

Systems Software/Programming Introduction

What is systems software/programming?

- Programming where software is written to operate the hardware or interface with operating system
- Software produced by systems programming provides services to other software rather than providing services to end user in case of application software
- Where can we use systems software/programming? **Its all around us**
 - Operating Systems, Device Drivers
 - Game Engines – Xbox, PlayStation etc
 - Industrial Automation – Robotics, Control Systems, Monitoring Systems etc
 - Cloud – Infrastructure As A Service (IAAS), Platform As A Service (PAAS), Software As A Service (SAAS)
 - Internet of Things (IOT): Traffic Control Automation, Smart House, Air/Water Quality Control, Farming, Smart Grid/Meters and many more...

Difference between Application and Systems Software

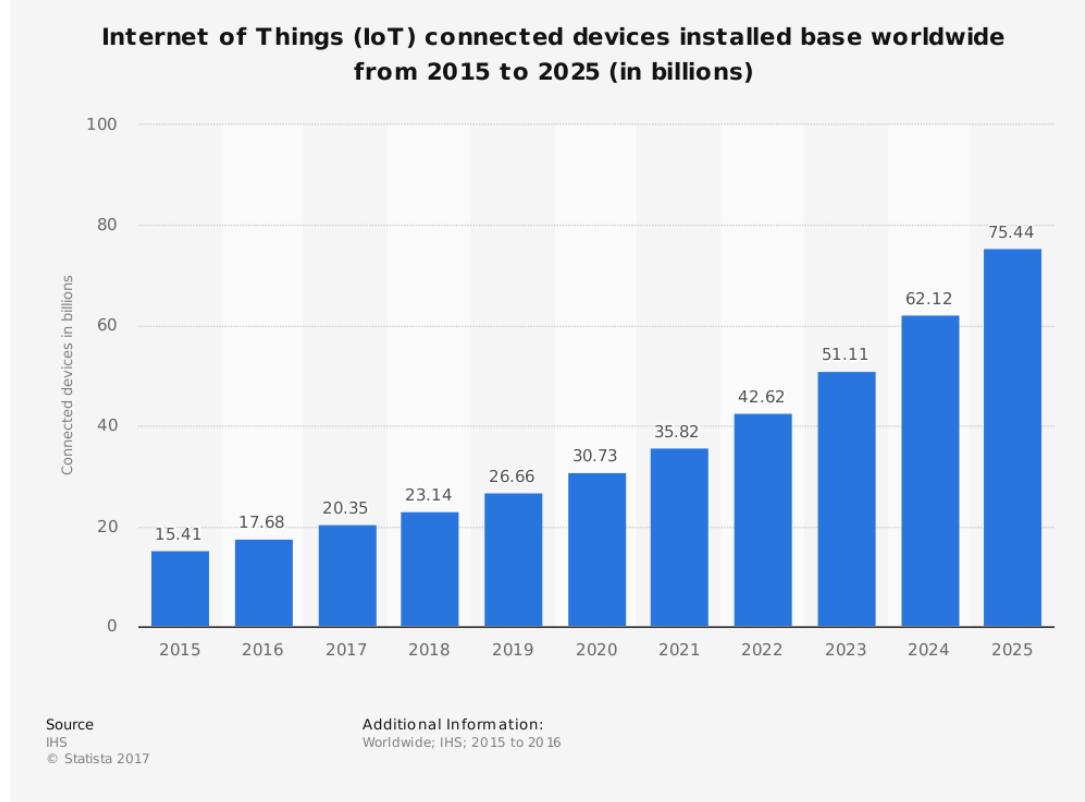
System Software

- System focused
- Requires detail knowledge of hardware and OS
- Provides services to other software such as application s/w
- Used to program hardware or interfaces with operating system
- Low level languages such as C/C++ and Assembly are used

Application Software

- User focused
- Does not require hardware knowledge
- Provides services to end user
- Used to provide functionality for a specific user application
- High level languages C#, Java, Python, HTML and so on are used

Why Systems Software/Programming is important for Internet of Things (IOT)?

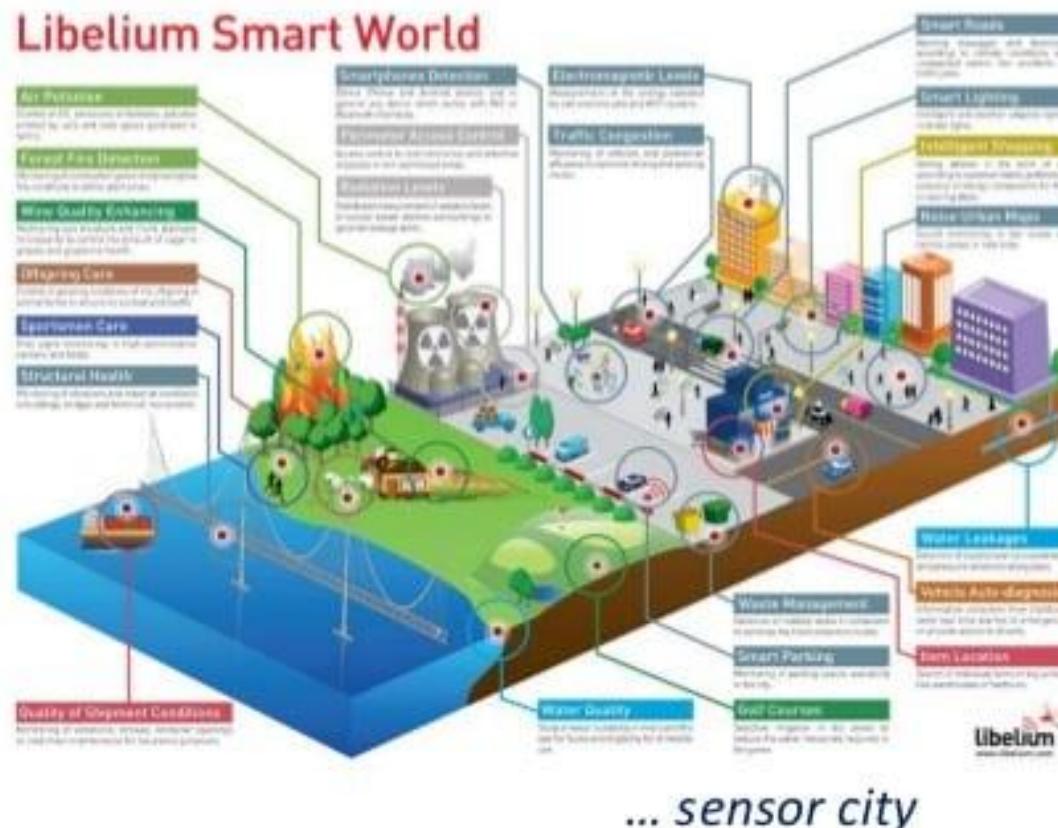


- Every device will need OS or Embedded Software to operate the hardware
- Almost every devices will require communication interface to share information with other devices
- Many devices will require sensing unit to be programmed
- Many devices will require logic to make decision based on the information received
- Many devices will require controller programmed to control the specific hardware unit

Source: <https://easternpeak.com/blog/iot-ideas-for-business/>

How IOT will be used for Smart City?

