence over global variables.
14. **Special Characters and Miscellaneous Information:**
    - **#include <stdio.h>**: Include this at the top of your program. It allows you to access some of C's built in functions and libraries.
      Note: There are other, similar tags.
    - **printf**: If you have #include <stdio.h>, then this is one of the built in features you can use. It allows you to print things on the console.
    - **%d:** Take the next argument and print it as an int.
    - **%s:** Take the next argument and print it as a string.
    - **\n:** Newline
    - **/* */:** Comments
    - **&**: Allows the user to access an item's memory location.
    - Because C is permissive, the error messages are often random and/or weird.
    - In C, 0 is false and 1 is true.
    - To compile a C program, do the following:
      gcc -Wall -o programName programName.c
      gcc is the compiler.
      -Wall may help create some meaningful error messages.
      -o programName creates the object file.

CSCB09 Week 4 Notes

1. **<u>Strings:</u>**
   - Strings are not a built-in data type.
   - A string is an array of characters terminated with a null character ('\0'). Therefore, the size of the array must be 1 larger than the length of the string.
   - Initializing a string (Either way works):
     - **char course_name[8] = {'c','s','c','b','0','9','h','\0'};**
     - **char course_name[8] = "cscb09h";**
   - C has no built-in support, but many string functions are provided in a library. We can use **#include<string.h>** to access the built in functions.
   - <u>Common string operations:</u>
     - <u>Length of a string:</u>
       - The length is the number of non-null characters.
       - E.g. Suppose we have
         char course_name[50] = "cscb09h";
         The length of the string is 7, because there are 7 non-null characters.
       - <u>Note:</u> The length of a string is not the same as the size of the array. In the example above, the size of the array is 50, while the length of the string is 7.
       - Library function that returns length of a string:
         strlen (const char *str)
       - Example of strlen:

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

int main (){
   int len;
   char x[8] = "cscb09";
   len = strlen(x);
   printf("Length of %s is %d\n", x, len);
   // The output is "Length of cscb09 is 6"
   return 0;
}
```

     - <u>Copying a string:</u>
       - Two library functions:
         1. **char *strcpy (char *dest, char * src)**
         2. **char *strncpy(char *dest, const char *src, int n)**
            Like strcpy, but copies at most n characters from src
       - Example of strcpy and strncpy.

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

```
int main (){
   char src[50], dest[50];
   strcpy(src,  "This is source");
   strncpy(dest, "This is destination", 10);
   printf("This is src : %s\n", src);
   // Prints "This is src : This is source"
   printf("This is dest : %s\n", dest);
   // Prints "This is dest : This is de"
   return 0;
}
```

- <u>Concatenating two strings:</u>
    - **char *strcat (char *dest, const char *src)**
      Appends src to dest (including the null byte of src), overwriting the null byte of dest.
    - **char *strncat(char *dest, const char *src, int n)**
      Like strcat, but takes at most n characters from src (up to null byte). If src has >= n characters, it takes n characters and adds null byte.
    - Both return a pointer which is usually ignored because it equals dest.
    - The problem with strncat is that it is unsafe, if src is too long.
      E.g. Consider the code below:
      ```
      char s1[6] = “abc”;
      strncat(s1, “def”, 6);
      ```
      The second line would cause an overflow because the size of the array has to be at least 1 bigger than the size of the string. Therefore, n should be <= sizeof(s1) - strlen(s1) - 1.
    - Example of strncat.
      ```
      #include <stdio.h>
      #include <stdlib.h>
      #include <string.h>

      int main (){
         char src[50] = “This is source”;
         char dest[50] = “This is destination”;
         strncat(dest, src, 14);
         printf("Final destination string : %s", dest);
         // Prints "Final destination string :This is
         destinationThis is source"
         return 0;
      }
      ```

- Example of strcat.

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

  int min(int num1, int num2);

  int main (){
     char src[50] = "This is source";
     char dest[50] = "This is destination";
     strcat(dest, src);
     printf("Final destination string : %s", dest);
     // Prints "Final destination string : This is
     destinationThis is source"
     return 0;
  }
  ```

- <u>Comparing two strings:</u>
  - **int *strcmp (const char *s1, const char *s2)**
  - if Return value < 0 then it indicates str1 is less than str2.
  - if Return value > 0 then it indicates str1 is more than str2.
  - if Return value = 0 then it indicates str1 is equal to str2.
  - **int *strncmp (const char *s1, const char *s2, int n)**
    Same, but compares only the first (at most) n characters.
  - Example of strcmp.

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

    int main (){
       char str[10] = "abcdefg";
       char str2[10] = "bcdefgh";
       int ret;

       ret = strcmp(str, str2);

       printf("%d", ret); // Prints -1

       return(0);
    }
    ```

  - Example of strncmp.

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

```
int main (){
   char str[10] = "abcdefg";
   char str2[10] = "bcdefgh";
   int ret;

   ret = strncmp(str, str2, 1);

   printf("%d", ret); // Prints -1

   return(0);
}
```

- Searching a string for occurrence of a character:
    - **char *strchr(const char *s, int c)**
      This finds the first occurrence.
    - **char *strrchr(const char *s, int c);**
      This finds the last occurrence.
    - Both return a pointer to the character if found, NULL otherwise.
    - Example of strchr.

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

int main () {
  int len;
  char str[] = "abcdefghc";
  char ch = 'c';
  char *ret;

  ret = strchr(str, ch);

 printf("String starting from the first %c is : %s", ch, ret);
 // Prints "String starting from the first c is : cdefghc"

  return(0);
}
```

- Example of strrchr.

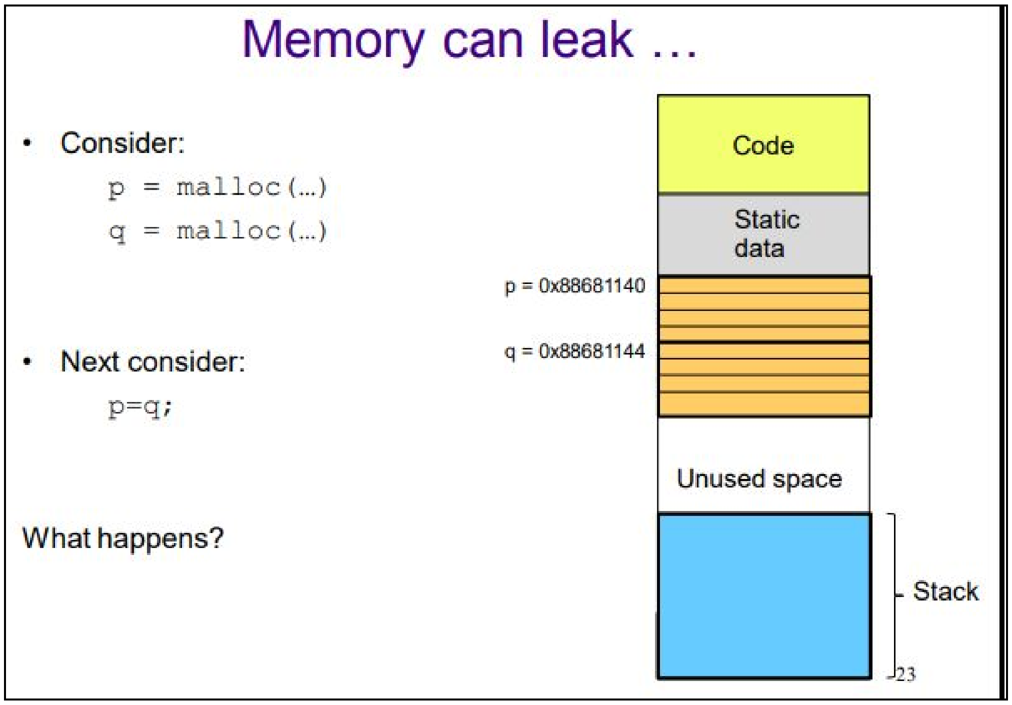
```
#include <stdio.h>
#include <string.h>

int main () {
   int len;
   char str[] = "abcdefghc";
   char ch = 'c';
   char *ret;

   ret = strrchr(str, ch);

  printf("String starting from the last %c is : %s", ch, ret);
  // Prints "String starting from the last c is : c"

   return(0);
}
```

2. **<u>Dynamic Memory Management:</u>**
   - An array is a collection of fixed number of values of a single type. That is, you need to declare the size of an array before you can use it. Sometimes, the size of array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.
   - Memory allocated here will never be released /freed automatically by the system. This is completely under programmers control.
   - Memory here can be allocated at any time during the run of a program.
   - Dynamic memory allocation allows manual control of memory allocation.
   - **<u>Malloc:</u>**
     - Used to allocate space in memory during the execution of the program.
       I.e. Malloc allocates size bytes in the Dynamic Data segment.
     - Does not initialize the memory allocated during execution. It also carries garbage value.
     - Returns a pointer to the newly acquired memory, or NULL if there is not enough available memory.
     - Syntax: **void *malloc(int num);**
     - This function allocates an array of num bytes and leave them uninitialized.
     - The size parameter malloc expects is in bytes, which is also the size of one char.
     - For anything besides char, use sizeof to obtain the size of one element.
     - E.g.

       ```
       int *a = malloc (4 * sizeof (int));
       ```

CSCB09 Week 4 Notes

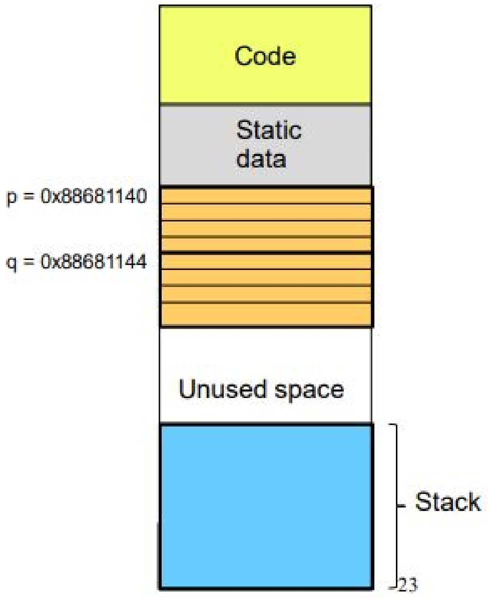
- **Calloc:**
    - Calloc is also like malloc, but calloc initializes the allocated memory to zero while malloc doesn't.
    - Furthermore, malloc allocates a single block of memory. Whereas, calloc allocates multiple blocks of memory.
    - Syntax: **void *calloc(int num, int size);**
- **Realloc:**
    - Modifies the allocated memory size by malloc and calloc to new size.
    - If enough space doesn't exist in memory of current block to extend, a new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block.
    - Syntax: **void *realloc(void *address, int newsize);**
    - This function re-allocates memory extending it upto newsize.
- **Free:**
    - Memory allocated via malloc is not released automatically. Other non-malloced memory is freed automatically when it goes out of scope.
    - If we keep calling malloc without releasing any of the memory, memory will run out.
    - Frees the allocated memory by malloc, calloc and realloc and returns the memory to the system.
    - Syntax: **void free(*address);**
    - This function releases a block of memory block specified by address.
- **Dangling Pointers:**
    - A pointer pointing to a memory location that has been deleted or freed is called dangling pointer. The pointer does not point to a valid object. This is sometimes referred to as a **premature free**.
    - The use of dangling pointers can result in a number of different types of problems, including:
        - Unpredictable behavior if the memory is accessed.
        - Segmentation faults when the memory is no longer accessible.
        - Potential security risks.
    - These types of problems can result when memory is accessed after it has been freed.
- **Memory Leak:**
    - Memory leak occurs when programmers create a memory in heap and forget to delete it.
    - To avoid memory leaks, memory allocated on heap should always be freed when no longer needed.

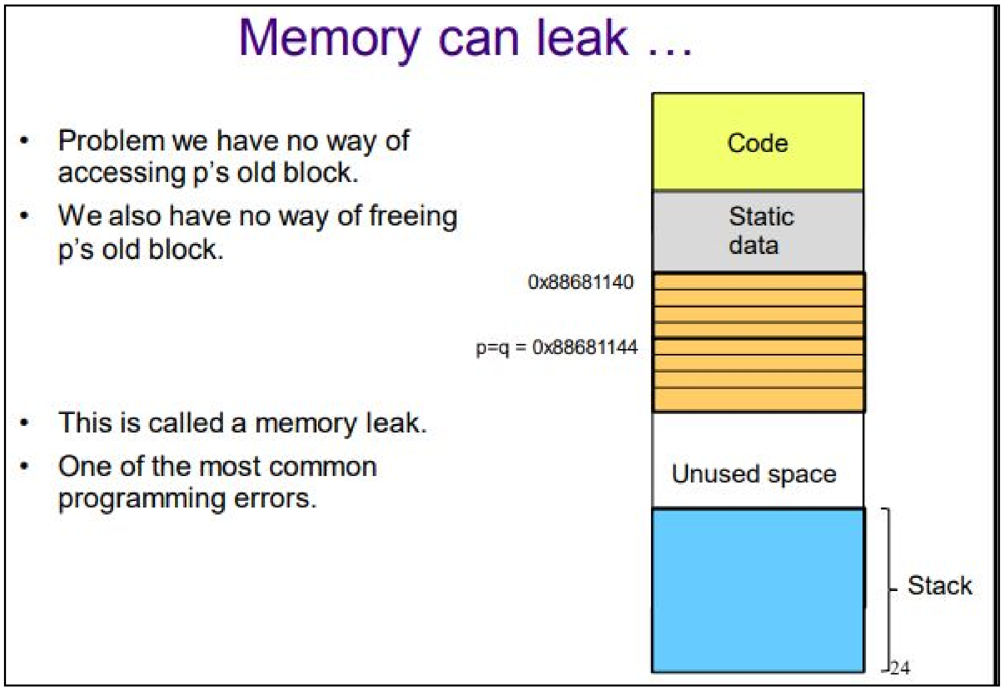


Memory can leak …

- Consider:
```
p = malloc(…)
q = malloc(…)
```
- Next consider:
```
p=q;
```

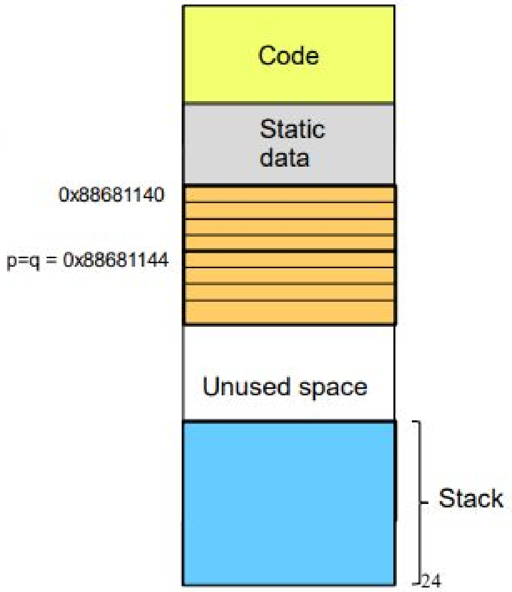
What happens?





Memory can leak …

- Problem we have no way of accessing p's old block.
- We also have no way of freeing p's old block.
- This is called a memory leak.
- One of the most common programming errors.



3. **Static VS Dynamic Memory Allocation:**

Static memory allocation	**Dynamic memory allocation**
Variables get allocated permanently and allocation is done before program execution (compile time).	Variables get allocated only if your program unit gets active and allocation is done during program execution (run time).
Consists of fixed sizes that cannot change throughout the program. I.e. Memory size can't be modified during execution.	Infinitely flexible. I.e. Memory size can be modified during execution.
Less efficient	More efficient
There is no memory reusability.	There is memory reusability and memory can be freed when not required.
Uses a stack for implementing static allocation.	Uses a heap for implementing dynamic allocation.

4. **Structs:**
    - A struct is a collection of related data items. It is a collection of variables (can be of different types) under a single name.
    - Structs are used for:
        - Serialization of data.
        - Passing multiple arguments in and out of functions through a single argument.
        - Data structures such as linked lists, binary trees, and more.
    - Syntax:
```
struct [name] {
Member definition
Member definition
...
Member definition
} [variable name]; // The variable name is optional.
```
    - Example:
```
struct student {
char firstName[20];
char secondName[20];
int year;
};
```

CSCB09 Week 4 Notes

- Struct members **cannot be** initialized with declaration. The reason for this is that when a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created.
  E.g. The code below will fail and cause an error.
  ```
  struct Point
  {
    int x = 0;
    int y = 0;
  };
  ```
- Pointers to structs:

  -
    ```
    struct student student1;
    …… // code for initializing student1 here
    struct student *p;
    p = &student1;
    ```

    How can we access student1's members?