The Future is Now *- Perspectives of a Smart City*



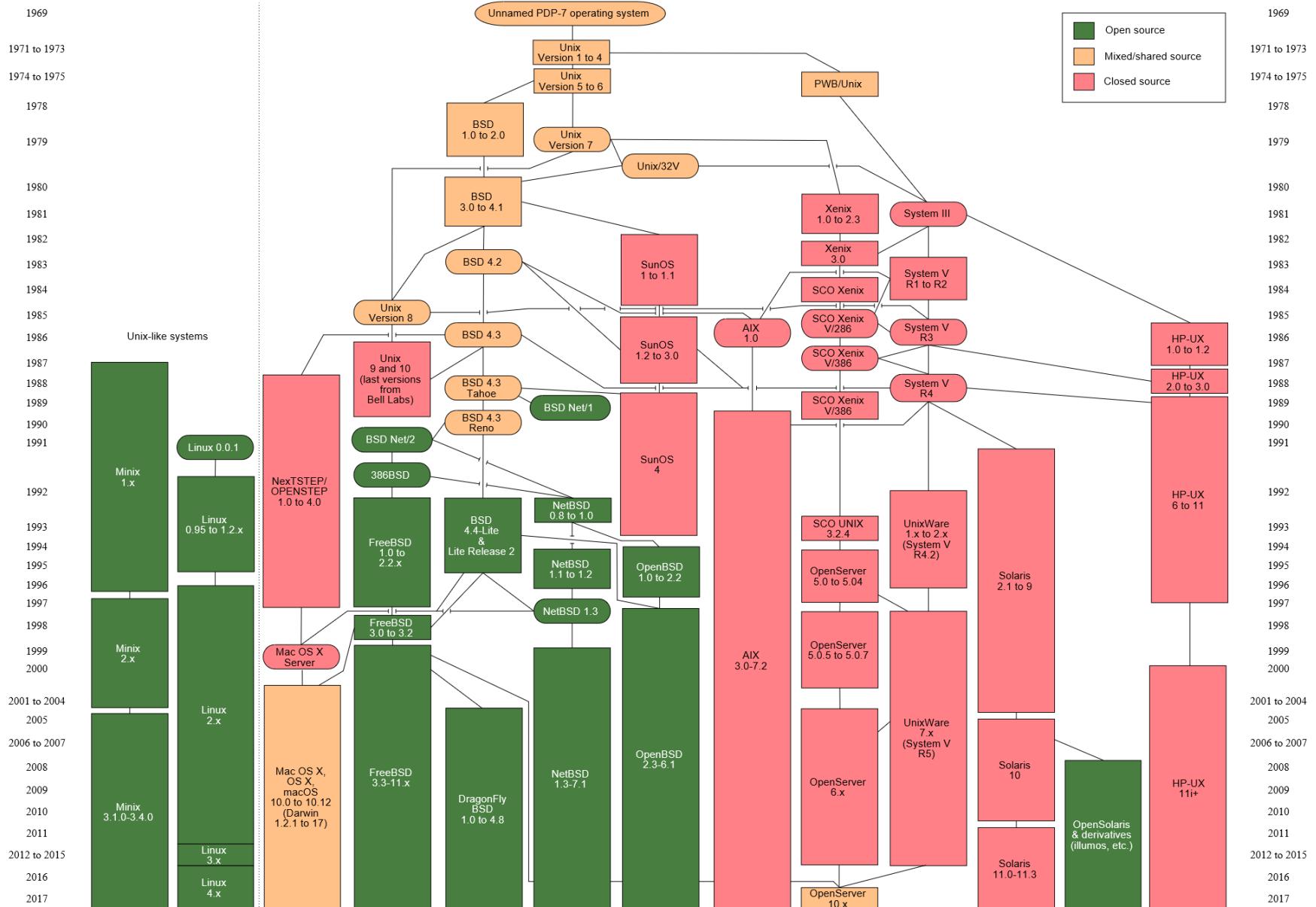
Jury Konga
eGovFutures Group

Source: <https://www.slideshare.net/JuryKonga/geosmart-cities-2020-beyond>

Sensors for

- Air pollution
- Fire detection
- Water quality
- Smart parking
- Traffic congestion
- Waste management
- Golf course conditions

We will be using Unix/Linux OS, why ?



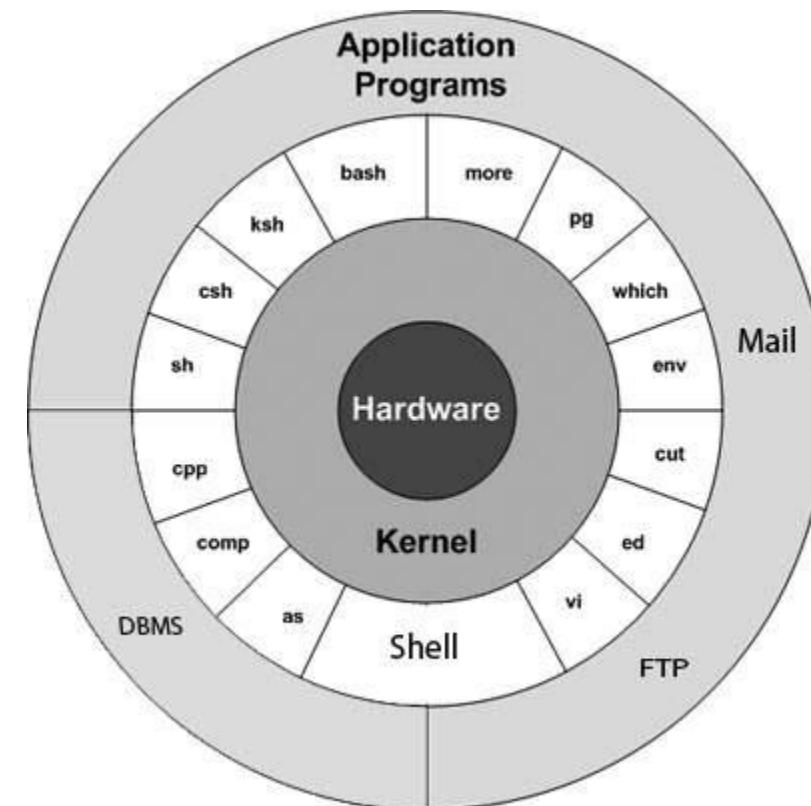
Source: https://en.wikipedia.org/wiki/UNIX_System_V

Unix flavors used today

- Apple iOS based on Darwin BSD, macOS is Unix-like
- Android based on Linux which is Unix like
 - Supported on many platforms :
 - DEC Alpha servers
 - ARM based embedded systems, mobile phones, tablets etc
 - Intel Skylake, Westmere, Nehalem, Xeon for desktops, laptops and servers
 - Apple-IBM-Motorola alliance PowerPC
 - Sun Microsystem (now acquired by Oracle corp) SPARC
- Playstation OS based on FreeBSD
- Ubuntu based on Linux
- Cloud OS is Unix-like
- And many more....

Unix/Linux Architecture

- Kernel – Heart of OS, interacts with hardware and performs tasks like memory management, process/thread/task scheduling, file management etc.
- Shell – Command line interface to run shell commands using variants C Shell, Bourn Shell, Korn Shell
- Commands and Utilities – 250+ standard commands and 3rd party utilities



Unix/Linux Basics

- How to logon?
 - Local computer → User ID and Password (Password is Case Sensitive)
 - Remote computer → Using ssh (Secure Shell) → userid@hostname or userid@IPAddress, enter password when prompted. You can also use ssh clients such as putty or MobaXTerm
- Current logged in user : **whoami command**
- Change Password: **passwd command**
- Getting help for commands and system functions: **man command**
- Text editors: **gedit** (GUI based), vi, vim, nano (non GUI)
- Compilers: **gcc for c/c++**, gfortran for fortran etc (**we will be using gcc so get familiarity with it**)

Vi editor regularly used commands

- vi filename
- a: enter insert mode, after the cursor
- i: enter insert mode, before the cursor
- O: enter insert mode, above the cursor
- o: enter insert mode, below the cursor
- r: replace one character under the cursor
- u: undo the last change to the file.
- x: delete character under the cursor
- yy: copy line
- dd: delete line
- :w: write
- :q: quit
- :q!: quit without saving changes
- /keyword : search for the keyword in text
- :n : go to line number n
- Vi tutorial:
<http://www.gnulamp.com/vi.html>

Files and Directories

Directory Operations

- `ls` – list content of directory
- `cd` – change directory
- `pwd` – present working directory
- `mkdir` – create new directory
- `rmdir` – Remove/delete existing directory (and subdirectories)
- Single dot `(.)` and double dots `(..)` represents current directory and immediate parent directory respectively

File Operations

- `cp` – copy a file and/or directory
- `scp` – secure copy files and/or directory to another machine using ssh (i.e. file transfer)
- `mv` – rename a file and/or directory or move file from one directory to another
- `rm` – remove/delete file
- `cat, head, tail, more, less, wc` : explore yourself

Files and Directories

- File Permissions: `rwx` (read, write and execute) for user, group and others e.g.
 - `-rwxrw-r-- 1 linaro linaro 4044 Aug 29 08:13 /home/linaro/.bashrc`
 - Indicates linaro user can read, write and execute, users from group linaro can read and write but can't execute; and other users who are not in linaro group can only read
- Change file permission: `chmod` command (using bit pattern e.g. 766 indicates `rwx` for user, `rw-` for group and `rw-` for other user)
- Change file ownership : `chown` command
- Executing commands as superuser (mostly root): `sudo` command

Standard Unix file streams

- Standard Input **STDIN** with file descriptor **0**
- Standard Output **STDOUT** with file descriptor **1**
- Standard Error **STDERR** with file descriptor **2**
- Redirection of standard output **>** or **>>** (**>** : create file, **>>** : append to existing file)
 - `ls -ltr > dirlist.txt`
 - `ls -ltr >> dirlist.txt`
- Redirection of standard input **<**
 - E.g. `cat < dirlist.txt`

Processes

- `ps` – lists currently active user processes
- `ps -aux` or `ps -ef` – lists all active processes in long format
- `kill n` – kill process with process id = n
- `kill -9 n` – force to kill process id = n
- Keys `CTRL-z` – force process to move to background
- `fg` – bring process to foreground
- `top` – provides system utilization information
- `time` – calculate time for a given command

Other Unix/Linux commonly used commands

- **awk** – pattern scanning and processing language

Print the total number of kilobytes used by files in the current directory

```
ls -l . | awk '{ x += $5 } ; END { print "total kilobytes: " (x /1024) }'
```

Print sorted list of login names

```
awk -F: '{ print $1 }' /etc/passwd | sort
```

Print only elements from column 2 that match pattern using stdin

```
awk '/pattern/ {print $2}'
```

- **grep** – search for files with a given pattern

Search /etc/passwd file for the user harry

```
grep harry /etc/passwd
```

Search all files in current directory and subdirectory for expression “harry potter” with ignore case

```
grep -r -i "harry potter" /home/joe/
```

-r : for recursive search
-i : for ignoring case

Other Unix/Linux commonly used commands

- **find** – search for files in directory hierarchy
- sed – stream editor for filtering and transforming text

Replace every occurrence of Nick with John in report.txt

```
sed 's/Nick/John/g' report.txt
```

Add 8 spaces to the left of a text for pretty printing.

```
sed 's/^/ /' file.txt >file_new.txt
```

Write all commands in script.sed and execute them

```
sed -f script.sed file.txt
```

Print only lines with three consecutive digits

```
sed '/[0-9]\{3\}/p' file.txt
```

Other Unix/Linux commonly used commands

- `cut` – remove section from each line of file
- `ping` – sends echo request to `host/ip` address
- `telnet/putty` – tool used to connect to remote host using ssh from non-unix based machine
- `uname` – print system information
- `lsb_release` – print distribution specific information
- `locate` – Locate files by name

What is profile?

- For each type of shell when you login default profile file gets executed
- e.g. for Bourne Again Shell (bash)
 - .bashrc file in HOME folder (first dot in the file name indicates hidden file) gets executed

Systems
Software/Programming
Shell Scripting

Types of Shell

- Bourne Shell – with \$ as default prompt
 - Bourne Shell (sh)
 - Korn Shell (ksh)
 - Bourne Again Shell (bash)
 - POSIX Shell (sh)
- C-type Shell – with % as default prompt
 - C Shell (csh)
 - TENEX/TOPS C Shell (tcsh)

Shell Scripts

- All shell scripts has extension .sh (e.g. test.sh)
- Use shebang (#!) to indicate which shell you are using for shell script e.g. test.sh will contain

```
#!/bin/bash
```

```
pwd
```

```
ls
```

- Use # for single line comments

```
#!/bin/bash
```

```
# Author : Amit
```

```
# Copyright (c)
```

```
# Script follows here:
```

Variables

- Shell Variables
 - Environment Variables
 - SHELL=/bin/bash
 - PATH=/usr/bin/
 - USER="logged in user"
 - HOME="path to home folder"
 - Many others...
 - Local Variables
 - Any variable that is defined in shell script by developer e.g. AUTHOR="Amit"
 - Access variables using \$ sign e.g. echo \$PATH will display the value stored in environment variable PATH

Special Variables

- **\$\$** - process id (PID) of current shell
 - **\$0** - shell script filename being executed
 - **\$1, \$2, \$3...\$n** – command line argument
 - **\$#** - number of command line argument
 - **\$?** – exit status of last command executed
 - 0 : success
 - 1 : error (all errors w/o specific code)
 - 130 terminated by Ctrl+C
 - **\$!** – PID of last background command
- echo command is used to print value of any constant or variable
[ShellScripts\test.sh](#)
Now run: test.sh abc xyz
Output will be:
File name: test.sh
First command line argument is: abc
Second command line argument is: xyz
- Execute C program and read the return status
[ShellScripts\retstat.sh](#)

Arrays

Definition

NAME[0] = "Deepak"

NAME[1] = "Renuka"

NAME[2] = "Joe"

NAME[3] = "Alex"

NAME[4] = "Amir"

Or in bash

array_name = (value1 ... valuen)

Accessing

- To display value from a specific index

```
echo "First Index: ${NAME[0]}" echo  
"Second Index: ${NAME[1]}"
```

- To display all values

```
echo "All Index: ${NAME[*]}"
```

OR

```
echo "All Index: ${NAME[@]}"
```

[ShellScripts\array_test.sh](#)

Operators

- Arithmetic Operators: `+ - * / % = == !=`
 - `c=`expr $a + $b`` add values from a and b and assign it to c
 - `a=$b` would assign value of b into a
 - `[$a == $b]` OR `[$a != $b]` would compare numeric values of a and b
- Relational Operators: `-eq -ne -gt -lt -ge -le`
- Boolean/Logical Operators: `! -o -a`
- String Operators: `-z -n`
 - `-z` (or `-n`) returns true if string length is zero (or non-zero)

Operators

- File Test Operators (assuming file variable holds the filename)
 - `-d file`: true if file is a directory
 - `-f file`: true if ordinary file instead of directory or special file
 - `-r file`: true if file is readable
 - `-w file`: true if file is writable
 - `-x file`: true if file is executable
 - `-s file`: true if file size > 0
 - `-e file`: true if file exists

Test if file exists

[ShellScripts\file_test.sh](#)

Test if file exist and display the content

[ShellScripts\file_read.sh](#)

Decision Making Conditional Statements

- If...fi statement
 - If...else...fi statement
 - If...elif...else...fi statement

case word in

pattern1)

Statement(s) to be executed if pattern1 matches ::

pattern2)

Statement(s) to be executed if pattern2 matches ::

pattern3)

Statement(s) to be executed if pattern3 matches ::

*

```
Default condition to be  
executed ;;  
esac
```

Loops – Nested Loops are Allowed

- While Loop:

```
while command
```

```
do
```

Statement(s) to be executed if command is
true

```
done
```

- Until Loop:

```
until command
```

```
do
```

Statement(s) to be executed until command is
true

```
done
```

- For Loop:

```
for var in word1 word2 ... wordN
```

```
do
```

Statement(s) to be executed for every word.

```
done
```

- Select Loop (Used for creating Menu):

```
select var in word1 word2 ... wordN
```

```
do
```

Statement(s) to be executed for every word.

```
done
```

[ShellScripts\for select test.sh](#)

Loop Control Statements

- break statement

```
#!/bin/sh a=0
while [ $a -lt 10 ]
do
    echo $a
    if [ $a -eq 5 ]
    then
        break
    fi
    a=`expr $a + 1`
done
```

- continue statement

```
#!/bin/sh
NUMS="1 2 3 4 5 6 7"
for NUM in $NUMS
do
    Q=`expr $NUM % 2`
    if [ $Q -eq 0 ]
    then
        echo "Number is an even
number!!"
        continue
    fi
    echo "Found odd number"
done
```

Substitution

- Escape sequences with echo command
 - \\ (backslash)
 - \b (backspace)
 - \c (suppress trailing newline)
 - \f (form feed)
 - \n (new line)
 - \r (carriage return)
 - \t (horizontal tab)
 - \v (vertical tab)
- For echo command, use -e (-E) option to enable (disable) interpretation of backslash escapes
- [ShellScripts\substitution.sh](#)

- Command Substitution using `command`

```
DATE=`date`  
echo "Date is $DATE"
```

```
USERS=`who | wc -l`  
echo "Logged in user are $USERS"
```
- Variable Substitution
 - \${var}
 - \${var:-defaultvalue} → defaultvalue is NOT assigned to var
 - \${var:=defaultval} → defaultvalue is assigned to var
 - \${var:?message} → display a message on empty

Input and Output Redirection

- Command/Program > file '`>`: STDOUT
 - Any output from command or program execution will be saved in file instead of displaying to STDOUT
 - New file will be created if does not exist or existing file will be erased first
- Command/Program >> file :
 - Any output from command or program execution will be appended to an existing file instead of displaying to STDOUT
 - New file will be created if does not exist but if file already exists then it is appended
- n >> file : output from stream with descriptor n is appended to a file
- n >& m : merges output from stream n with stream m
[ShellScripts\merge_stdout_stderr.sh](#)
- Command/Program < file : Input to the command or program is fed from data in file '`<` : STDIN
- | (called pipe) : Takes output from one process and feed into another process

Functions

```
# Define your function here
Hello () {
    if [ $# > 1 ]
    then
        for param in $*
        do
            echo $param
        done
    fi
    return $#
}
```

create array of command line arguments

```
# Invoke your function
Hello $*
# Capture value returned by last
command
ret=$?
ShellScripts\function.sh
```

Note: Nested functions and recursive
functions are allowed

Functions can be accessed in shell
prompt by placing them in .sh file and
executing that .sh file in a shell prompt

Exercises

- Print sum of all command line integer arguments
- Print the factorial of a given number using fact() function
- Print the day of the week for all the command line values provided between 1 and 7
- Given path (e.g. /usr/local/lib or /etc/passwd), first check if that path exists and then check if it is a file or a directory and print appropriate message

Print sum of all command line arguments

- [ShellScripts\sum_args.sh](#)

Quiz – Write a shell script

1. Multiple logs files with filenames *.log are created while compilation of large application such as python from source. Task is to concatenate all the *.log files and create a new file warnings.log with all the lines where warning has occurred (i.e. lines with ‘warning’ word)
2. User provided directory as command line argument has multiple program files with filenames *.out. Task is to execute each program and store exit codes in an array.

Systems
Software/Programming
C Programming Revision

Why C for Systems Software/Programming ?

- Unix/Linux OS has been written in C/C++, hence call to system functions is easier in C.
- C can perform memory management through dynamic memory allocation.
- C can implement different data structures efficiently using structures and pointers such as stack, queues, linked list, trees etc.
- Using C, hardware systems can be programmed directly.
- C can call assembly routines and assembly code can call C routines on many processors if required.