    ```
    student1.year = 3;
    (*p).year = 3;
    p->year = 3;
    ```
- Structs and malloc:
  - Struct can be used with malloc like any other datatype.

  -
    ```
    struct student *s;
    s = malloc (sizeof (struct student));
    s->year = 3;
    ```
- Structs and functions:
  - Functions can take structs as parameters and access their elements.

  -
    ```
    void PrintStudent (struct student s) {
            printf ("First name: %s\n", s.firstName);
            printf ("Last name: %s\n", s.lastName);
            printf ("Year: %s\n", s.year);
    }
    ```
- Pointers and functions:
  - If we want to modify the arguments of a function, we need to pass a pointer. This is because parameters in C are **passed by value**, which means that the functions works on the copy, not the original variable.

Dynamic Memory Allocation

1. **Malloc:**
   - Used to allocate space in memory during the execution of the program. I.e. Malloc allocates size bytes in the Dynamic Data segment.
   - Does not initialize the memory allocated during execution. It also carries garbage value.
   - Returns a pointer to the newly acquired memory, or NULL if there is not enough available memory.
   - Because the function doesn't know what type of data we're planning on storing, it can't return a pointer to an ordinary type, like int or char. Instead, it returns a value of type void *. A void * value is a generic pointer, which is just a memory address.
   - Syntax: **void *malloc(int num);**
   - This function allocates an array of num bytes and leave them uninitialized.
   - The size parameter malloc expects is in bytes, which is also the size of one char.
   - For anything besides char, use sizeof to obtain the size of one element.
   - E.g.
     int *a = malloc (4 * sizeof (int));
   - Malloc is more efficient than calloc because it doesn't have to clear the memory block that it allocates.
2. **Calloc:**
   - Calloc is also like malloc, but calloc initializes the allocated memory to zero while malloc doesn't.
   - Furthermore, malloc allocates a single block of memory. Whereas, calloc allocates multiple blocks of memory.
   - Syntax: **void *calloc(int num, int size);**
   - Calloc also clears the memory that it allocates.
3. **Realloc:**
   - Modifies the allocated memory size by malloc and calloc to new size.
   - If enough space doesn't exist in memory of current block to extend, a new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block.
   - Syntax: **void *realloc(void *address, int newsize);**
   - This function re-allocates memory extending it upto newsize.
   - **Note:** Address must point to a memory block obtained by a previous call of malloc, calloc or realloc.
4. **Free:**
   - Malloc and other memory allocation functions obtain memory blocks from a storage pool known as a **heap**. If we call these functions too often, or if we ask them for large block of memory, it can exhaust the heap which will cause the functions to return a null pointer.
   - Memory allocated via malloc is not released automatically. Other non-malloced memory is freed automatically when it goes out of scope.

- If we keep calling malloc without releasing any of the memory, memory will run out.
- Furthermore, sometimes, there may be a **memory leak**. A block of memory that is no longer accessible to a program is said to be **garbage**. A program that leaves behind garbage has a memory leak. Unlike some languages, C does not have a garbage collector. To release unneeded memory, we can use the free function.
- I.e. Frees the allocated memory by malloc, calloc and realloc and returns the memory to the system.
- This function releases a block of memory block specified by address.
- Syntax: **void free(*address);**
- E.g. Consider the code below.

  a = malloc(...);

  b = malloc(...);

  a = b; ← Here, a's old memory block is no longer accessible, but it has not been removed. This is an example of a memory leak.

  However, if we use the free function, we can reuse that block of memory later on.

  a = malloc(...);

  b = malloc(...);

  free(a);

  a = b;
- **Note:** The argument to free() must be a pointer that was previously returned by a memory allocation function. The argument may also be a null pointer, in which case free() has no effect. However, passing a pointer to any other object, like an array or a variable, to free() will cause an undefined behaviour.
- Be careful of dangling pointers when using free().
- A pointer pointing to a memory location that has been deleted or freed is called **dangling pointer**.
- These types of problems can result when memory is accessed after it has been freed.
- When you use free(), you deallocated the memory block that a pointer points to, but you didn't change the pointer. If we try to access or modify a deallocated memory block, then we will get errors.
- **Note:** When a block of memory is freed, all the pointers that pointed to it are now dangling pointers.
- The pointer does not point to a valid object. This is sometimes referred to as a **premature free**.

- The use of dangling pointers can result in a number of different types of problems, including:
    - Unpredictable behavior if the memory is accessed.
    - Segmentation faults when the memory is no longer accessible.
    - Potential security risks.

5. **Linked Lists:**
    - A **linked list** consists of a chain of structures, called **nodes**, with each node containing a pointer to the next node in the chain.
      Note: The last node in the linked list contains a null pointer.
    - A linked list is more flexible than an array because we can easily insert or delete nodes.
      I.e. Items can be added or removed from the middle of the list.
      Furthermore, there is no need to define an initial size.
    - However, accessing a specific node in the linked list will take longer if the node is near the end.
      I.e. There is no "random" access - it is impossible to reach the nth item in the array without first iterating over all items up until that item. This means we have to start from the beginning of the list and count how many times we advance in the list until we get to the desired item.
      Furthemore, dynamic memory allocation and pointers are required, which complicates the code and increases the risk of memory leaks and segment faults.
    - **Note:** **(*node).value** can be written as **node->value**. The -> operator functions is a combination of the * and . operators.

CSCB09 Week 5 Notes

1. **Pointer to Pointer:**
   - Also known as a **Double Pointer**.
   - Syntax: type **var
   - Used to store the memory address of pointers.
   - **Note:** When dereferencing a double pointer, we do not get the final object. To get the final object, we must dereference it twice.
   - **Note:** We can also have triple pointer, quadruple pointers, etc, but we don't need to know them for CSCB09.
2. **Passing command line arguments to a C program:**
   - The command line arguments are handled using main function arguments.
   - We need the main function to take in 2 arguments:
     1. int argc: The number of command line arguments.
     2. char *argv[]: A pointer array which points to each argument passed to the program.
        a. argv[0] is the name of the program.
        b. The remaining elements of argv contain the arguments
   - **Note:** argv[1] is a pointer to the first command line argument supplied, and *argv[n] is the last argument.
   - If no arguments are supplied, argc is 1, and if you pass one argument then argc is 2.
   - E.g. Consider the C code below:

```
#include <stdio.h>
int main(int argc, char *argv[]){
  // Prints the value of argc.
  printf("The value of argc is %d\n", argc);
  printf("The value(s) of argv are:");
  // Prints out all  the values of argv.
  for (int i = 0; i < argc; i++){
     printf("%s\n", argv[i]);
  }
  return 0;
}
```

Suppose the function name is test.c. If I successfully compile test.c and do ./test a b c d e f g, then the output is:

The value of argc is 8 // This prints the value of argc, which is 8.

./test // This is argv[0].

a // This is argv[1].

b // This is argv[2].

c // This is argv[3].

d // This is argv[4].

e // This is argv[5].

f // This is argv[6].

g // This is argv[7].

3. **Typedef:**
   - Typedefs allow us to define types with a different name - which can come in handy when dealing with structs and pointers.
     I.e. You can use typedef to give a name to your user defined data types.
   - typedef also allows you to define a short-hand.
   - E.g. Consider the C codes below:

```
#include <stdio.h>
typedef struct Example1{
  int x;
  int y;
  int z;
} p1; // We can now use p1 as a data type.
int main(){
  // Using p1 as a data type.
  p1 P1 = {1,2,3};
  printf("P1.x is %d\n", P1.x);
  printf("P1.y is %d\n", P1.y);
  printf("P1.z is %d\n", P1.z);
  // Using Example1 as a data type.
  struct Example1 P2 = {1, 2, 3};
  printf("P2.x is %d\n", P2.x);
  printf("P2.y is %d\n", P2.y);
  printf("P2.z is %d\n", P2.z);
  return 0;
}
```

- If we use typedef and we use p1 as the data type, then we do not need to use struct. However, if we are using Example1 as a data type, we still need to use struct.
- Typedefs is good for large programs, so you don't have to repeatedly use structs.

4. **Organizing your program:**
   - C does not offer classes or similar concepts to organize your code.
   - Large programs are organized by breaking them down into multiple files.
   - Suppose you have these 3 files as part of a project:
     1. main.c
        ```
        #include "list.h"
        int main() {
        // Some code
        }
        ```
     2. list.c
        ```
        #include "list.h"
        int isEmpty(List *h) {
        // Some code
        }
        void add(List *h, int v) {
        // Some code
        }
        void remove(List *h, int v) {
        // Some code
        }
        ```
     3. list.h
        ```
        struct node {
                int value;
                struct node * next;
        } ;
        typedef struct node List;
        int isEmpty(List *);
        void add(List *, int);
        void remove(List *, int)
        ```
   - We can compile the above functions using **gcc main.c list.c –o myprogram**.
   - What gcc main.c list.c –o myprogram is doing is:
     1. First the **pre-processor** runs, it looks for example for #include directives and includes the corresponding .h file.

2. Then the **compiler** runs on each .c file. It produces machine code/object code for each .c file and places it in a .o file (main.o, list.o).
3. Then the **linker** takes all the object files and combines them into one executable (called myprogram in our example).

- We have recompile the program each time an edit is made because when we compile the C code, we are translating the C code to machine level code that runs directly on your hardware. Therefore, every time you make an edit a new executable has to be created.
- **Note:** To run the files together, they must be in the same folder/directory.
- **Note:** We did not have to compile each time a change was made in a shell script. This is because the commands inside a shell script are interpreted by the shell, they are not translated to a machine language program.
- The problem with manually calling gcc is that if you have a lot of files, the list can grow every long. Furthermore, this recompiles every module, even if it has not changed.
  E.g. gcc file1.c file2.c file3.c file4.c ……… –o myprogram
- We can also compile C code using this method:
  gcc –c list.c # this produces list.o
  gcc –c main.c # this produces main.o
  gcc –o myprogram main.o list.o # produces executable
- The above code only recompiles files that has changed by separating compilation & linking. However, this is still not efficient.
- We can use **Makefiles** to solve this problem.
- **Makefiles:**
  - Makefiles are processed by a program called **make**.
  - Contain information about “targets” and “dependencies”.
    E.g. myprogram depends on list.o and main.o
    E.g. list.o depends on list.c and list.h
  - Contain information about “rules”.
    E.g. To produce list.o run “gcc –c list.c”
  - **Make** looks at timestamps, and only recompiles a target if one or more of its dependencies are newer.
    I.e. Make allows a programmer to easily keep track of a project by maintaining current versions of their programs from separate sources.
  - Furthermore, make can automate various tasks for you, not only compiling proper branch of source code from the project tree, but

    helping you automate other tasks, such as cleaning directories, organizing output, and even debugging.
    - Syntax:
      <target> : <dependencies>
      action
      Note: There can be more than 1 dependency.
    - **Note:** Makefile reads from top to bottom. If Makefile finds anything that was changed, then it will compile that part only. It won't touch the stuff that wasn't changed.
    - To run Makefile, use the **make** command.
    - To run clean, do **make clean**.
  - **Pre-processor directives:**
    - Include Header Files:
      - #include: Inserts a particular header from another file.
    - Define Marcos:
      - #define: Substitutes a preprocessor macro.
      - **Note:** A macro is like a constant.
    - Conditional Inclusions:
      - #ifdef: Returns true if this macro is defined.
      - #ifndef: Returns true if this macro is not defined.
      - #undef: Undefines a preprocessor macro.
      - #endif: Ends preprocessor conditional.
      - #error: Prints error message on stderr.

5. **Error Handling:**
  - The return value of library functions tells you if there was an error.
    E.g. If malloc returns NULL, then there's an error.
  - Many library functions use a global variable called **errno** to store more information on what went wrong.
  - errno is declared in errno.h.
    I.e. #include <errno.h>
  - At process creation time errno is zero. A value of 0 indicates that there is no error in the program.
  - When a library call error occurs, errno is set.
  - Some possible values of errno:
    - ENOMEM: "Not enough space"
    - EDOM: "Domain error"
    - EACCESS: "Permission denied"
  - However, we need to be careful when we're using errno.

- Consider the C code below:

  ```
  if (somecall() == -1) {

          printf("somecall() failed\n");

          if (errno == ...)

          {

           ...

          }

  }
  ```

  The problem with this is that printf might change the value of errno if it encounters an error. If printf works successfully, then it returns a 0. Since errno looks at the last return statement, it will check the return statement of printf and see the 0, instead of the -1 from somecall(). Therefore, we need to save the value of errno before doing any further processing.

- **perror():**
  - Syntax: void perror(char *str)
  - perror displays str, then a colon(:), then an English description of the error as defined in errno.h.
  - Protocol:
    - check system calls for a return value of -1 or NULL
    - call perror() for an error description

6. **<u>I/O in C:</u>**

- A file represents a sequence of bytes.
- Two main mechanisms for managing file access:
  1. File descriptors (low-level):
     - Each open file is identified by a small integer.
     - Use for pipes, sockets (will see later what those are …)
  2. File pointers (regular files):
     - You use a pointer to a file structure (FILE *) as handle to a file.
     - The file struct contains a file descriptor and a buffer.
     - Use for regular files
- **Standard streams:**
  - All programs automatically have three files open:
    - FILE *stdin;
    - FILE *stdout;

        - FILE *stderr;

	stdio name	File descriptor	Default Location
Standard input	stdin	0	Comes in from keyboard
Standard output	stdout	1	Comes out on screen
Standard error	stderr	2	Comes out on screen

- **Opening Files:**
    - You can use the fopen function to create a new file or to open an existing file.
    - Syntax:
      FILE *fopen(const char *filename, const char *mode);
    - filename identifies the file to open.
    - mode tells how to open the file:
        - "r" for reading
          I.e. Opens an existing text file for reading purpose.
        - "w" for writing
          I.e. Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file.
        - "a" for appending
          I.e. Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here, your program will start appending content in the existing file content.
    - This returns a pointer to a FILE struct which is the handle to the file. This pointer will be used in subsequent operations.
- **Closing Files:**
    - Syntax: void fclose(FILE *stream);
- **Writing Files:**
    - fputc
        - Syntax: int fputc(int char, FILE *stream);
        - Writes the character value of the argument char to the output stream referenced by stream.
        - It returns the written character written on success otherwise EOF if there is an error.
    - fputs
        - Syntax: int fputs(const char *str, FILE *stream);

        - Writes the string str to the output stream referenced by stream.
        - Returns a non-negative value on success, otherwise EOF is returned in case of any error.
    - fprintf
        - Syntax: int fprintf(FILE *stream, const char *format, ...);
        - Stream: This is the pointer to a FILE object that identifies the stream.
        - Format: This is the C string that contains the text to be written to the stream. It can optionally contain embedded format tags that are replaced by the values specified in subsequent additional arguments and formatted as requested.
- **Reading Files:**
    - fgets:
        - Syntax: char *fgets(char *s, int size, FILE *stream);
        - fgets() reads up to size-1 characters from the input stream referenced by stream. It copies the read string into the buffer s, appending a null character to terminate the string.
        - **s:** Pointer to an array of chars where the string read is copied.
        - **size:** Maximum number of characters to be copied into str (including the terminating null-character).
        - ***stream:** Pointer to a FILE object that identifies an input stream.
          stdin can be used as argument to read from the standard input.
        - **returns:** The function returns s.
        - It is **safe** to use because it checks the array bound.
        - It keep on reading until new line character encountered or maximum limit of character array.
        - We can get fgets to read from the keyboard by using stdin for stream. E.g. fgets(s, size, stdin).
    - gets:
        - Another function to read from keyboard.
        - Syntax: char *gets(char *s);
        - Reads from keyboard until \n and stores results where buffer s points to.