C – Primitive Data types

- **char** : character - **1 byte**
- **short**: short integer - **2 bytes**
- **int**: integer - **4 bytes**
- **long**: long integer - **4 bytes**
- **float**: floating point - **4 bytes**
- **double** - double precision
floating point - **8 bytes**
- **%d**: integers
- **%f**: floating point
- **%c**: characters
- **%s**: string
- **%x**: hexadecimal
- **%u**: unsigned int

Operators

- Arithmetic: + - * %
- Relational: > >= < <= == !=
- Logical: && || !
- Increment and Decrement: ++ and -- (pre and post)
- Bitwise: & | ^ << >> ~
- Assignment: = += -= /= *= %= &= |= ^= <<= >>=
- Ternary: ?:
- Special: .(dot) -> (structure member access using pointer)

Constants, Macros and Variables

```
#define COURSE "Systems  
Programming"
```

```
#define NO_OF_STUDENTS 120
```

```
#define max(a,b) a>b?a:b
```

```
int i;  
float f = 10.8f;  
double d = 17.8;  
char c = 65;
```

C Statements

- Conditionals
 - IF, IF-ELSE, Nested IF-ELSE
 - switch..case...default
- Loops
 - `for(initialization;condition;iteration)`
 - while
 - do..while
 - break
 - Continue
 - goto

Arrays

- Defining an array is easy:

```
int a[3]; /* a is an array of 3 integers */
```

- Array indexes go from 0 to n-1:

```
a[0] = 2; a[1] = 4; a[2] = a[0] + a[1];
int x = a[a[0]]; /* what is the value of x? */
```

► **Beware:** in this example a[3] does not exist, but your compiler will not complain if you use it!

★ But your program may have a very strange behavior...

- You can create multidimensional arrays:

```
int matrix[3][2];
matrix[0][1] = 42;
```

- Arrays can be of primitive data type int, float, double, char
- Array can be of user defined data type using `typedef`

```
typedef char[20] char[20];
```

```
char[20] c20arr[10];
```

- Array can be of structure or union

```
struct user {
```

```
    int user_id;
```

```
    char[20] user_name;
```

```
};
```

```
struct user users[20];
```

What are the different ways to initialize the array values ?

```
int a[10]={0,1,4,6,7,12,8,9,-1,34};
```

```
char c[3]={'a','b','c'};
```

```
int b[3][2]={{2,5},{6,10},{-1,6}};
```

```
struct user u[2]={{1,"xyz"},{2,"abc"}}
```

OR

- Similarly using loops when don't know the input values

```
for(i=0; i<10; i++)  
    scanf("%d", &a[i]);
```

```
for(i=0; i<3; i++) {  
    for(j=0;j<2;j++)  
        scanf("%d", &b[i]);  
}
```

Strings

Difference between Character Array and String

- Character Array does not have a '\0' (NULL) to represent the end i.e. last character is regular ASCII character

I	n	t	r	o		2		p	r	o	g
---	---	---	---	---	--	---	--	---	---	---	---

- String ALWAYS has '\0' as a last character to represent end of string.

I	n	t	r	o		2		p	r	o	g	\0
---	---	---	---	---	--	---	--	---	---	---	---	----

Difference between Character Array and String - How to read user input ?

All 10 locations will have ASCII Character value

```
char c_arr[10];
for (i=0; i<10; i++)
    scanf("%c",&c_arr[i])
; OR
    c_arr[i] = getchar();
```

Only first 9 locations can be used for storing actual string characters, last one will have '\0'

```
char str[10];
scanf("%s", str); OR
gets(str);
```

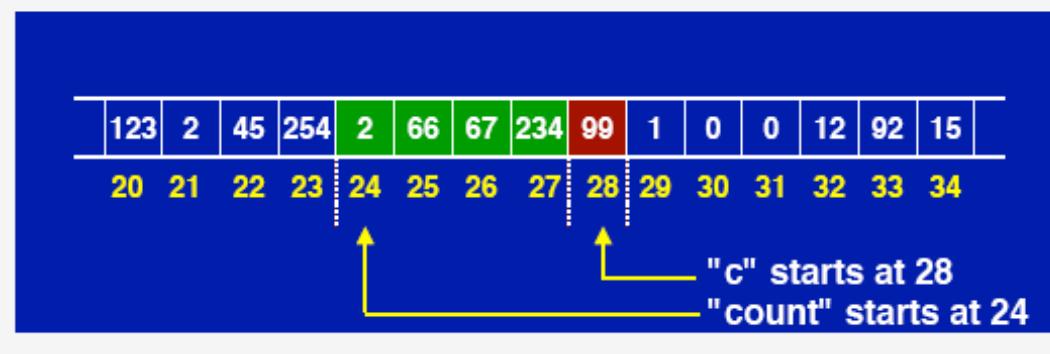
String Functions – Does not work for Character Array

- `strlen(s)` – returns the length of the string **excluding '\0'**
- `strcmp(s1, s2)` – returns 1 (+ve No) if $s1 > s2$, -1 (-ve No) if $s1 < s2$ and 0 if $s1$ equals $s2$
- `strcpy(dst, src)` – copies src string into dst string
- `strcat(s1, s2)` – returns concatenation of two string (i.e. $s1+s2$)
- `strstr(lrgstr, smlstr)` – index of 1st occurrence of smlstr into lrgstr

Memory Allocation for Variables

Declare variable int count and unsigned char c (what is difference between signed and unsigned value?)

- Memory is reserved to store the variables
- And the compiler 'remembers their location'



- As a result, each variable has two properties:

- The 'value' stored in the variable
 - If you use the name of the variable, you refer to the variable's value
- The 'address' of the memory used to store this value
 - Similar to a reference in Java (but not exactly the same)
 - A variable that stores the address of another variable is called a pointer

- Pointers can be declared using the * character

```
int *ptr;          /* Pointer to an int */  
unsigned char *ch; /* Pointer to an unsigned char */  
struct ComplexNumber *c; /* Pointer to a struct ComplexNumber */  
int **pp;          /* Pointer to a pointer to an int */  
void *v;           /* Pointer to anything (use with care!) */
```

Defining Pointers for primitive types

```
int i = 8, j=8;  
int *p, *p1, *p2;  
p = &i; p1=&i; p2=&j;  
double *d = &i // will this work?  
double d1 = i // will this work?  
double d2=1000.0008;  
i = d2; //will this work?  
i = (int) d2; will this work? What (int) in this  
case?  
  
sizeof(p) == sizeof(d)  
sizeof(*p) == sizeof(*d)
```

p1 == p2
*p1 == *p2
p == p1
What will be the result in each case?

```
int k = 8; int *p3;  
p3 = &k; *p3 = 12;  
Value of k?
```

Note: * Has 2 uses: 1. for pointer declaration 2. for getting the data value pointed by pointer variable
[C_revision\pointers_knowledge_check.c](#)

Arrays and Pointers

```
int a[10], *ptr;
```

```
ptr= &a[0]; // ptr will point to first  
array element
```

```
printf("%d", *(ptr+5) );// ptr+ 5 = ?
```

If address of a[0] is 2000 then
what will be the value of ptr+ 5?

ptr[7] = 10; // will this work?

ptr=a; // will this work?

```
printf("%d", *ptr++);
```

```
printf("%d", *(ptr++));
```

What is the difference?

Be aware of precedence of
operators

Structures and Pointer to Structures

- You can build higher-level data types by creating structures:

```
struct Complex {  
    float real;  
    float imag;  
};  
struct Complex number;  
number.real = 3.2;  
number.imag = -2;  
  
struct Parameter {  
    struct Complex number;  
    char description[32];  
};  
struct Parameter p;  
p.number.real = 42;  
p.number.imag = 12.3;  
strncpy(p.description, "My nice number", 31);
```

Structures and Pointer to Structures

- We very often use statements like:

```
(*pointer).field = value;
```

- There is another notation which means exactly the same:

```
pointer->field = value;
```

- For example:

```
struct data {  
    int counter;  
    double value;  
};  
  
void add(struct data *d, double value) {  
    d->counter++;  
    d->value += value;  
}
```

Functions: Library and User Defined

Library

- From `stdio.h`: `printf`, `scanf`
- From `string.h`: `strcpy`, `strcmp`, `strstr`,
`strlen`, `strcat`
- From `unistd.h`: `read`, `write`, `Iseek`
- From `stdlib.h`: `open`, `close`

User Defined

- Definition, Declaration and Call
- Scope of the variables
- Pass parameters by value or by reference
 ?)[C revision\pass by value or ref.c](#)
- Defined in same file vs different file
- Use of Header file: extern functions
[C revision\file1 main.c](#),
[C revision\file2 other.c](#),
[C revision\myheader.h](#)
- Recursion

C pointer to array function

- Declare Array of Function pointers and initialize

```
int add(int a, int b);
```

```
int sub(int a, int b);
```

```
int mul(int a, int b);
```

```
int div(int a, int b);
```

```
int (*oper[4])(int a, int b) = {add, sub, mul, div};
```

- Call the function using initialized function pointer array

```
int result = oper[i](a,b);
```

[C revision\function_ptr_example.c](#)

Recursive Functions – Calls itself until terminating condition

```
int fibo(int num) {  
    if (num == 0)    {  
        return 0;  
    }  
    else if (num == 1)  {  
        return 1;  
    }  
    else  {  
        return(fibo(num - 1) + fibo(num - 2));  
    }  
}
```

Dynamic Memory Management

```
#include <stdlib.h>
int a[10];
int *ptr_i = (int *) malloc(10 * sizeof(int));
```

Are these equivalent?