    - We should never use gets because it does not check the array bound.
- **Binary:**
    - Block I/O allows you to read and write binary data, i.e. you read and write byte-for-byte rather than lines of characters.
    - Suppose you store the number 1,999,999 as follows:
      fprintf(fp, “1999999”);
      This takes 7 bytes because each character is a byte. However, by using block I/O, we can use 3 bytes to store it.
    - fread:
        - Syntax: size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
        - Reads data from the given stream into the array pointed to, by ptr.
        - Read nmemb * size bytes into memory at ptr.
        - Returns number of items read. If this number differs from the nmemb parameter, then either an error had occurred or the End Of File was reached.
        - Ptr: This is the pointer to a block of memory with a minimum size of size*nmemb bytes.
        - Size: This is the size in bytes of each element to be read.
        - Nmemb: This is the number of elements, each one with a size of size bytes.
        - Stream: This is the pointer to a FILE object that specifies an input stream.
    - fwrite:
        - Syntax: size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
        - Writes nmemb * size bytes from ptr to the file pointer stream
        - Returns number of items written. If this number differs from the nmemb parameter, it will show an error.
        - Ptr: This is the pointer to the array of elements to be written.
        - Size: This is the size in bytes of each element to be written.
        - Nmemb: This is the number of elements, each one with a size of size bytes.
        - Stream: This is the pointer to a FILE object that specifies an output stream.

CSCB09 Week 6 Notes

1. **Forks:**
   - The fork system call creates a child process and a duplicate of the currently running program. Both processes run concurrently and independently. The child gets a new PID (Process ID) and PPID (Parent Process ID).
   - The fork function creates a new process by duplicating the calling process. The new process, called the child, is an exact duplicate of the calling process, referred to as parent, except for the following :
     1. The child has its own unique process ID, and this PID does not match the ID of any existing process group.
     2. The child's parent process ID is the same as the parent's process ID.
   - The return value from the fork call is different:
     1. On success:
        - fork() returns 0 to the child.
        - fork() returns a positive value, which is the child's PID, to the parent.
     2. On failure:
        - No child is created, fork returns a negative value, usually -1, to the parent, and errno is set appropriately.
   - The fork function takes no arguments.
     pid_t fork(void); is the syntax for the fork function.
   - You need to use **#include <unistd.h>**.
   - A child process terminates when the program's main function returns or the program calls exit.
   - In shell, you can use $? to get the exit status of the most recent command. The exit status is set by the exit function or main's return.
2. **Wait:**
   - Wait suspends execution of the calling process until one of its children terminates.
   - Syntax: **pid_t wait(int *status);**
   - After calling wait() a process will:
     - Block the calling process if all of its children are still running.
     - Return immediately with the PID of a terminated child, if there is a terminated child.
     - Return immediately with an error, -1, if it doesn't have any child processes.
   - Wait returns the pid of the terminated child or -1 on error.
   - **status** encodes the exit status of the child and how a child exited (normally or killed by signal).
   - There are macros to process exit status:
     1. **WIFEXITED** tells you if child terminated normally
     2. **WEXITSTATUS** gives you the exit status
   - A child becomes a zombie when it terminates, but its parent process is not waiting for it. The child's exit code is kept around as a zombie until parent

collects its exit code through wait or until parent terminates. Shows up as Z in ps.
- A child becomes an orphan if the parent process terminates before the child.
- Orphans get adopted by the init process.
- init is the first process started during booting. It's the root of the process hierarchy. init has a PID of 1 so the PPID of orphans is 1.
- Waitpid is used if a process wants to wait for a particular child rather than any child or if a process does not want to block when no child has terminated.
- Syntax: **pid_t waitpid(pid_t pid, int *status, int options);**
- First parameter specifies PID of child to wait for.
- If pid is -1 then it means any arbitrarily child. Here waitpid() work same as wait() work.
- If options is 0, waitpid blocks, just like wait.
- If options is WNOHANG, it immediately returns 0 instead of blocking when no terminated child.

3. **Exec:**
    - The exec family of functions replaces the current process with a new process. The new program starts executing from the beginning.
    - On success, exec never returns, on failure, exec returns -1.
    - The new process inherits from calling process:
        - PID and PPID, UID, GID
        - Controlling terminal
        - CWD, resource limits
        - Pending signals
    - Exec is not one specific function, but a family of functions.
        1. **execvp**:
           Syntax: int execvp (const char *file, char *const argv[]);
           file: Points to the file name associated with the file being executed.
           argv: Is a null terminated array of character pointers**.**

        2. **execlp**:
           Syntax: **int execlp(const char *file, const char *arg,.../* (char *) NULL */);**
           file: The file name associated with the file being executed.
           const char *arg and ellipses: Describes a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program.

        3. **execv**:
           Syntax: int execv(const char *path, char *const argv[]);
           path: A pointer that points to the path of the file being executed.
           argv[]: is a null terminated array of character pointers.

4. **execl**:
   Syntax: int execl(const char *path, const char *arg,.../* (char *) NULL */);
   file: The file name associated with the file being executed.
   const char *arg and ellipses: Describes a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program.

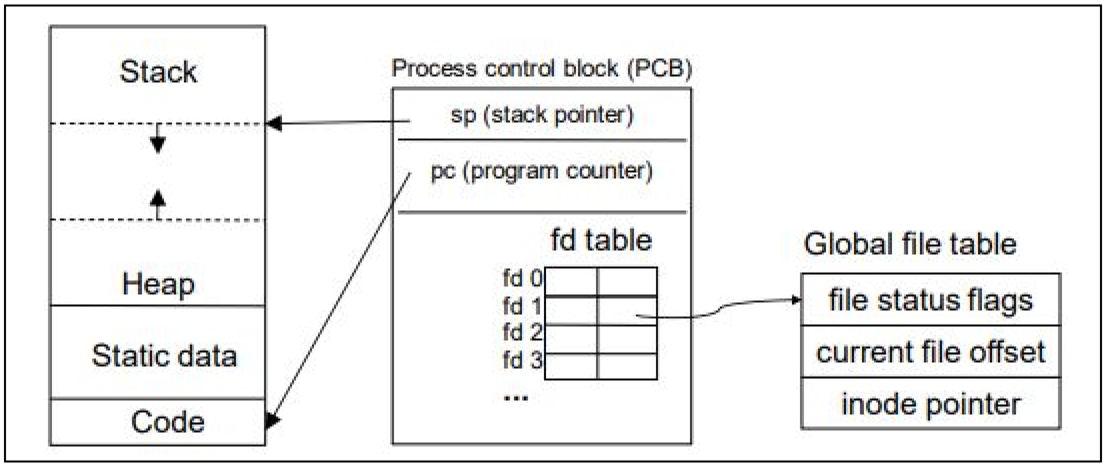
- First parameter: name of executable; then commandline parameters for executable; these are passed as argv[0], argv[1], …, to the main program of the executable.
- execl and execv differ from each other only in how the arguments for the new program are passed.
- execlp and execvp differ from execl and execv only in that you don't have to specify full path to new program.

4. **<u>Difference Between Fork and Exec:</u>**
   - Fork starts a new process which is a copy of the one that calls it while exec replaces the current process with a different one.
     I.e. Fork creates a duplicate of the current process while exec replaces the current process with a different one.
5. **<u>How a shell runs commands:</u>**
   - When a command is typed, shell forks and then execs the typed command.
6. **<u>Processes and File Descriptors:</u>**
   - File descriptors are handles to open files.
   - They belong to processes not programs.
   - They are a process's link to the outside world.
7. **<u>Initializing Unix:</u>**
   - Use the top or ps –aux command to see what's running.
   - The only way to create a new process is to duplicate an existing process. Therefore the ancestor of all processes is init with pid = 1.
   - The only way to run a program is with exec.

CSCB09 Week 8 Notes

1. **Parallel programs:**
   - Hardware is not getting that much faster anymore. Therefore, we can use more hardware to solve a problem.
     I.e. Multiple processors/cores and multiple machines.
   - Processes often need to communicate. This means that processes of a parallel program need to exchange data and/or synchronize. However, if you use fork, the parent and the child cannot use the same variables as each process has a separate copy of each variable.
   - We can use pipes to solve this problem.
2. **Pipes:**
   - Pipes are a one-way (half-duplex) communication channel.
   - Pipes are a communication medium between two or more related or interrelated processes.
   - It can be either within one process or a communication between the child and the parent processes.
   - Communication can also be multi-level such as communication between the parent, the child and the grand-child, etc.
   - Pipes are buffers managed by the OS.
   - Processes use low-level file descriptors for pipe operations.
   - Communication is achieved by one process writing into the pipe and other reading from the pipe. To achieve the pipe system call, create two files, one to write into the file and another to read from the file.
3. **I/O mechanisms in C:**
   - So far we have used **file pointers** (regular files).
     I.e. FILE *file
   - For pipes, we need to use **file descriptors (fd)**, which are low level.
   - A file descriptor is an integer that uniquely identifies an open file in a computer's operating system.
   - When a process makes a successful request to open a file, the kernel returns a file descriptor which points to an entry in the kernel's global file table. The file table entry contains information such as the inode of the file, byte offset, and the access restrictions for that data stream (read-only, write-only, etc).
   - A **file descriptor table (fd table)** is a collection of file descriptors in which elements are pointers to the file table's entries.
   - A unique file descriptor table is provided in the operating system for each process.
   - Each process's fd table has all the open files of that process.
   - The global fd table has all the open files system-wide.

CSCB09 Week 8 Notes



- The fd table is preserved if you exec.
- If you fork, a child gets a copy of parent's fd table, so it will have same files open. Both the parent's and the child's fd will point to same entry in the global file table, so they see the same offset.
- **Note:** File descriptors created after calling fork are not shared.
- All programs automatically have three files open:
    - FILE *stdin
    - FILE *stdout
    - FILE *stderr;
- These are known as the **Standard File Descriptors**.

	Stdio name	File descriptor
Standard input	stdin	0 (STDIN_FILENO)
Standard output	stdout	1 (STDOUT_FILENO)
Standard error	stderr	2 (STDERR_FILENO)

- A useful system call to get the corresponding fd for a FILE object is: **int fileno(FILE *stream);**
- The operations for file descriptors are open, close, read and write.
- **Open:**
    - Used to open the file for reading, writing or both. This returns a file descriptor upon success or -1 upon failure.
    - Syntax: **int open(const char *pathname, int flags);**
    - **Pathname** is the path to the file that you want to use. Use an absolute path that begins with "/", if you are not working in same directory as the file. Use a relative path if you are working in same directory as the file.
    - **Flags:**
        - **O_RDONLY:** Opens the file for read-only.
        - **O_WRONLY:** Opens the file for write-only.
        - **O_RDWR:** Opens the file for both reading and writing.

- **Note:** To use the open function, you must include the following:
  - **#include<sys/types.h>**
  - **#include<sys/stat.h>**
  - **#include <fcntl.h>**
- **Read:**
  - This reads count bytes of input into the memory area indicted by buf from the file indicated by fd. This returns the number of bytes read on success, 0 on reaching the end of file and -1 on error.
  - Syntax: **ssize_t read(int fd, void *buf, size_t count);**
  - **buf:**
    - The buffer (Memory Area) that stores the data when you read it.
    - buf needs to point to a valid memory location with size not smaller than count.
  - **count:**
    - This is the length of the buffer.
      I.e. This is the number of bytes requested to be read.
    - **Note:** Sometimes, the number of bytes requested to be read is not the same as the actual number of bytes read.
- **Write:**
  - This writes count bytes from buf to the file indicated by fd. If cnt is zero, write simply returns 0 without attempting any other action. Furthermore, write will overwrite the contents of the file, if there are any. This returns the number of bytes written on success and -1 on error.
  - Syntax: **ssize_t write(int fd, const void *buf, size_t count);**
  - **buf:**
    - The buffer that stores the data being written from.
    - buf needs to point to a valid memory location with size not smaller than count.
  - **count:**
    - The length of the buffer.
      I.e. This is the number of bytes requested to be written.
    - **Note:** Sometimes, the number of bytes requested to be written is not the same as the actual number of bytes written.
- **Close:**
  - Closes the file pointed by fd.
  - This returns 0 on success and -1 on failure.
  - Syntax: **int close(int fd);**

4. **Programming With Pipes:**
   - You need to use **#include<unistd.h>**.
   - Syntax: **int pipe(int pipefd[2]);**
   - You pass a pointer to two integers (i.e. an array or a malloc of two ints) and pipe fills it with two newly opened FDs.

I.e. pipe creates an OS internal system buffer and two file descriptors, one for reading and one for writing.

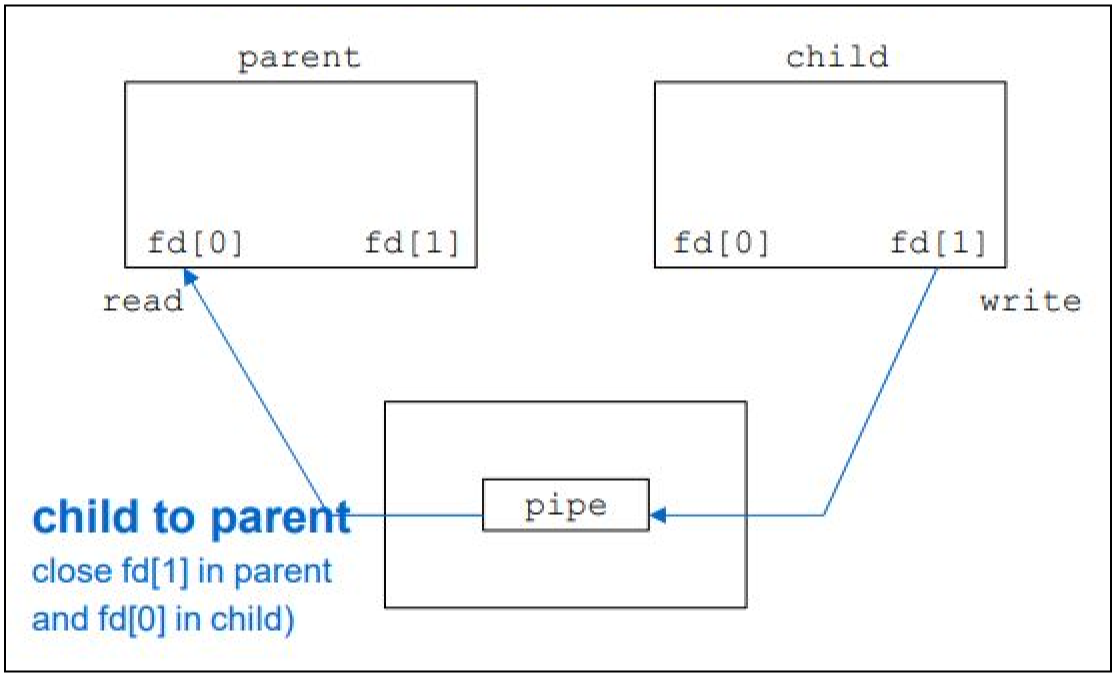
- pipefd[0] is used for reading.
- pipefd[1] is used for writing.
- Whatever is written into pipefd[1] can be read from pipefd[0].
- Returns 0 on success and -1 on error. To know the cause of failure, check with errno or perror.
- Read blocks until data is available in the pipe. If the writing end is closed, read detects EOF and returns 0.
- All open pipes (and other FDs) are closed when a process exits.
- In general, each process will read or write a pipe (not both). You should close the end you are not using.
  I.e. Before reading: close pipefd[1].
  Before writing: close pipefd[0].
- If you don't close the end you are not using, your code might hang.
- E.g.

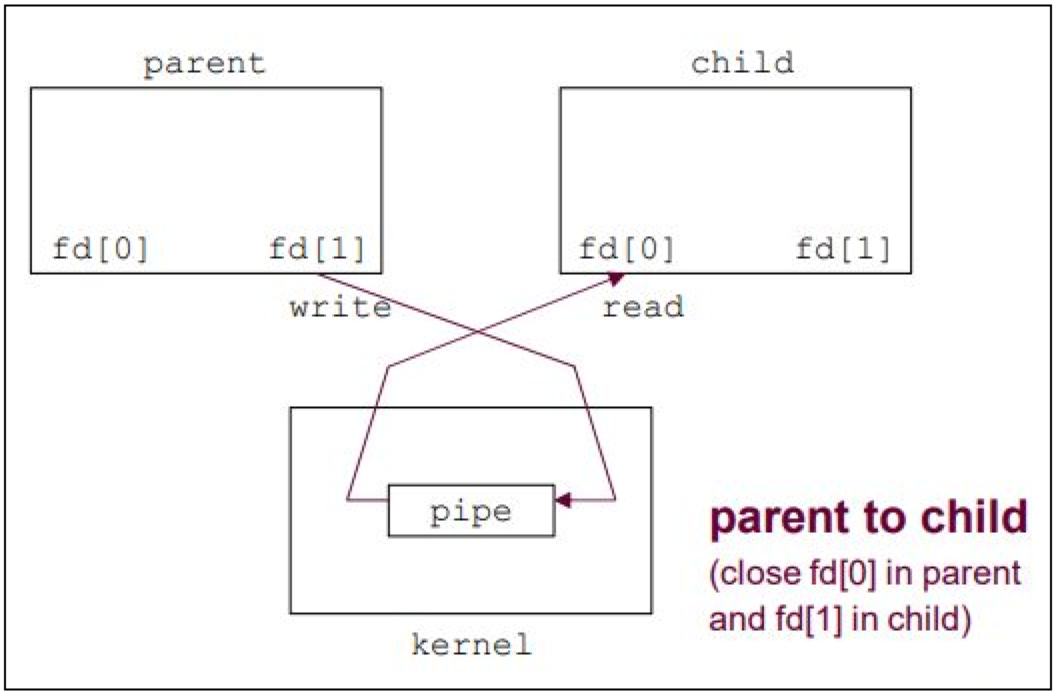
```
if (fork == 0){
   write(p[1], msg, msgsize);
}else{
   while((nbytes=read(p[0], buf, bufsize)) > 0){
      write(STDOUT_FILENO, buf, nbytes);
   }
}
```

This code will hang because parent wants to read until EOF and write. However, because p[1] wasn't closed, read is always expecting more data, so it blocks.

```
if (fork == 0){
   write(p[1], msg, msgsize);
}else{
  close(p[1]);
   while((nbytes=read(p[0], buf, bufsize)) > 0){
      write(STDOUT_FILENO, buf, nbytes);
   }
}
```

Because we closed p[1] before reading, now the parent won't hang.

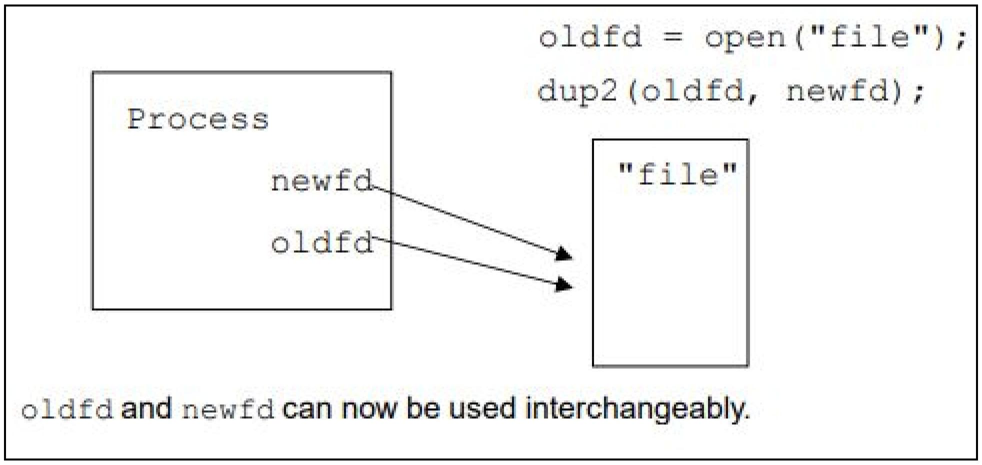




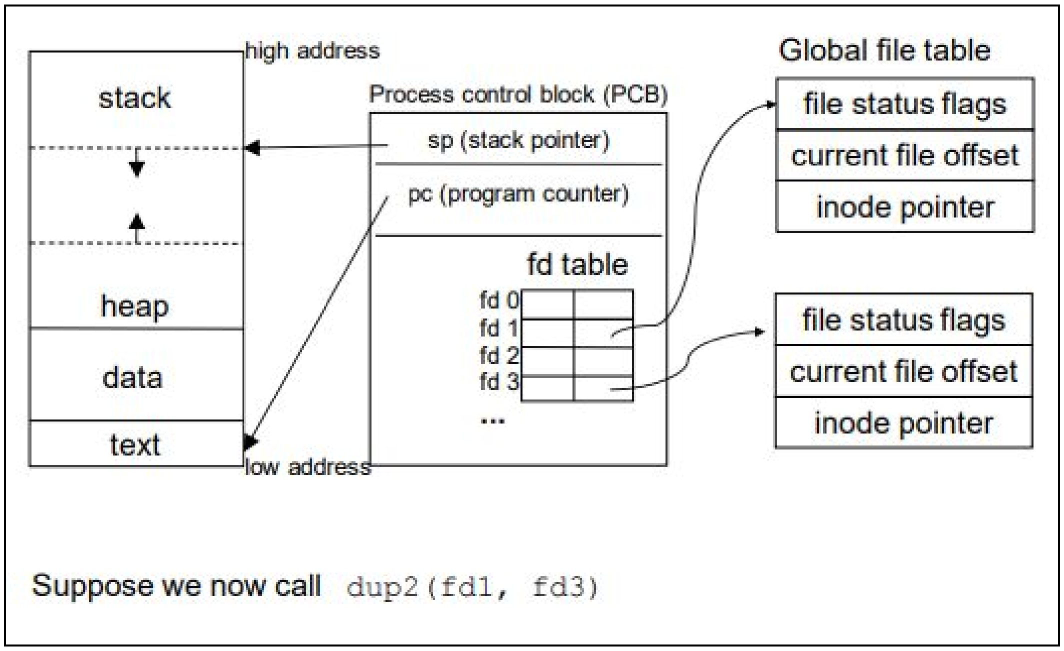
-
- If you try to write to a pipe and no one has reading end open, you get a signal (SIGPIPE) that will terminate program.
- If you write to a pipe and the buffer is full, write blocks until space frees up.
- If you read from a pipe and nobody has a writing end open, read returns EOF.
- If you read from a pipe and there is no data read blocks.

5. Dup2:

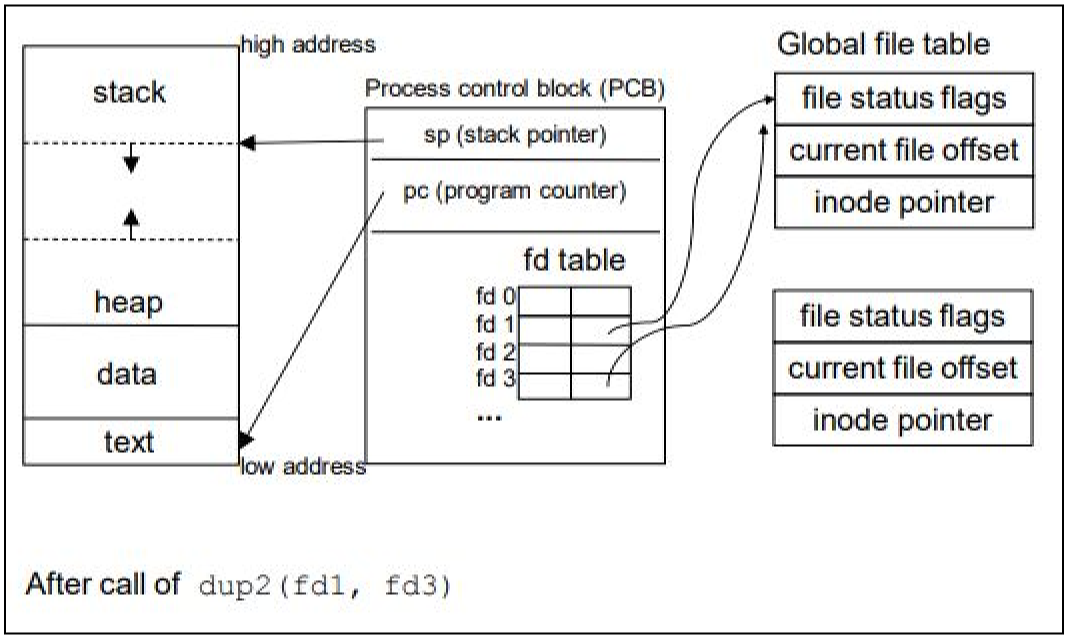
- Syntax: **int dup2(oldfd, newfd);**
- dup2 duplicates the file descriptor of oldfd into newfd.
- If the descriptor newfd was previously open, it is silently closed before being reused.
- If oldfd is not a valid file descriptor, then the call fails, and newfd is not closed.
- If oldfd is a valid file descriptor, and newfd has the same value as oldfd, then dup2() does nothing, and returns newfd.
- Note: You must include **#include<unistd.h>** to use dup2.
- On success, dup2 returns a new file descriptor, newfd, that has the following in common with the original:
    - Same open file or pipe.
    - Same file pointer (both file descriptors share one file pointer).
    - Same access mode (read, write, or read/write).
- On failure, dup2 returns -1 and errno is set accordingly.
- dup2 is used to redirect input and output. It works similar to “>” and “<” from Unix commands.
- E.g.



- E.g.



Suppose we now call `dup2(fd1, fd3)`



After call of `dup2(fd1, fd3)`

CSCB09 Week 9 Notes

1. **Signals:**
    - A signal is a software generated interrupt that is sent to a process by the OS because the user pressed ctrl-c or another process wants to tell something to this process.
    - A form of Inter-Process Communication (IPC).
    - Unexpected/unpredictable asynchronous events, called **interrupts**, can happen at any time. Some interrupts are:
        - Floating point error
        - Death of a child
        - Interval timer expired (alarm clock)
        - control-C (termination request)
        - control-Z (suspend request)
    - When the kernel recognizes an event, it sends a signal to the process.
    - Normal processes may send signals, too.
    - Signals are generated by:
        - Machine interrupts
        - The program itself
        - Other programs
        - The user (You can send signals from the shell)
    - Signals do not transfer data between processes.
    - <sys/signal.h> lists the signal types.
    - Use the command **man 7 signal** to get some description of various signals.
    - Syntax: **void (*signal(int sig, void (*func)(int)))(int)**
    - **Sig:** This is the signal number to which a handling function is set.
      I have included a list of useful signals at the end.
    - **Func:** This is a pointer to a function. This can be a function defined by the programmer or one of the following predefined functions:
        - SIG_IGN
        - SIG_DFL

      Note: Both are defined under signal handlers.
    - You need to use **#include <signal.h>**
2. **What Signals Are Used For:**
    - When a program forks into 2 or more processes, they rarely execute independently. The processes usually require some form of synchronization which is often handled by signals.
3. **Types of Signals:**
    - **SIGINT:** Terminates the process by pressing CTRL-C (^C)
    - **SIGSTOP:** Suspends the process by pressing CTRL-Z (^Z)
    - **SIGSEGV:** Segmentation fault, usually from bad pointer usage
    - **SIGPIPE:** Writing to a pipe whose read end is closed
4. **Signal Handlers:**
    - When a C program receives a signal, control is immediately passed to a function called a **signal handler**.
    - **SIG_IGN** will ignore the signal.

- **SIG_DFL** will take the default action, which is usually to terminate the process. Every signal has a default action associated with it. The default action for a signal is the action that a script or program performs when it receives a signal.
- The signal handler can execute some C statements and exit in 3 different ways:
    - Return control to the place in the program which was executing when the signal occurred.
    - Return control to some other point in the program.
    - Terminate the program by calling exit.

5. **How a Process can deal with the Signal:**
    - Take a specified user-defined action.
    - Ignore the signal altogether and carry on processing.
    - Take the default action, which is usually to terminate the process.
6. **Signal table:**
    - For each process, Unix maintains a table of actions that should be performed for each kind of signal (see below).
    - The default action can be changed for most signal types using the sigaction() function. The exceptions are SIGKILL and SIGSTOP.

Signal	**Default Action**	**Comment**
SIGINT	Terminate	Interrupt from keyboard
SIGSEGV	Terminate/Dump core	Invalid memory reference
SIGKILL	Terminate (Cannot ignore)	Kill
SIGCHLD	Ignore	Child stopped or terminated
SIGSTOP	Stop (cannot ignore)	Stop process
SIGCONT	Resume execution at the point where the process was stopped, after first handling any pending unblocked signals	Continue if stopped

7. **Sigaction:**
    - The sigaction system call is used to change the action taken by a process on receipt of a specific signal.
    - Syntax: **int sigaction(int sig, const struct sigaction *act, struct sigaction *oldact);**
    - sig specifies the signal and can be any valid signal except SIGKILL and SIGSTOP.
    - If act is non-null, the new action for signal sig is installed from act.
    - If oldact is non-null, the previous action is saved in oldact.