You must call `free()` for any pointer which has been allocated a memory. What problem will it cause if you don't?

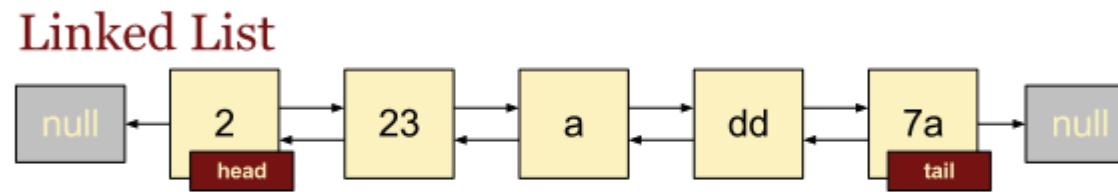
Similarly you can use pointer to structure to access the members of the structure

```
struct person { int p_id; char p_name[20];} *p
p = (struct person *) malloc(sizeof(person)); // will store only 1 person info
scanf("%d",&p->p_id); scanf("%s", p->p_name);
```

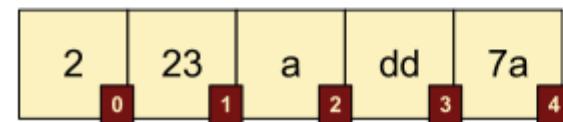
Data structures using C pointers

- Array vs Linked List

Array vs. Linked List

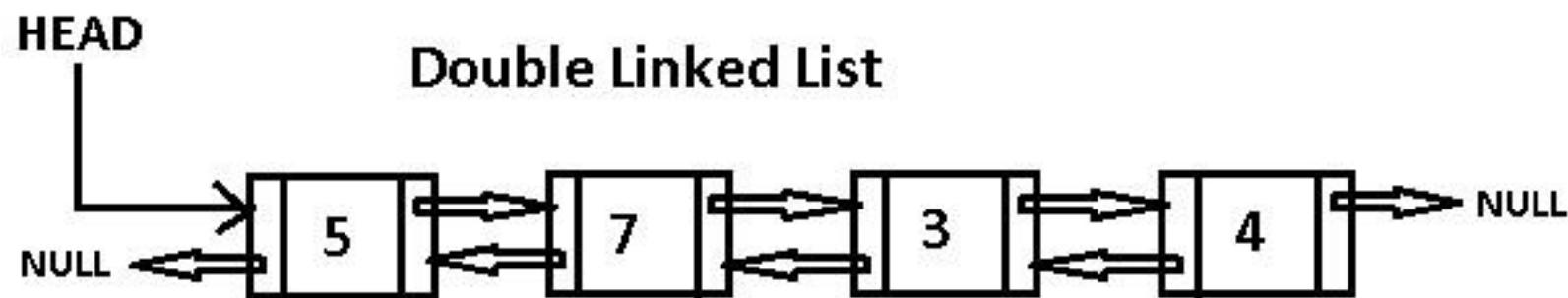
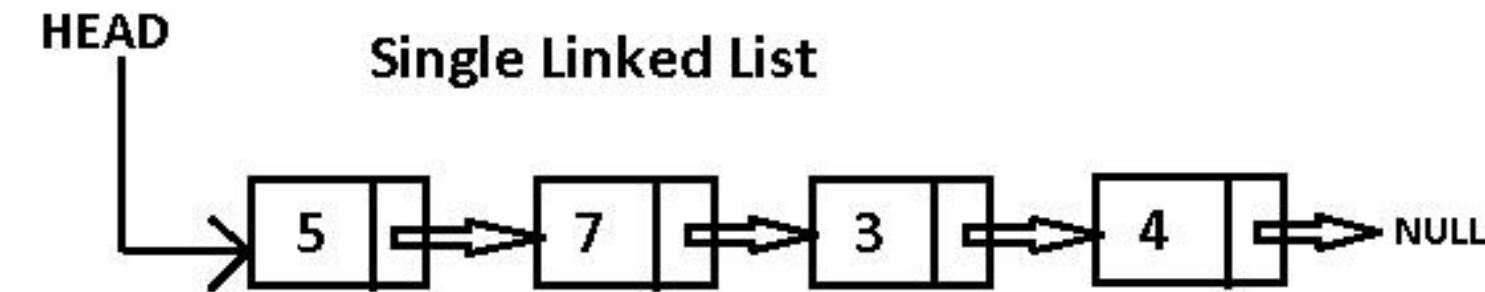