- The sigaction structure is defined as something like:

```
struct sigaction {
    void (*sa_handler)(int);
    // SIG_DFL, SIG_IGN, or pointer to function.
    // Pointer to a signal-catching function or one of the
       macros SIG_IGN or SIG_DFL.

    sigset_t sa_mask;
    // Signals to block during handler.
    // Additional set of signals to be blocked during
       execution of a signal-catching function.

    int sa_flags;
    // flags and options, SA_RESTART, SA_RESETHAND
    // Special flags to affect the behavior of signals.
    };
```

- **Note:** There are more extensions for the struct.
- If a signal is received when a signal handler is running, the first signal handler is suspended and the newly invoked handler runs, then the first handler is resumed.
- sa_mask can be used to block out signals while one handler is running.

8. **Blocking Signals:**
    - Signals can arrive at any time.
    - To temporarily prevent a signal from being delivered, we block it.
    - The signal is held until the process unblocks the signal, then it is delivered.
    - When a process ignores a signal, it is thrown away.
    - Blocked signals are held until they are unblocked, this is different than being ignored.
9. **Signal Set:**
    - Signal sets are used to store the set of signals that are currently blocked or unblocked.
    - Operations on signal sets:
        - **int sigemptyset(sigset_t *set);**
        - **int sigfillset(sigset_t *set);**
        - **int sigaddset(sigset_t *set, int signo);**
        - **int sigdelset(sigset_t *set, int signo);**
        - **int sigismember(const sigset_t *set, int signo);**
    - Example:
        - **sigset_t mask;**
          sigfillset(&mask) // Blocks every signal during handler
          sigemptyset(&mask) // Don't block any signals during the handler
          sigaddset(&mask, SIGQUIT) // Block SIGQUIT
          sigdelset(&mask, SIGALRM) // Unblock SIGALRM
          sigismember(&mask, SIGHUP) // Check if SIGHUP is waiting

10. **Sigprocmask:**
    - Used to fetch and/or change the signal mask of the calling thread. I.e. It examines and changes blocked signals.
    - Syntax: **int sigprocmask(int how, const sigset_t *set, sigset_t *oset);**
    - how indicates how the signal will be modified:
        - SIG_BLOCK: Add to those currently blocked
        - SIG_UNBLOCK: Delete from those currently blocked
        - SIG_SETMASK: Set the collection of signals being blocked
    - set points to the set of signals to be used for modifying the mask.
    - oset on return holds the set of signals that were blocked before the call.
11. **Sending a signal:**
    - You can use the kill command to send a signal to a process.
    - Syntax: **kill [-signal] pid [pid]…**
    - If you don't specify which signal to send, by default the **TERM** signal is sent, which terminates the process.
    - The signal can be specified by the number or the name without the SIG.
    - E.g.
        - kill -QUIT 8883
        - kill -STOP 78911
        - kill -9 76433 This is the same as kill -KILL 76433 because 9 == KILL.
12. **Signalling between processes:**
    - One process can send a signal to another process using the kill function.
    - **Note:** The kill function is not the same as the kill command.
    - Syntax: **int kill(pid_t pid, int sig);**
    - **pid:** id of destination process
    - **signal:** the type of signal to send
    - **Return value:** 0 if signal was sent successfully and -1 if the signal was not sent successfully.
    - The kill function sends a signal specified by sig to a process or a group of processes specified by pid.
    - Need to use the following:
        - **#include <sys/types.h>**
        - **#include <signal.h>**
    - Signalling between processes can be used for many purposes:
        - Kill errant processes
        - Temporarily suspend execution of a process
        - Make a process aware of the passage of time
        - Synchronize the actions of processes.
13. **Timer signals:**
    - Three interval timers are maintained for each process:
        - **SIGALRM:** Real-time alarm, like a stopwatch
        - **SIGVTALRM:** Virtual-time alarm, measuring CPU time
        - **SIGPROF:** Used for profilers

- Useful functions to set and get timer info:
    - **sleep():** Suspends the calling process
    - **usleep():** Like sleep() but at a finer granularity
    - **alarm():** Sets SIGALRM
    - **pause():** Suspends the process until next signal arrives
    - **setitimer()**
    - **getitimer()**
- sleep() and usleep() are interruptible by other signals.

14. Limitations:

- Signals don't contain data, all you know is the signal type.
- Multiple signals:
    - If repeated signals are sent before a process has a chance to check for its signals, the effect is the same as being sent only once (second signal is lost).
    - If multiple (different) signals are sent, all will be received when the process has a chance to check.
- Signals are usually used to indicate conditions and synchronization.

15. List of Signals:

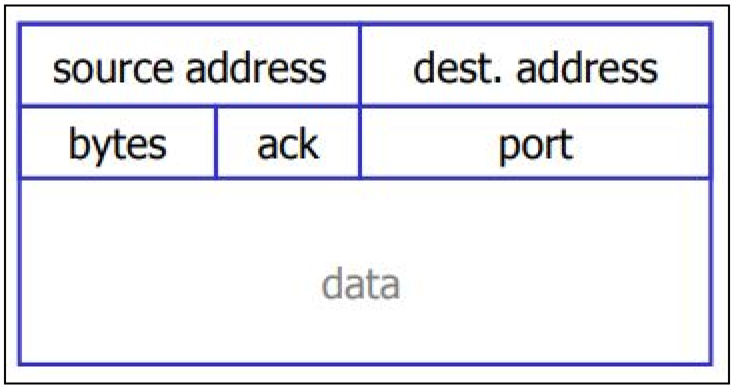
- **Note:** You can use kill -l to get this.

Number	**Signal**	**What it does**
1	SIGHUP	Used to report the termination of the controlling process on a terminal to jobs associated with that session. This termination effectively disconnects all processes in the session from the controlling terminal. (Ctrl + D)
2	SIGINT	Issued if the user sends an interrupt signal (Ctrl + C)
3	SIGQUIT	Issued if the user sends a quit signal (Ctrl + \)
8	SIGFPE	Reports a fatal arithmetic error. Although the name is derived from “floating-point exception”, this signal actually covers all arithmetic errors, including division by zero and overflow.
9	SIGKILL	If a process gets this signal it must quit immediately and will not perform any clean-up operations. Cannot be blocked or ignored.
11	SIGSEGV	Generated when a program tries to read or write outside the memory that is allocated for it, or to write memory that can only be read.
13	SIGPIPE	Writing to a pipe whose read end is closed.
14	SIGALRM	Alarm clock signal (Used for timers)
15	SIGTERM	Terminates the process. (Sent by the kill command by default)
17	SIGCHLD	Sent to a parent process whenever one of its child processes terminates or stops.
18	SIGCONT	Continues a process.
19	SIGSTOP	Stops the process. (Ctrl + Z) It cannot be handled, ignored, or blocked.
26	SIGVTALRM	Virtual-time alarm, measuring CPU time.
27	SIGPROF	Indicates expiration of a timer that measures both CPU time used by the current process, and CPU time expended on behalf of the process by the system. This is used to implement code profiling facilities.

CSCB09 Week 10 Notes

1. **IP Address:**
    - Every computer on the Internet has an internet address, called its **IP (Internet Protocol) address**.
    - An IP address is 4 numbers separated by dots.
2. **Protocols:**
    - Computers use several layers of general protocols to communicate.
    - A **network protocol** is a set of established rules that dictates how to format, transmit and receive data so computer network devices can communicate regardless of the differences in their underlying infrastructures, designs or standards.
    - For example, HTTP is a high-level protocol specific to the web.
3. **TCP/IP:**
    - **TCP** stands for **Transmission Control Protocol**.
    - **TCP/IP (Transmission Control Protocol/Internet Protocol)** is a suite of communication protocols used to interconnect network devices on the internet. It can also be used as a communications protocol in a private network, such as an intranet or an extranet.
    - TCP/IP specifies how data is exchanged over the internet by providing end-to-end communications that identify how it should be broken into packets, addressed, transmitted, routed and received at the destination.
    - TCP/IP requires little central management, and it is designed to make networks reliable, with the ability to recover automatically from the failure of any device on the network.

      I.e. TCP/IP tells the computer how to package up the data.



    - The TCP defines how applications can create channels of communication across a network. It also manages how data is assembled into smaller **packets** before they are then transmitted over the internet and reassembled in the right order at the destination address.
    - A **packet** is the unit of data that is routed between an origin and a destination on any packet-switched network, including the internet.
    - The IP defines how to address and route each packet to make sure it reaches the right destination. Each gateway computer on the network checks this IP address to determine where to forward the message.
    - E.g. When any file is sent from one place to another on the internet, the TCP layer divides the file into chunks of an efficient size for routing. Each

of these packets is separately numbered and includes the internet address of the destination. The individual packets for a given file may travel different routes through the internet. When they have all arrived, they are reassembled into the original file by the TCP layer at the receiving end.
- TCP/IP uses the **client/server model of communication**.
- In the **Client-Server model of communication**, a client process wants to talk to a server process. First, the client must find server by doing a DNS lookup. Then, the client must find process on server by using ports. Finally the client must establish a connection so two processes can talk.
- Advantages of TCP/IP include:
    - It is non-proprietary and, as a result, is not controlled by any single company. Therefore, the internet protocol suite can be modified easily.
    - It is compatible with all operating systems, so it can communicate with any other system.
    - The internet protocol suite is also compatible with all types of computer hardware and networks.
- Sometimes a packet might not arrive, because of traffic overload or bit corruption. In that case, the receiver asks for missing packets to be resent. While we want to send data as fast as possible, sending data too fast wastes resources. We can use **TCP Congestion Control** to solve this problem.

4. **TCP Congestion Control:**
    - Network congestion may occur when a sender overflows the network with too many packets. At the time of congestion, the network cannot handle this traffic properly.
    - What the TCP Congestion Control does is that it creates a window that allows some sent packets to be ack'd. As more packets are **ack'd**, it increases the capacity of the window. If a packet loss is discovered, it decreases the capacity of the window.
    - **Ack** is the name of a signal that data has been received successfully.
5. **Introduction to Sockets:**
    - Sockets allow communication between two different processes on the same or different machines using standard Unix file descriptors.
    - Similar to pipes, except sockets can be used between processes on different machines.
    - Sockets are built on top of the TCP layer.
    - A Unix Socket is used in a client-server application framework.
    - A **server** is a process that performs some functions on request from a client.
6. **Types of Sockets:**
    - There are two main categories of sockets:
        1. **UNIX domain:** Both processes are on the same machine.
        2. **INET domain:** The processes are on different machines.

- There are three main types of sockets:
    1. **SOCK_STREAM (Stream Sockets):** Delivery in a networked environment is guaranteed. If you send through the stream socket three items "A, B, C", they will arrive in the same order, "A, B, C". These sockets use TCP for data transmission. If delivery is impossible, the sender receives an error indicator. Data records do not have any boundaries.
    2. **SOCK_DGRAM (Datagram Sockets):** Delivery in a networked environment is not guaranteed. They use UDP (User Datagram Protocol) and are connectionless because they don't need to have an open connection. Instead, you build a packet with the destination information and send it out.
    3. **SOCK_RAW (Raw Sockets):** These provide users access to the underlying communication protocols, which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more cryptic facilities of an existing protocol.

7. **Addresses and Ports:**
    - A **socket pair** is the two endpoints of the connection.
    - An endpoint is identified by an IP address and a port.
    - IPv4 (IP version 4) addresses are four 8-bit numbers.
    - A **port** is an endpoint for communication.
    - Ports are identified on a server for each protocol and address, known as the **port number**.
    - We use ports because multiple processes can communicate with a single machine, so we need another identifier.
    - The port assignments to network services can be found using the command **cat /etc/services**.
    - Well-known ports are from 0 - 1023.
    - E.g.:
        - 80 = http
        - 22 = ssh
        - 23 = telnet
    - Registered ports are from 1024 - 49151.
    - E.g.:
        - 2709 = supermon
        - 26000 = quake
        - 3724 = world of warcraft
    - Dynamic (private) ports are from 49152 - 65535.

8. **Server Side:**
    - Need to use **#include <sys/socket.h>**
    - Steps:
        1. Create a socket: socket().
        2. Assign a name to a socket: bind().
        3. Establish a queue for connections: listen().
        4. Get a connection from the queue: accept().
    - **Syntax for socket():**
        - This creates a socket. It returns a socket descriptor, an integer (like a file-handle). If the return number is -1, then there was an error.
        - **int socket(family, type, protocol)**
        - **family:** Specifies protocol family:
            - PF_INET: IPv4
            - PF_LOCAL: Unix domain
        - **type:** The communication type.
            - SOCK_STREAM
            - SOCK_DGRAM
            - SOCK_RAW
        - **protocol:** Set to 0 except for RAW sockets.
    - **Syntax for bind():**
        - The bind function assigns a local protocol address to a socket. This function is called by TCP server only. This returns 0 if it successfully binds to the address, otherwise it returns -1 on error.
        - **int bind(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);**
        - **Sockfd:** The number returned by socket().
        - **struct sockaddr_in {**

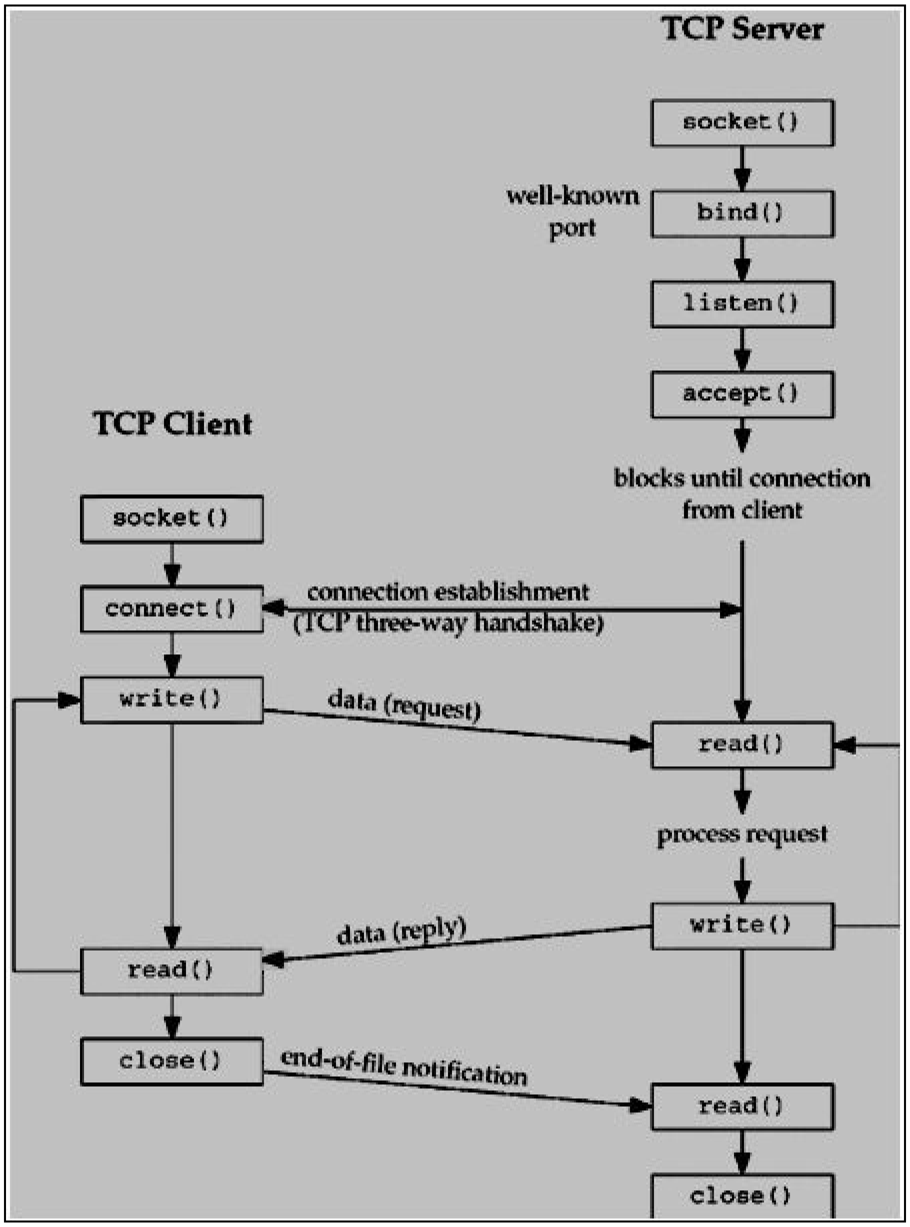
```
struct sockaddr_in {
  short int           sin_family;
  unsigned short int  sin_port;
  struct in_addr      sin_addr;
  unsigned char       sin_zero[8];
};
```

Attribute	**Values**	**Description**
sin_family	PF_INET	Represents the protocol family.
sin_port	Service Port	A 16-bit port number in Network Byte Order.
sin_addr	IP Address	A 32-bit IP address in Network Byte Order. **Note:** sin_addr can be set to **INADDR_ANY** to communicate on any network interface.
sin_zero[8]	Filling	You just set this value to NULL as this is not being used.

    - **Servaddr:** A pointer to struct sockaddr that contains the local IP address and port.
    - **Addrlen** should be set to sizeof(struct sockaddr).
- **Syntax for listen():**
    - After calling listen, a socket is ready to accept connections.
    - It prepares a queue in the kernel where partially completed connections wait to be accepted.
    - **int listen(int sockfd, int backlog**)
    - **Sockfd:** The number returned by socket().
    - **Backlog:** The maximum number of partially completed connections that the kernel should queue.
- **Syntax for accept():**
    - It blocks, waiting for a connection from the queue, creates a new connected socket, and returns a new descriptor, which refers to the TCP connection with the client.
    - At this point, connection is established between client and server, and they are ready to transfer data.
    - Reads and writes on the connection will use the socket returned by accept.
    - **int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);**
    - **Sockfd:** Is the listening socket.
    - **Cliaddr:** A pointer to struct sockaddr that contains client IP address and port.
    - **Addrlen** should be set to sizeof(struct sockaddr).

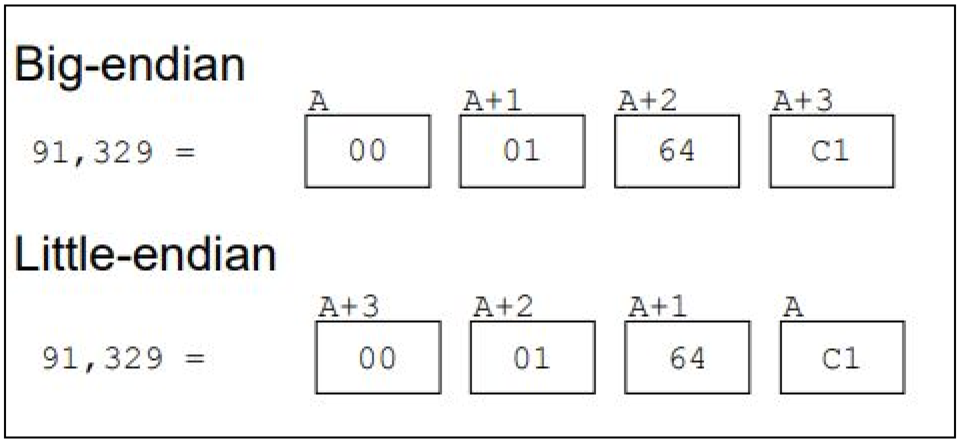
9. **Client Side:**
    - Need to use **#include <sys/socket.h>**
    - Steps:
        1. Create a socket: socket(). This is the same as the server side.
        2. Initiate a connection: connect().
    - **Syntax for connect():**
        - The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by servaddr. The server's address and port is specified in servaddr.
        - The kernel will choose a dynamic port and source IP address.
        - Returns 0 on success and -1 on failure setting errno.

- Initiates the three-way handshake (The picture below).



- **int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);**
- **Sockfd:** The number returned by socket().
- **Servaddr:** A pointer to struct sockaddr that contains destination IP address and port.
- **Addrlen** should be set it to sizeof(struct sockaddr).

10. **Byte order:**
    - Unfortunately, not all computers store the bytes that comprise a multibyte value in the same order. Consider a 16-bit internet that is made up of 2 bytes. There are two ways to store this value:
        1. **Little Endian:** In this scheme, a low-order byte is stored on the starting address (A) and a high-order byte is stored on the next address (A + 1).
           I.e. Little Endian byte ordering places the least significant byte first.
        2. **Big Endian:** In this scheme, a high-order byte is stored on the starting address (A) and a low-order byte is stored on the next address (A + 1).
           I.e. Big Endian byte ordering places the most significant byte first.



    - Intel is little-endian, and Sparc is big-endian.
    - To allow machines with different byte order conventions communicate with each other, we convert numbers to **network byte order** (big-endian) before we send them.
    - There are functions provided to do this:

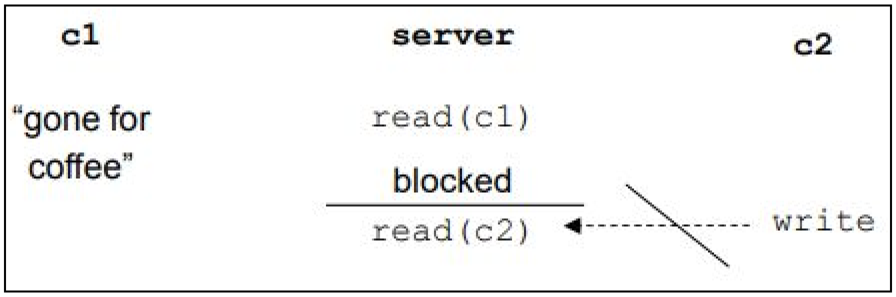
Function	**Description**
unsigned long htonl(unsigned long)	This function converts 32-bit quantities from host byte order to network byte order.
unsigned short htons(unsigned short)	This function converts 16-bit quantities from host byte order to network byte order.
unsigned long ntohl(unsigned long)	This function converts 32-bit quantities from network byte order to host byte order.
unsigned short ntohs(unsigned short)	This function converts 16-bit quantities from network byte order to host byte order.

11. **Sending and Receiving Data:**
    - Read and write calls work on sockets, but sometimes we want more control.
    - We can use the send function and the receive function instead.
    - **Send():**
        - The send function is used to send data over stream sockets or connected datagram sockets.
        - Syntax: **ssize_t send(int fd, const void *buf, size_t len, int flags);**
        - This call returns the number of bytes sent out, otherwise it will return -1 on error.
        - **fd** is the socket descriptor returned by the socket function.
        - **buf** is a pointer to the data you want to send.
        - **len** is the length of the data you want to send (in bytes).
        - **flags:**
            - If flags==0, then send() works like write.
            - If flags is MSG_OOB, then it sends out-of-band data on sockets that support out-of-band communications. The significance and semantics of out-of-band data are protocol-specific.
            - If flags is MSG_DONTROUTE, then it doesn't include routing information in the message.
            - If flags is MSG_DONTWAIT, then it enables a non-blocking operation.
    - **Recv():**
        - The recv function is used to receive data over stream sockets or connected datagram sockets.
        - Syntax: **ssize_t recv(int fd, void *buf, size_t len, int flags);**
        - This call returns the number of bytes read into the buffer, otherwise it will return -1 on error.
        - **fd** is the socket descriptor returned by the socket function.
        - **buf** is the buffer to read the information into.
        - **len** is the maximum length of the buffer.
        - **flags:**
            - If flags==0, then recv() works like read.
            - If flags is MSG_OOB, then it requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific.
            - If flags is MSG_WAITALL, then it requests that the function block until the full amount of data can be returned.
            - If flags is MSG_PEEK, then it peeks at an incoming message. The data is treated as unread and the next *recv*() or similar function shall still return this data.

12. **Close:**
    - The close function is used to close the communication between the client and the server. When finished using a socket, the socket should be closed.
    - Syntax: **int close(int socketfd)**
    - Socketfd is the file descriptor of the socket being closed.
    - This returns 0 on success, otherwise it returns -1 on error.
    - Closing a socket:
        - Closes a connection for SOCK_STREAM.
        - Frees up the port used by the socket.
13. **PF Vs AF:**
    - PF stands for **Protocol Family**. It refers to anything in the protocol, usually sockets and ports.
    - AF stands for **Address Family**. It refers to addresses from the internet, IP addresses specifically.

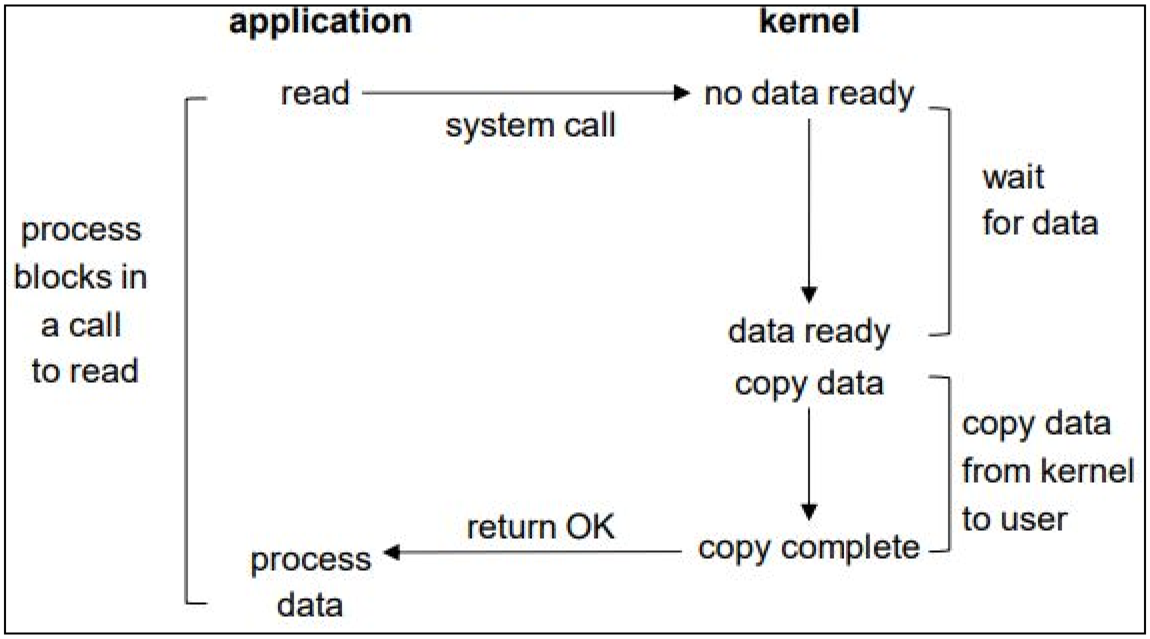
Week 11 Notes

1. **I/O Multiplexing:**
    - **I/O multiplexing** is a process which is used to monitor multiple file descriptors and choose among them which fd is ready for reading or writing to a socket.
    - Consider this problem: A server is ready to read with its clients. However, the server can only read from 1 client at a time. If the first client is gone, then all the other clients are blocked.
      I.e.

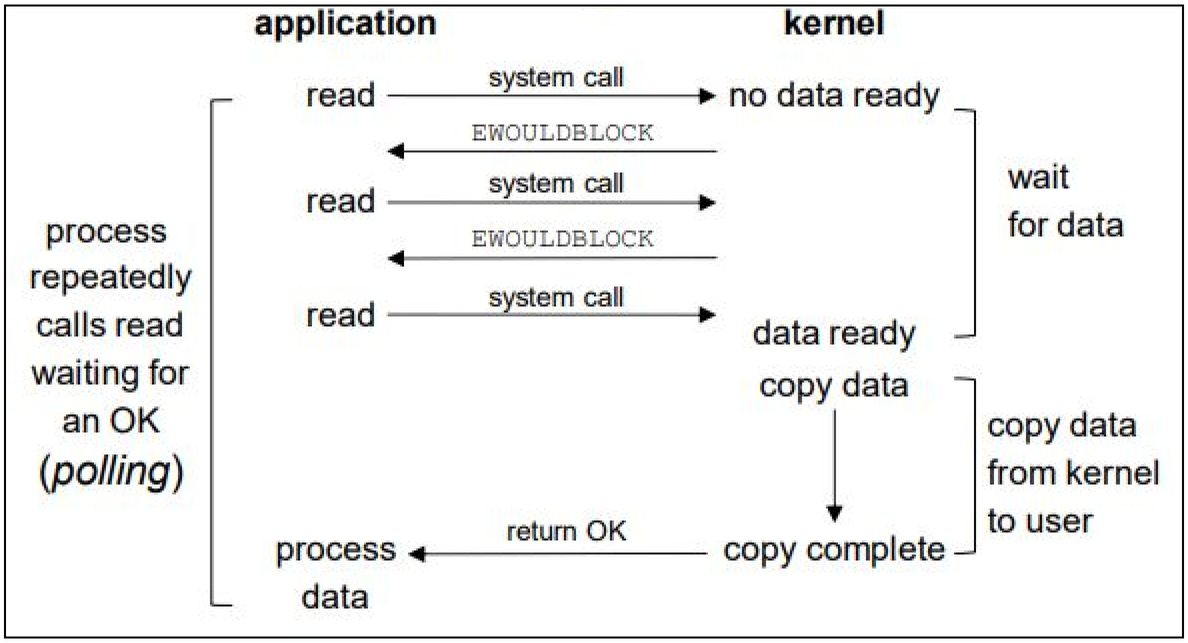


    - **Note:** If you are using sockets, read and accept will both block until the client is done.
    - When reading from multiple sources, blocking on one of the sources could be bad. A possible and simple solution would be to create one process for every client. However, a downside to this is that if there a lot of clients, then a lot of processes have to be made, which is inefficient. I/O multiplexing can solve this problem.
    - There are normally two distinct phases for an input operation:
        1. Waiting for the data to be ready. This involves waiting for data to arrive on the network. When the packet arrives, it is copied into a buffer within the kernel.
        2. Copying the data from the kernel to the process. This means copying the (ready) data from the kernel's buffer into our application buffer.