Array



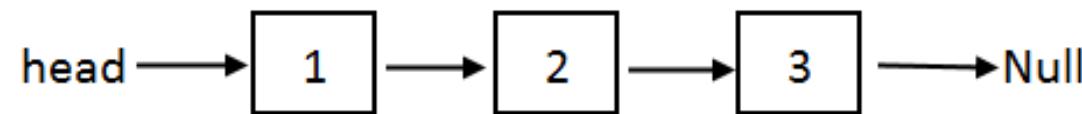
Data structures using C pointers

- Singly vs Doubly Linked List

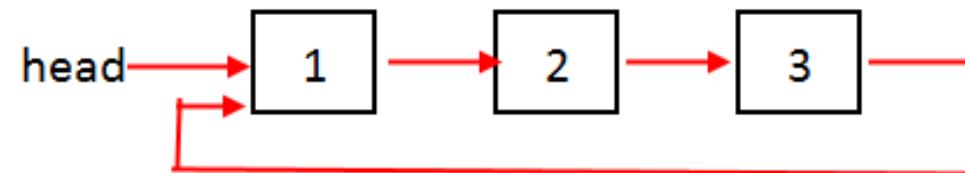


Data structures using C pointers

- Singly Regular vs Singly Circular Linked List



Singly Linked List

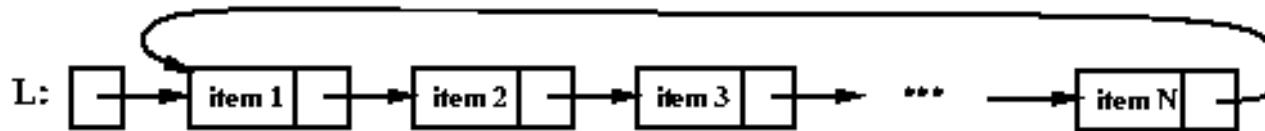


Circular Linked List

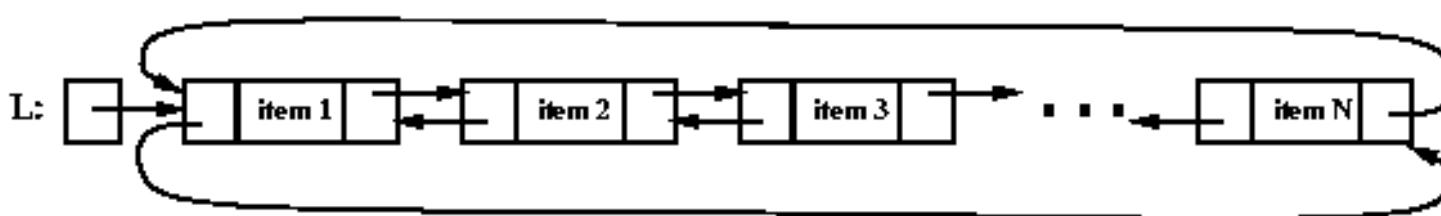
Data structures using C pointers

- Singly Circular vs Doubly Circular Linked List

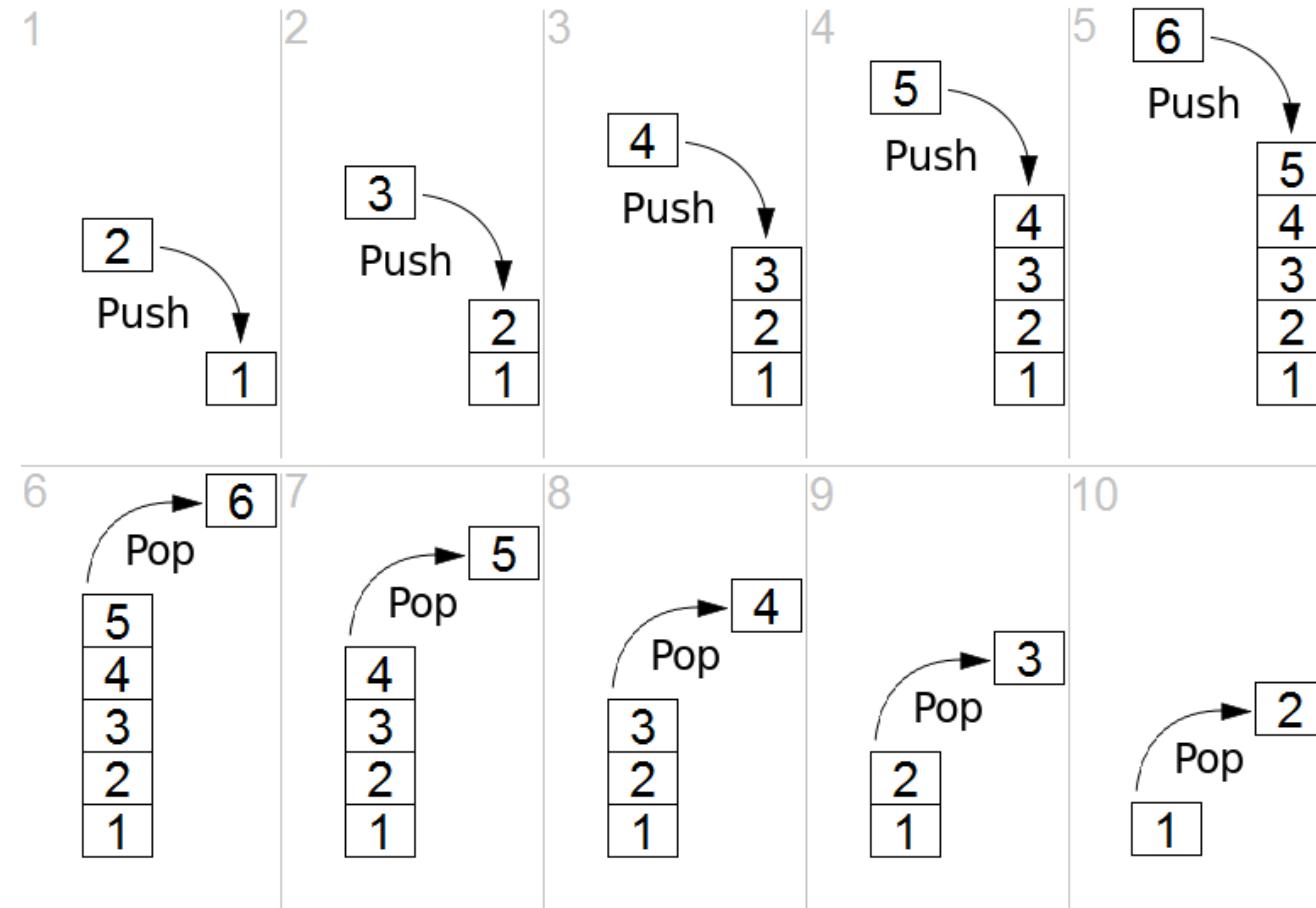
Circular, singly linked list:



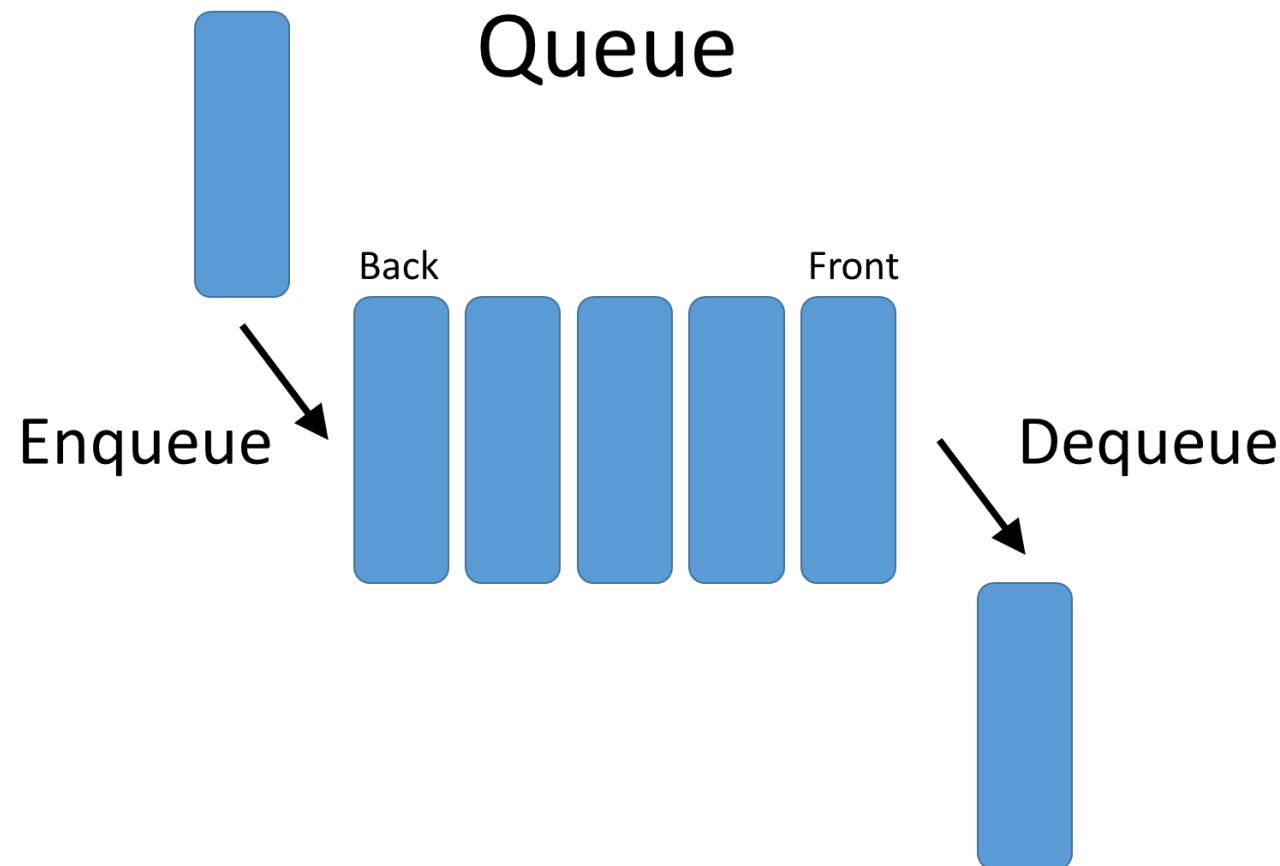
Circular, doubly linked list:



Other Data structure implementation - Stack



Other Data structure implementation - Queue



Other Data structure implementation - Trees

- Binary Trees: Binary Tree, Binary Search Tree, AVL Tree, Red-Black Tree, Splay Tree
- B-Tree: B-Tree, B+ Tree, B sharp tree
- General Trees: Trie, Radix Tree, Suffix Tree, B Trie
- Multiway Tree: Ternary Tree, K-ary Tree, And-or Tree, (a,b)-tree
- Space Partitioning Tree: K-d Tree, R-tree, Segment tree, Range tree, Octree, Herbert R-tree
- Application Specific Tree: Parse Tree, Decision Tree, MinMax Tree, Expression Tree

Example for Circular Double Linked List

```
struct user {  
    int user_id;  
    char[20] user_name;  
    struct user *prev;  
    struct user *next;  
} head, tail;
```

Example for Circular Doubly Linked List

Adding a new node in the list:

```
struct user *new = (struct user *)malloc(sizeof(struct));
new->user_id=<new id>; new->user_name=<new name>;
new.next = NULL; new.prev = NULL;
if(head == NULL)
    head = new; tail = new; head->prev = head; head->next = head
else {
    tail->next = new;
    new->prev = tail;
    new->next = head;
    head->prev = new;
    tail = new;
}
```

Example for Circular Doubly Linked List

Delete a new node from the list for a given user_id

```
if(head==NULL) {  
    struct user temp;  
    for(temp=head; temp!=NULL; temp->next)  
        if(temp->user_id == <given user_id>)  
            break;  
    if(temp != NULL) {  
        temp->prev->next = temp->next;  
        temp->next->prev = temp->prev;  
        temp->prev=NULL;  
        temp->next=NULL;  
        free(temp);  
    }  
}
```

Headers defined by the ISO (International Standard Organization) C standard

- `<assert.h>` : Conditionally compiled macro that compares its argument to zero
- `<complex.h>` (since C99) : Complex number arithmetic
- `<ctype.h>` : Functions to determine the type contained in character data
- `<errno.h>` : Macros reporting error conditions
- `<fenv.h>` (since C99) : Floating-point environment
- `<float.h>` : Limits of float types
- `<inttypes.h>` (since C99) : Format conversion of integer types
- `<iso646.h>` (since C95) _ Alternative operator spellings
- `<limits.h>` : Sizes of basic types
- `<locale.h>` : Localization utilities
- `<math.h>` : Common mathematics functions
- `<setjmp.h>` : Nonlocal jumps
- `<signal.h>` : Signal handling
- `<stdalign.h>` (since C11) : alignas and alignof convenience macros
- `<stdarg.h>` : Variable arguments

Headers defined by the ISO (International Standard Organization) C standard

- `<stdatomic.h>` (since C11) : Atomic types
- `<stdbool.h>` (since C99) : Boolean type
- `<stddef.h>` : Common macro definitions
- `<stdint.h>` (since C99) : Fixed-width integer types
- `<stdio.h>` : Input/output
- `<stdlib.h>` : General utilities: memory management, program utilities, string conversions, random numbers
- `<stdnoreturn.h>` (since C11) : noreturn convenience macros
- `<string.h>` : String handling
- `<tgmath.h>` (since C99) : Type-generic math (macros wrapping math.h and complex.h)
- `<threads.h>` (since C11) : Thread library
- `<time.h>` : Time/date utilities
- `<uchar.h>` (since C11) : UTF-16 and UTF-32 character utilities
- `<wchar.h>` (since C95) : Extended multibyte and wide character utilities
- `<wctype.h>` (since C95) : Functions to determine the type contained in wide character data

Required Headers for IEEE POSIX (Portable Operating System Interface) Standard (This is in addition to ISO C Standard)

- <direct.h> : directory entries
- <**fcntl.h**> : file control
- <fnmatch.h> : filename-matching types
- <glob.h> : pathname-pattern matching types
- <grp.h> : group file
- <netdb.h> : network database operation
- <pwd.h> : password file
- <regex.h> : regular expressions
- <tar.h> : tar archive values
- <termios.h> : terminal I/O
- <**unistd.h**> : symbolic constants
- <utime.h> : file times
- <wordexp.h> : word-expansion types

Required Headers for IEEE POSIX (Portable Operating System Interface) Standard (This is in addition to ISO C Standard)

- `<arpa/inet.h>` : Internet definition
- `<net/if.h>` : socket local interface
- `<netinet/in.h>` : internet address family
- `<netinet/tcp.h>` : Transmission Control Protocol definition
- `<sys/nman.h>` : memory management declaration
- `<sys/select.h>` : select function
- `<sys/socket.h>` : socket interface
- `<sys/stat.h>` : file status
- `<sys/times.h>` : process times
- `<sys/types.h>` primitive system datatypes
- `<sys/un.h>` : UNIX domain socket definition
- `<sys/utsname.h>` : system name
- `<sys/wait.h>` process control

Exercise Setup

- You are given a task to keep track of number of queries running in database management system. Below information regarding queries need to be kept in C structure.

```
struct query {  
    int query_id;      // unique id  
    char* query_text; // actual query text  
    int status;        // submitted, running, finished  
    int time_elapsed; // time in microsec it took to run the query  
}
```

Exercise Problem 1

- Create an array of structure for struct query defined previously using dynamic memory allocation and to provide the following functions.
For each of functionality you can use user menu selection
 - New query is submitted by a SQL developer i.e. new array entry is added with status as submitted
 - Query has started running so update to running
 - Query is running so periodically need to update time_elapsed
 - Query has finished running so update the status as finished and remove the entry from array

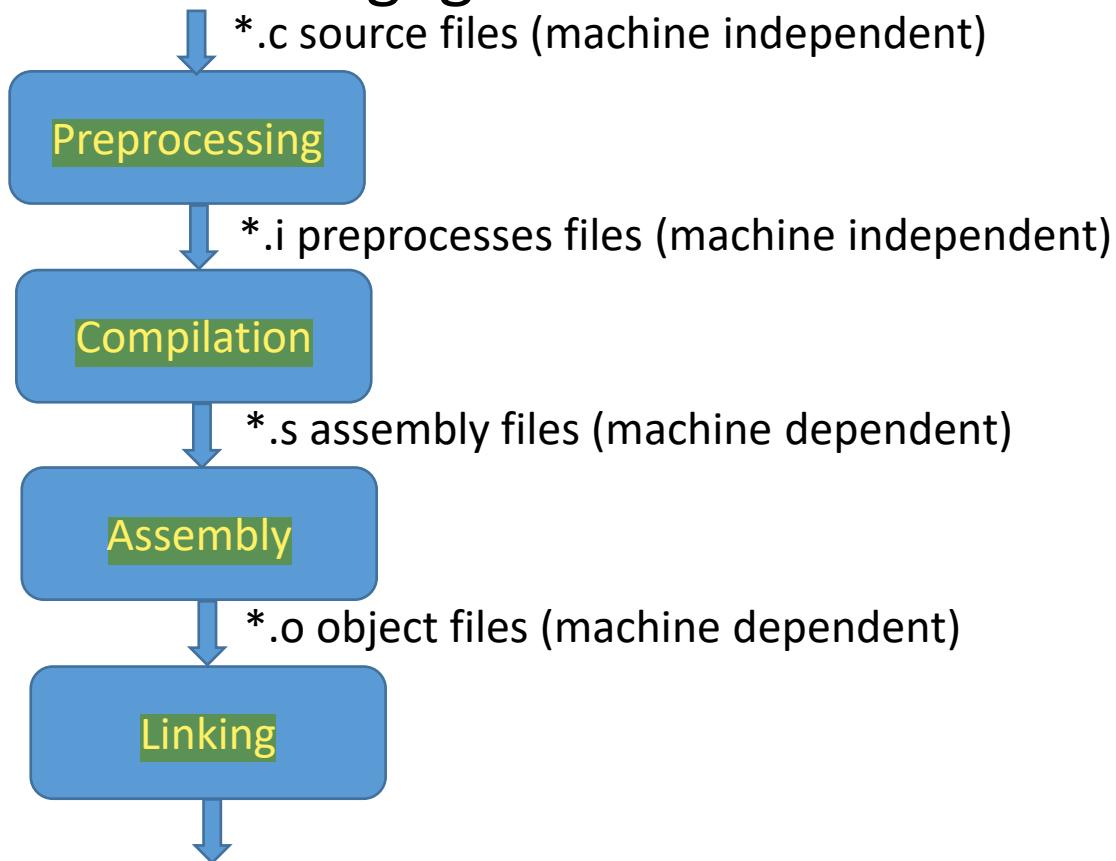
Exercise Problem 2

- Functionality of Problem 2 is same as Problem1
 - New query is submitted by a SQL developer i.e. new array entry is added with status as submitted
 - Query has started running so update to running
 - Query is running so periodically need to update time_elapsed
 - Query has finished running so update the status as finished and remove the entry from array
- But Problem 2 should be implemented using linked list instead of array
- Complete the code in C revision\excercise2.c

Systems
Software/Programming
Preprocessing, Compilation,
Assembly, Linking, Loading

GCC GNU compiler for Unix (and Unix like OS)

- Documentation available at: <https://gcc.gnu.org/onlinedocs/>
- Steps followed by gcc when executing: gcc file.c
 - Preprocessing
 - Compilation
 - Assembly
 - Linking



Preprocessing

- `cpp` is preprocessor for `gcc` which is invoked automatically by `gcc`

```
cpp helloworld.c > helloworld.i
```

The same can be done with `-E` switch in `gcc`

```
gcc -E helloworld.c > helloworld.i
```

- Handles preprocessor directives such as `#define`, `#include`, `#ifdef`, `#endif` etc
- E.g. `#include <stdio.h>` tells preprocessor to read the content of `stdio.h` and insert into program text. Generally creates new file with `.i` extension which is not saved unless `-f-savetemps` option is used with `gcc`
- [C_revision\helloworld.c](#), [C_revision\helloworld.h](#), [C_revision\helloworld.i](#)

Compilation

- Compiler cc translates .i file which contains preprocessed program text to .s file an assembly code

```
cc -S helloworld.i
```

OR

```
gcc -S helloworld.i
```

- -S option with gcc is used to create assembly code
- **What is assembly code?**

[C_revision\helloworld.s](#)

Assembly

- Assembler (as) translates .s file containing assembly code into object file (.o) that has code in machine language

```
as helloworld.s -o helloworld.o
```

Note: Generally we don't follow the process of preprocessing, compilation and assembly separately. Instead we can use following command directly

```
gcc -c helloworld.c -o helloworld.o
```

-c option will perform all 3 steps (preprocessing, compilation and assembly) and create object file directly

Object Files are binary files so cannot be viewed without binary editor

For more readable assembly code you can use

```
gcc -g -c helloworld.c → -g option puts debugging information in object file
```

```
objdump -d -M intel -S helloworld.o > helloworld.asm
```

[C_revision\helloworld.asm](#)

Linking: ld linker program used within gcc

- C program uses many function declared as user defined (fib, fact etc) or system defined library functions (scanf, printf etc)
- Linking is the process of linking multiple object files to generate a single executable

e.g. progmain.c contains main() function and uses user defined function fib() from fib.c

After first 3 steps 2 object files are created: progmain.o, fib.o

Linking will take the reference of fib() function from fib.o object file and update in progmain.o so that the fib function can be called

```
gcc helloworld.o -o helloworld
```

Will generate helloworld as executable (default name of executable in unix/linux is a.out when -o option is not used)

[C_revision\helloworld.out.assembly](#)

Use of multiple C files

- file1_main.c → only compile using –c flag
 - Compile with “gcc –c file1_main.c” will create file1_main.o object file
- file2_other.c → only compile using –c flag
 - Compile with “gcc –c file2_other.c” will create file2_other.o object file
- myheader.h – user defined header file with function declaration and external variables
- Finally run linker to build the executable
 - “gcc file1_main.o file2_other.o –o file1_main.out”

Linking Static or Shared(Dynamic) Library

- Shared library (unix extension .so, windows extension .dll)
- Static library (unix extension .a, windows extension .lib)

Static Library	Shared Library
Referenced function code is part of program code so loaded with program code	Referenced function code is loaded only on needed basis (when function is called)
Object code of referenced function is embedded in program object code from static library	Object code of referenced function is NOT embedded in program object code from shared library
Object code of referenced function is linked at compile time (static linking)	Object code of referenced function is linked at runtime (dynamic linking)
Increases the size of executable	Increase the runtime of running program

Create Static Library

- Create Library File using User Defined Functions

Create 2 source files add.c and sub.c which has functions “int add(int, int)” and “int sub(int, int)” for performing addition and subtraction.