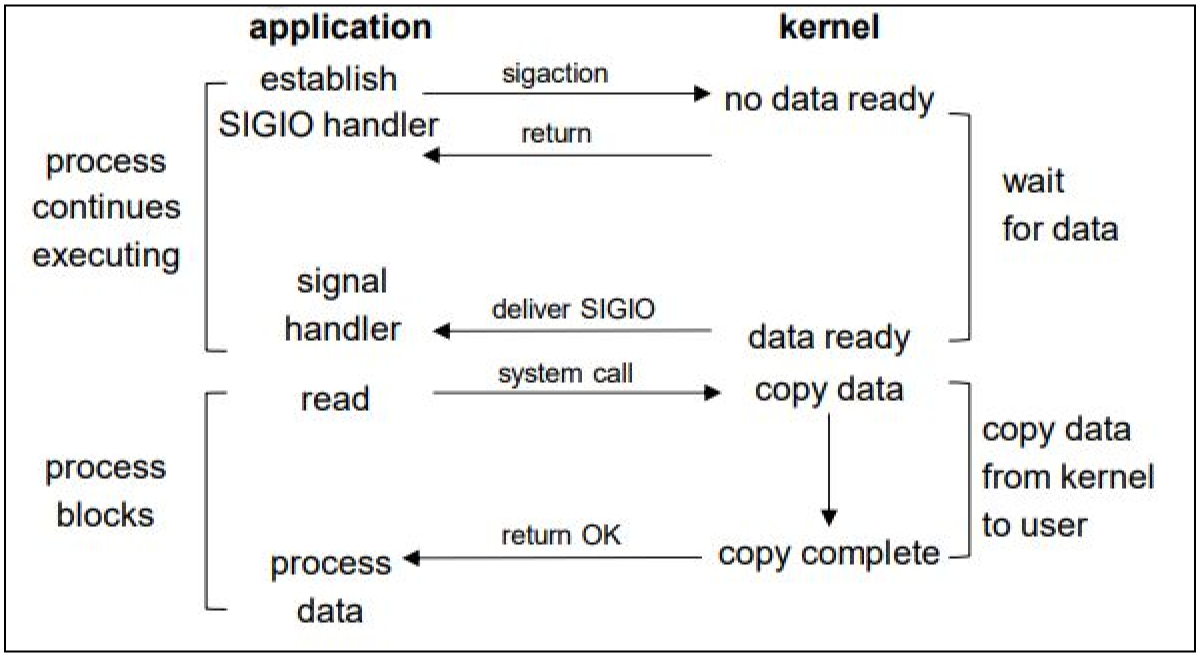
2. **<u>Blocking I/O Model:</u>**
    - The most prevalent model for I/O is the blocking I/O model.
    - By default, all sockets are blocking.



3. **<u>Nonblocking I/O Model:</u>**
    - When a socket is set to be non-blocking, we are telling the kernel "When an I/O operation that I requested cannot be completed without putting the process to sleep, do not put the process to sleep, but return an error instead".

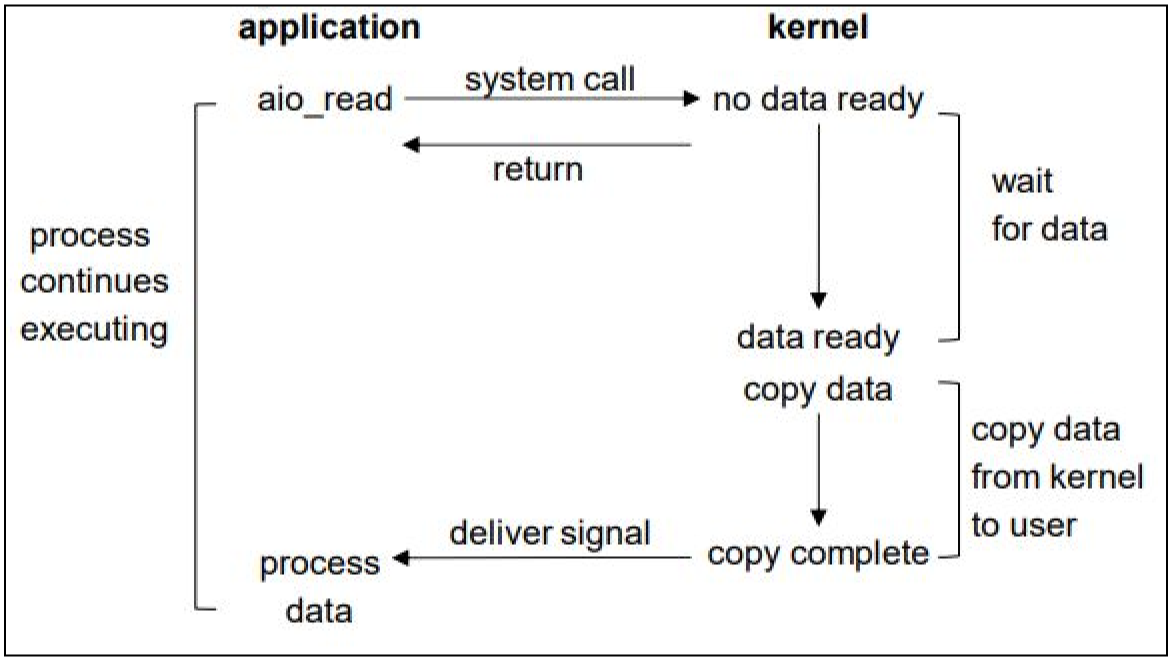


4. **Signal Driven I/O Model:**
    - The signal driven I/O Model tells the kernel to notify us with the SIGIO signal when the descriptor is ready.
    - We first enable the socket for signal-driven I/O and install a signal handler using the sigaction system call. The return from this system call is immediate and our process continues; it is not blocked.
    - When the datagram is ready to be read, the SIGIO signal is generated for our process.
    - The advantage to this model is that we are not blocked while waiting for the datagram to arrive. The main loop can continue executing and just wait to be notified by the signal handler that either the data is ready to process or the datagram is ready to be read.

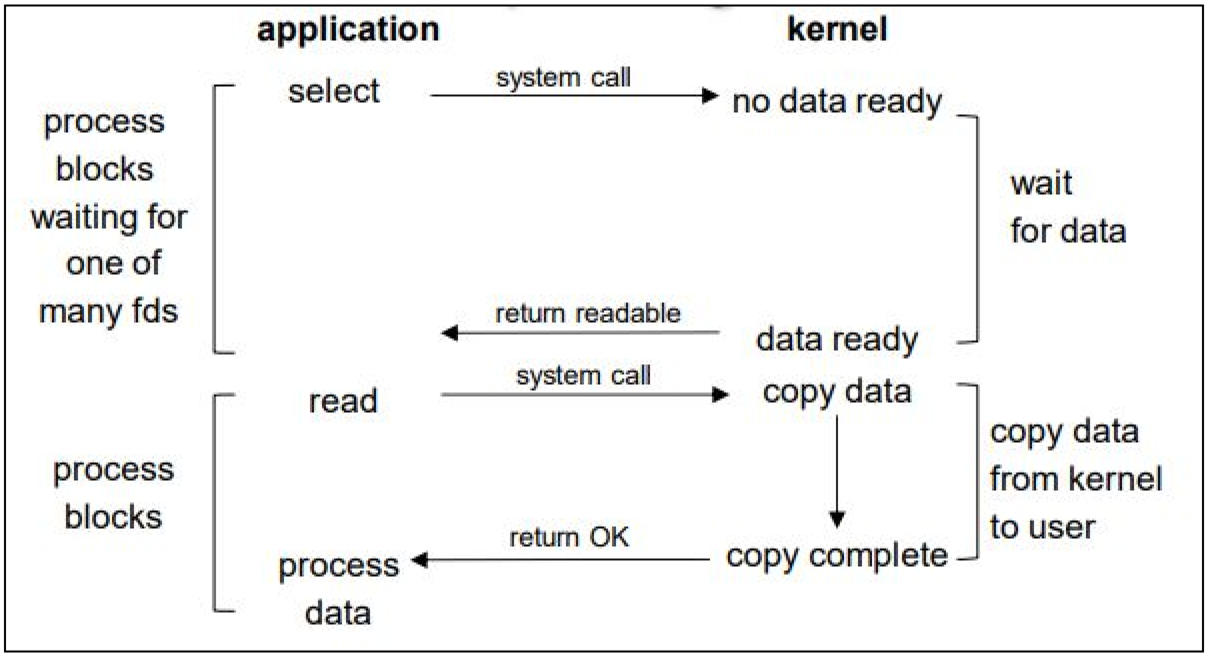


5. **Asynchronous I/O Model:**
    - This works by telling the kernel to start the operation and to notify us when the entire operation,including the copy of the data from the kernel to our buffer, is complete.
    - The main difference between this model and the signal-driven I/O model is that with signal-driven I/O, the kernel tells us when an I/O operation can be initiated, but with asynchronous I/O, the kernel tells us when an I/O operation is complete.
    - We call aio_read and pass the kernel the following:
        - A file descriptor, file offset, buffer pointer and buffer size.
        - How to notify us when the entire operation is complete.
    - This system call returns immediately and our process is not blocked while waiting for the I/O to complete.

Week 11 Notes



6. **I/O Multiplexing Model:**
    - With I/O multiplexing, we call select or poll and block in one of these two system calls, instead of blocking in the actual I/O system call.
    - We block in a call to select, waiting for the datagram socket to be readable. When select returns that the socket is readable, we then call read to copy the datagram into our application buffer.
    - Disadvantage: Using select requires two system calls, select and read, instead of one.
    - Advantage: We can wait for more than one file descriptor to be ready.



7. **Select:**
    - Select allows you to monitor several FDs without using fork or tight looping.
    - Select allows a process to instruct the kernel what descriptors we are interested in (for reading, writing, or an exception condition) and how long to wait.
    - Sometimes a program needs to accept input on multiple input channels whenever input arrives. E.g. A program that acts as a server to several other processes via pipes or sockets. You cannot normally use read for this purpose, because read blocks the program until input is available on one particular file descriptor. A solution is to use select. Select blocks the program until input or output is ready on a specified set of file descriptors, or until a timer expires, whichever comes first.
    - Select blocks until at least one of these FDs has an action.
    - The FDs that remain in the FD sets are the ones on which one can read and write without blocking.
    - Syntax: **int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset, const struct timeval *timeout);**

- The fd_set data type represents file descriptor sets for the select function. It is actually a bit array.
- maxfdp1 specifies the number of descriptors to test. Its value is the maximum descriptor to be tested plus one.
- The reason the maxfdp1 argument exists, along with the burden of calculating its value, is for efficiency. Although each fd_set has room for many descriptors, typically 1024, this is much more than the number used by a typical process. The kernel gains efficiency by not copying unneeded portions of the descriptor set between the process and the kernel, and by not testing bits that are always 0.
- A call to select returns when one of the file descriptors in one of the sets is ready for I/O.
- Select returns the positive number of ready descriptors on success, 0 if we timed out, and -1 on error.
- Select is ready to read when:
    - There is data in the receive buffer to be read.
    - End-of-file state on file descriptor.
    - The socket is a listening socket and there is a connection pending.
    - A socket error is pending and you need to catch it.
- Select is ready to write when:
    - There is space available in the write buffer.
    - A socket error is pending and you need to catch it.
- Select is ready to handle exception conditions when:
    - There is TCP out-of-band data.
- Typically, only the readset is used.
- Timeout specifies how long we're willing to wait for a fd to become ready.
- **struct timeval {**
  long tv_sec; /* seconds */
  long tv_usec; /* microseconds */
  };
- There are 3 cases for timeout:
    - If timeout is NULL, we wait forever or until we catch a signal.
    - If timeout is zero, we test and return immediately.
    - Otherwise, we wait up to specified time.

8. <u>**Descriptor Sets:**</u>
    - Typically implemented as an array of integers where each bit corresponds to a descriptor (except in Windows).
    - Implementation is hidden in the fd_set data type.
    - FD_SETSIZE is the number of descriptors in the data type.
    - Macros:
        - **void FD_ZERO(fd_set *fdset);** //Clears all bits in fdset.
        - **void FD_SET(int fd, fd_set *fdset);** //Turn on the bit for fd in fdset.
        - **void FD_CLR(int fd, fd_set *fdset);** //Turn off the bit for fd in fdset.
        - **int FD_ISSET(int fd, fd_set *fdset);** //Is the bit for fd on in fdset?

- E.g. Consider the code snippet below.
```
fd_set rset;

FD_ZERO(&rset);         // Initialize the set. All bits are off.
FD_SET(1, &rset);       // Turn on bit for fd 1.
FD_SET(4, &rset);       // Turn on bit for fd 4.
FD_SET(5, &rset);       // Turn on bit for fd 5.
```
- Select modifies the descriptor sets pointed to by the readset, writeset, and exceptset pointers.When we call the function, we specify the values of the descriptors that we are interested in, and on return, the result indicates which descriptors are ready. We use the FD_ISSET macro on return to test a specific descriptor in an fd_set structure. Any descriptor that is not ready on return will have its corresponding bit cleared in the descriptor set. To handle this, we turn on all the bits in which we are interested in all the descriptor sets each time we call select.

9. **Bit Strings:**
   - Signal mask and file descriptor sets are implemented using bit array or bit strings.
   - Each bit represents an element of the set:
     - 1 means that the element is in the set.
     - 0 means that the element is not in the set
   - **Bitwise operators:**

Operator and Name	**Description**
& (AND)	It copies a bit to the result if it exists in both operands.
| (OR)	It copies a bit to the result if it exists in either operand.
^ (XOR)	It copies the bit to the result if it is in exactly one operand.
~ (Complement)	Flips all 0's to 1's and vice versa.
<< (Left shift)	All bits are shifted left by the given number.
>> (Right shift)	All bits are shifted left by the given number.

   - **Note:** Left and right shift will cause bits to fall off the ends.
   - E.g. Suppose that A is 0011 1100 and B is 0000 1101. Then:
     - (A&B) is 0000 1100
     - (A|B) is 0011 1101
     - (A^B) is 0011 0001
     - (~A) is 1100 0011
     - (A << 2) is 1111 0000
     - (B >> 3) is 0000 0001

Week 11 Notes

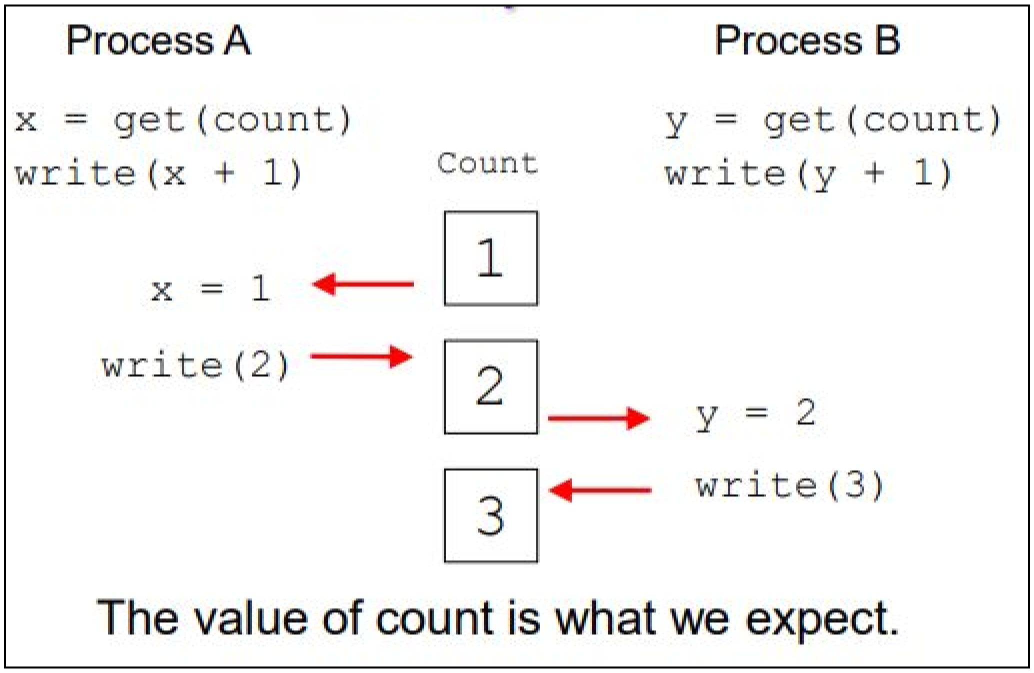
- Precedence of bit operators:
  Highest: ~
  &
  ^
  Lowest: |
- Flags are often implemented as a bit mask.
  E.g.
  ```
  open("temp", O_WRONLY | O_CREAT);
  #define O_RDONLY  00
  #define O_WRONLY 01
  #define O_RDWR     02
  #define O_CREAT 0100
  ```
- FD_SETSIZE is bigger than 32.

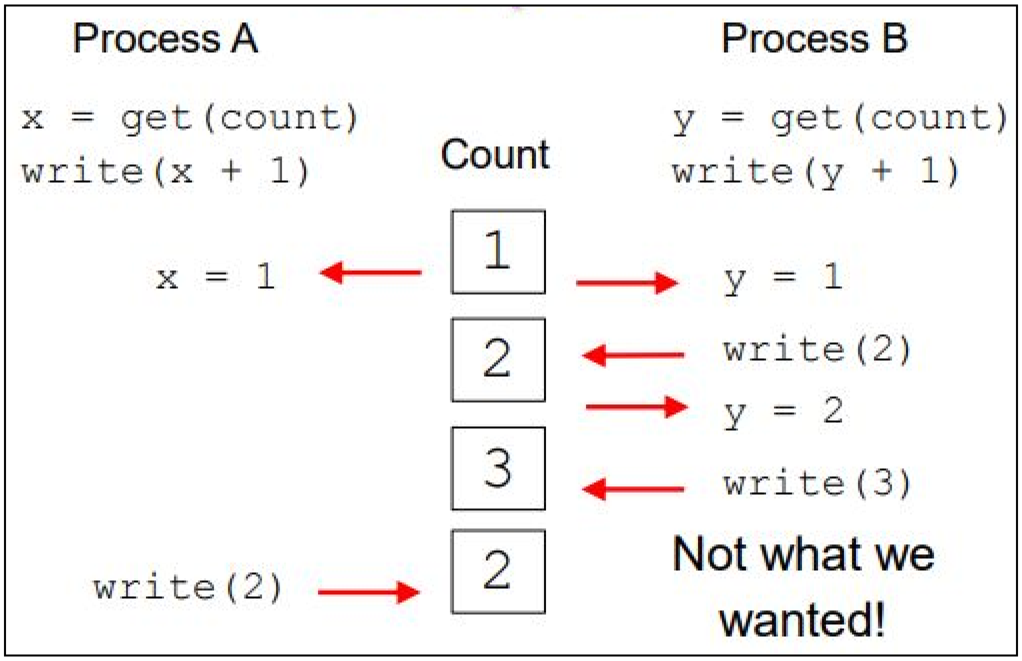
-
  ```
  struct bits {
          unsigned int field[N];
  }
  typedef struct bits Bitstring;
  Bitstring a, b;
  setzero(&a);
  b = a;
  a.field[0] = ~0;
  ```
- The set function sets one bit to 1. This is predefined in the C library.
  ```
  int set(unsigned int bit, Bitstring *b) {
          int index = bit / 32;
          b->field[index] |= 1 << (bit % 32);
          return 1;
  }
  ```
- The unset function sets one bit to 0. This is predefined in the C library.
  ```
  int unset(unsigned int bit, Bitstring *b) {
          int index = bit / 32;
          b->field[index] &= ~(1 << (bit % 32));
  }
  ```
- The ifset function checks if a bit is set in the bit string. This is predefined in the C library.
  ```
  int ifset(unsigned int bit, Bitstring *b) {
          int index = bit / 32;
          return ( (1 << (bit % 32)) & b->field[index]);
  }
  ```
- The setzero function sets all bits to 0. This is predefined in the C library.
  ```
  int setzero(Bitstring *b){
          if(memset(b,0, sizeof(Bitstring)) == NULL)
                  return 0;
          else
                  return 1;
  }
  ```

- The intToBinary function converts an int to binary. This is predefined in the C library.

```
char *intToBinary(unsigned int number) {
        char *binaryString = malloc(32+1);
        int i; binaryString[32] = '\0';
        for (i = 31; i >= 0; i--) {
                binaryString[i] = ((number & 1) + '0');
                number = number >> 1;
        }
        return binaryString;
}
```

10. **Concurrency:**
    - The two key concepts driving computer systems and applications are:
        - **Communication:** The conveying of information from one entity to another.
        - **Concurrency:** The sharing of resources in the same time frame.
    - Concurrency can exist in a single processor as well as in a multiprocessor system.
    - Managing concurrency is difficult, as execution behaviour is not always reproducible.
11. **Race conditions:**
    - A **race condition** occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run.
    - E.g.

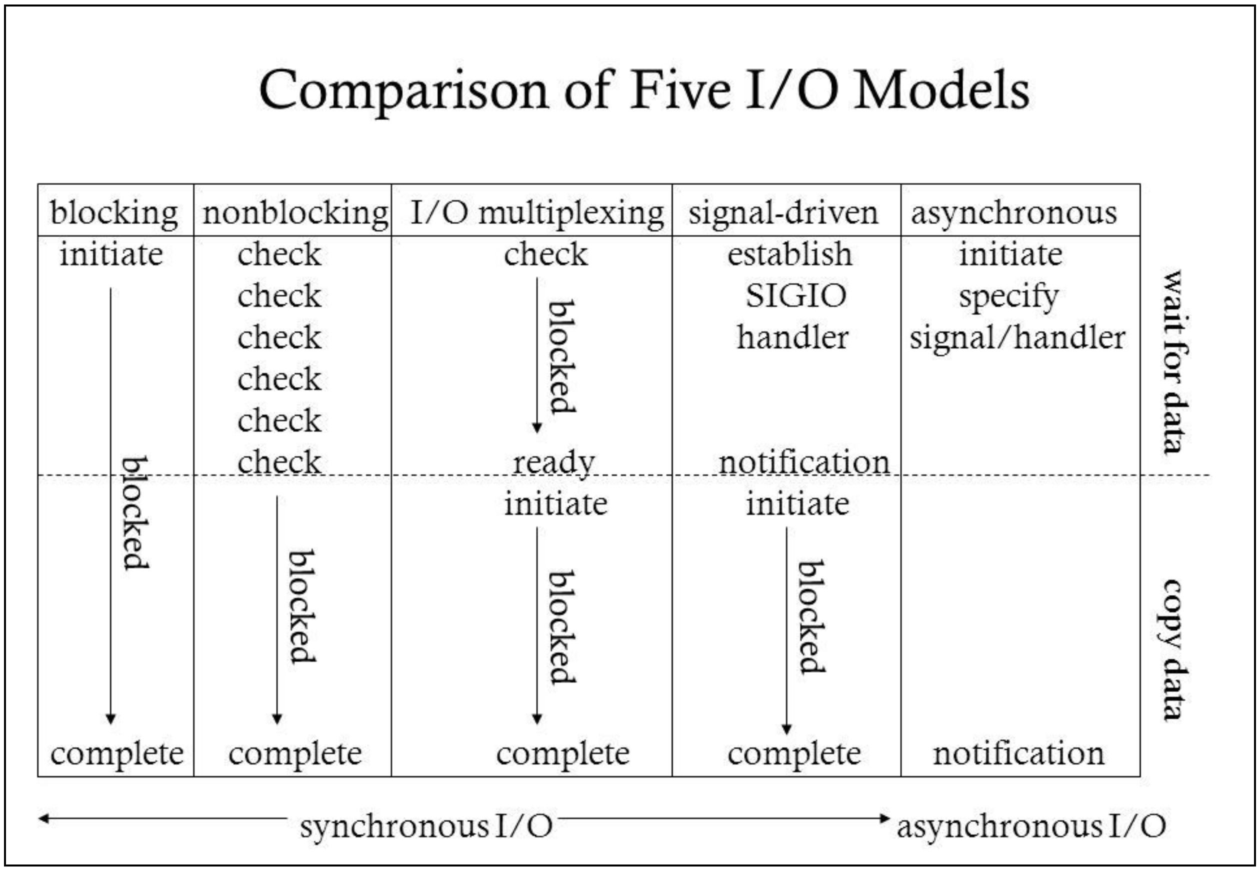




12. **Producer/Consumer Problem:**
    - The scenario goes as follows:
        - Suppose there are two processes, the producer and the consumer, who share a common, fixed-size buffer. The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming (removing) the data, one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.
    - The consumer should be blocked when buffer is empty.
    - The producer should be blocked when buffer is full.
    - The producer and consumer should run independently as far as buffer capacity and contents permit.
    - The producer and consumer should never be updating the buffer at the same time (otherwise data integrity cannot be guaranteed).
    - The producer/consumer is a harder problem if there are more than one consumer and/or more than one producer.
    - Programs that manage shared resources must protect the integrity of the shared resources.
    - Operations that modify the shared resource are called critical sections.
    - Critical section must be executed in a mutually exclusive manner.
    - Semaphores are commonly used to protect critical sections.

13. <u>Semaphores:</u>

- Code that modifies shared data usually has the following parts:
    - **Entry section:** The code that requests permission to modify the shared data.
    - **Critical Section:** The code that modifies the shared variable.
    - **Exit Section:** The code that releases access to the shared data.
    - **Remainder:** The remaining code.
- Useful in process synchronization and multithreading.
- The way semaphores work is that:
    1. You call a function, acquire.
       acquire(v): Blocks until the value of the semaphore variable v until it is greater than 0 then it decrements v.
    2. You call a function, release.
       release(v): Increments v.
- The structure of how semaphores work is that:
    1. Create a semaphore variable v.
    2. init(v)
    3. acquire(v)
    4. Critical section of code.
    5. release(v)
    6. Remainder of code.



List of Unix Commands:

Note: Some of these commands may have extensions/options. Use the man command to read more about them.

Commands	**What it does**
bg job	Puts job in the background.
cat file1	Displays the content of file1.
cd dir	Changes directory from the current working directory to dir.
chmod (option) file1	Lets you change the read, write, and execute permissions on file1.
clear	Clears the terminal.
cp file1 file2	Creates a copy of file1 called file2.
cp file1 dir1	Copies file1 into dir1.
cp -r dir1 dir2	Copies dir1 into dir2
echo string	Prints string on the terminal.
fg job	Puts job in the foreground.
grep pattern file1	Searches file1 for pattern, and displays all lines that contain pattern.
head (option) (num) file1	Prints the first 10 lines of file1, by default. If you want to set the number of lines this prints, you need to do: head -n num file1 and it will print the first num lines of file1. E.g. head file1 will print the first 10 lines of file1. E.g. head -n 3 file1 will print the first 3 lines of file1.
history	Prints a list of all past commands typed in the current terminal session.
less file1	Displays the contents of file1. Less is a similar to more, but which allows backward and forward movement in the file. Furthermore, less doesn't have to read the entire input file before starting, so it is faster than some text editors for large input files.
ln target link	This creates a hard link between target and link.
ln -s target link	This creates a soft link between target and link.
ls	Lists all files and directories in the current directory.

ls - R	Lists files in the sub-directories, as well
ls - a	Lists hidden files as well
ls - al	Lists files and directories with detailed information like permissions, size, owner, etc.
jobs	Gives you a list of jobs, each of which is associated with a job number.
man command	Gives help information on command.
mkdir directory	Creates a new directory in the current working directory or at the specified path.
more file1	Displays the contents of file1.
mv file1 file2	Moves file1 into file2. If file2 doesn't exist, then it renames file1 to file2.
ps	Prints the current running processes.
pwd	Prints the absolute path of the current working directory.
rm file1	Deletes file1.
rmdir directory1	Deletes directory1.

sort file1	This rearranges the lines in file1 so that they are sorted, numerically and alphabetically. By default, the rules for sorting are: 1. Lines starting with a number will appear before lines starting with a letter. 2. Lines starting with a letter that appears earlier in the alphabet will appear before lines starting with a letter that appears later in the alphabet. 3. If two or more lines, of different lengths, share the first x number of letters, then sort will print the lines from shortest length to longest. 4. Lines starting with a lowercase letter will appear before lines starting with the same letter in uppercase, if they have the same length. **Note:** sort does not affect the original file in any way. E.g. Suppose we have a file called file1 that contains these lines my name is rick My name is rick 1 Then, if we do sort file1, we get the following: 1 my name is rick My name is rick E.g. Suppose we have a file called file2 that contains these lines my name is rick lan My name is rick 1 Then, if we do sort file2, we get the following: 1 My name is rick my name is rick lan
stat file1/dir1	Gives detailed information about file1/dir1
tail (option) (num) file1	Prints the last 10 lines of file1, by default. If you want to set the number of lines this prints, you need to do: last -n num file1 and it will print the last num lines of file1. E.g. last file1 will print the last 10 lines of file1. E.g. last -n 3 file1 will print the last 3 lines of file1

uniq file1	Prints out the contents of file1 but removes all the duplicates in adjacent lines. E.g. Suppose we have a file called file1 and it contains the following: cat cat cat dog dog Then, if we do uniq file1, it would print cat dog E.g. Suppose we have a file called file2 and it contains the following: cat dog cat dog cat Then, if we do uniq file2, it would print cat dog cat dog cat
vi file1	Creates a file called file1.
wc file1	Prints the number of lines, words, and characters file1 has. E.g. Suppose we have a file called file1 and it contains the following: cat dog cat dog cat Then, if we do wc file1, we get: 5 5 20 file1 The first 5 means there are 5 lines in file1. The second 5 means there are 5 words in file1. The 20 means there are 20 characters in file1. Note that because there is a space after each line, there are 20 characters.

Redirection:

1. **Standard Input (stdin):**
    - The file descriptor for stdin is 0.
    - Denoted as <.
    - E.g. cat < file1 works the same as cat file1.
2. **Standard Output (stdout):**
    - The file descriptor for stdout is 1.
    - Denoted as either > or >>.
    - The general form is **command > file** or **command >> file**, where **command** is any command that outputs to stdout.
    - **Note:** If **file** was not created before, then this will create it for you.
    - > will redirect the output to a file. This will overwrite whatever was in the file before.
    - >> will append the output to a file. This will add the output to the end of whatever was in the file before.
    - E.g. Suppose we have 2 files, file1 and file2.
      Suppose the contents of file1 is 12345 and the contents of file2 is abcde.
      If we do cat file1 >> file2 and then do cat file2, the output of cat file2 is:
      abcde
      12345
      If we do cat file2 > file1 and then do cat file1, the output of cat file1 is:
      abcde
      12345
3. **Standard Error (stderr):**
    - The file descriptor for stderr is 2.
    - Denoted as 2> or 2>>.
    - The general form is **command error 2> file** or **command error 2>> file**.
    - **Note:** If **file** was not created before, then this will create it for you.
    - 2> will redirect the error message into the file.
    - 2>> will append the error message into the file.
    - This is useful for shell scripting because you usually do not want error messages cluttering up the normal program output.
    - E.g. Suppose that there is no file in the current working directory called x. Suppose we do cat x 2> file1. This will overwrite the contents of file1 with the error message.
    - E.g. Suppose that there is no file in the current working directory called x. Suppose we do cat x 2>> file1. This will append the contents of file1 with the error message.