Step1: **Compile sources into object files (use -c option of gcc):**

gcc -c add.c and gcc -c sub.c (or gcc -c add.c sub.c)

This will create add.o and sub.o object files

Step2: **Create static library using object files:**

ar rs libmymath.a add.o sub.o

Create Static Library Continue...

- Use Static Library functions add() and sub() in Application Program

Step3: Include functions add() and sub() by declaring them in a user-defined header file or declaring them in application program source.

Step4: Create application program source file to call library function and compile

```
gcc -c call_lib_function.c
```

Step5: Now create an executable using library function

```
gcc -o call_lib_func.out call_lib_func.o libmymath.a OR
```

```
gcc -o call_lib_func.out -L . call_lib_func.o -lmymath
```

Create Shared(Dynamic) Library

Consider you have 2 source files add.c and sub.c which has functions for performing addition and subtraction.

Step1: Compile using –fPIC option for “Position Independent Code”

```
gcc -fPIC -c add.c sub.c
```

Create new object file with “Position Independent Code”

Step2: Create shared library:

```
gcc -shared -o libmymath.so add.o sub.o
```

This will create shared library

Step3: Create a proprietary header file e.g. myheader.h to declare functions add() and sub()

Step4: Create c file to call library function and compile

```
gcc -c call_lib_function.c
```

Create Shared Library Cont...

Step5: Now you can create an executable using shared library

```
gcc -o call_lib_func call_lib_func.o libmymath.so OR
```

```
gcc -o call_lib_func -L . Call_lib_func.o -lmymath
```

Compare Disassembly

C_revision\call_lib_func_static.assembly

```
689:    e8 80 00 00 00      call  70e <add>
000000000000070e <add>:
70e:    55                  push   rbp
70f:    48 89 e5            mov    rbp,rs
712:    f2 0f 11 45 f8      movsd  QWORD PTR [rbp-
0x8],xmm0
717:    f2 0f 11 4d f0      movsd  QWORD PTR [rbp-
0x10],xmm1
71c:    f2 0f 10 45 f8      movsd  xmm0,QWORD PTR
[rbp-0x8]
721:    f2 0f 58 45 f0      addsd  xmm0,QWORD PTR
[rbp-0x10]
726:    5d                  pop    rbp
727:    c3                  ret
```

C_revision\call_lib_func_dynamic.assembly

```
7d9:    e8 72 fe ff ff      call  650 <add@plt>
0000000000000650 <add@plt>:
650:    ff 25 6a 09 20 00    jmp   QWORD PTR
[rip+0x20096a]    # 200fc0 <add>
656:    68 00 00 00 00      push   0x0
65b:    e9 e0 ff ff ff      jmp   640 <.plt>
0000000000000640 <.plt>:
640:    ff 35 6a 09 20 00    push   QWORD PTR
[rip+0x20096a]    # 200fb0
<_GLOBAL_OFFSET_TABLE_+0x8> 646: ff 25 6c 09 20
00      jmp   QWORD PTR [rip+0x20096c]    #
200fb8 <_GLOBAL_OFFSET_TABLE_+0x10> 64c: 0f 1f
40 00      nop   DWORD PTR [rax+0x0]
```

Executing code using static library

- Just run the executable
C revision\call lib func static.out since all the library function are part of binary executable code

Executing code using dynamic library

- ldd utility shows which library is not reachable
 - ldd call_lib_func_dynamic.out
- linux-vdso.so.1 (0x00007ffe308d7000)
- libmymath.so => not found
- libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
(0x00007ff535154000)
/lib64/ld-linux-x86-64.so.2 (0x00007ff535747000)
- Perform Option1, 2 or 3 shown next...

Executing code using dynamic library Cont...

Set search path for Shared Library file you have created (**why do we need this?**)

Option1: Set `LD_LIBRARY_PATH` environment variable (You can modify `.bashrc` file if this path needs to be added at the start of the terminal)

```
export LD_LIBRARY_PATH=<path to library  
file>:$LD_LIBRARY_PATH
```

This will look for .so file in the `<path to library file>`

Executing code using dynamic library Cont...

Option2 (not good option):

1. Copy the .so file to folder which is the default locations where OS search for library file
2. run Idconfig to add the shared directory in Id (dynamic linker). This will register the .so file with Id to look in /usr/lib folder

Note: default paths for shared library in *.conf files under /etc/ld.so.conf.d

Executing code using dynamic library Cont...

Option3:

1. Create your own conf file in
`/etc/ld.so.conf.d/myapplib.conf` with path to *.so
library.
2. Run `ldconfig` with sudo access to register the .so file
with ld to look in your own library folder

Loading

- When executable is used as command on shell prompt, it first need to be loaded from disk to memory
- Executable code is loaded (except shared library functions used in program code) plus data memory (heap) and stack is allocated
- As the function is called from program to shared library they are loaded from library files containing object code of the referenced function

gcc optimization levels

- -O0 → No optimization (the default); generates unoptimized code but has the fastest compilation time
- -O1 → Moderate optimization; optimizes reasonably well but does not degrade compilation time significantly
- -O2 → Full optimization; generates highly optimized code and has the slowest compilation time
- -O3 → Full optimization as in -O2; also uses more aggressive automatic inlining of subprograms within a unit (Inlining of Subprograms) and attempts to vectorize loops
- -Os → Optimize space usage (code and data) of resulting program.

GNU Make

(<https://www.gnu.org/software/make/>)

- Assuming we have following 3 files:
- hellomake.c

```
#include <hellomake.h>

int main() {
    // call function in another file
    myPrintHelloMake();
    return(0);
}
```

- hellofunc.c

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

void myPrintHelloMake(void) {      printf("Hello makefiles!\n");
    return;
}
```

- hellomake.h

```
/* example include file */
void myPrintHelloMake(void);
```

How do compile and link using gcc on command line
gcc -o hellomake hellomake.c hellofunc.c -I.

-I. Indicates look for header file in current directory not only the default location of header files /usr/include

What are the problem with this approach?

1. If you type this command on one computer and also need to run on another computer you have to retype
2. Even if one c file is modified all (both) the files will be recompiled waisting precious CPU time

Make

- 1st problem is solved by creating Makefile or makefile with following command and then just type make on command prompt which will run this makefile : [C revision\makefile1](#)

hellomake: hellomake.c hellofunc.c

 gcc -o hellomake hellomake.c hellofunc.c -l.

Shows that hellomake.c and hellofunc.c are the dependent files to generate executable hellomake

Using makefile you can run this command on any computer but problem 2 is not yet solved i.e. if any of the source file is changed both will be recompiled

Make

To solve problem 2 we modify makefile as below : [C_revision\makefile2](#)

CC=gcc

CFLAGS=-I.

hellomake: hellomake.o hellofunc.o

\$(CC) -o hellomake hellomake.o hellofunc.o -I.

Make uses some standard Make variables; CC indicates which C compiler to use and CFLAGS indicates the compiler flags

By specifying object files as dependency of executable rather than source files it informs make that source files need to be compiled individually which means if the source file has not been modified don't recompile that source file

What is the problem now? If the only the hellomake.h header file is modified then make will not compile any of the source files

Make

makefile3

CC=gcc

CFLAGS=-I.

DEPS = helломake.h

OBJ = helломake.o hellofunc.o

Just to recompile only those C file whose header file is modified

% .o: %.c \$(DEPS)

\$(CC) -c -o \$@ \$< \$(CFLAGS)

→ only compile with source and dependent header file

helломake: \$(OBJ)

\$(CC) -o \$@ \$^ \$(CFLAGS) → only generate executable using object files

Now we add DEPS as another variable and mention all the header files which on which source file depends

New rule has to be added for object file creation.

-o \$@ says to put the output of the compilation in the file named on the left side of the :

\$< is the first item in the dependencies list

\$^ indicates take the values from **left** side of the previous command i.e. all the object files in this case

right

Make

- what if we want to start putting our .h files in an include directory, our source code in a src directory, and some local libraries in a lib directory? Also, can we somehow hide those annoying .o files that hang around all over the place?

Make – Format generally used for professional software

Folder Structure for Project : [make_proj](#)

```
$ tree
```

```
.  
├── include  
│   └── hellomake.h  
└── src  
    ├── hellofunc.c  
    ├── hellomake  
    ├── hellomake.c  
    ├── makefile → need to run this makefile  
    └── obj  
        ├── hellofunc.o  
        └── hellomake.o
```

Make – Format generally used for professional software

IDIR =../include → you can put include files in this directory

CC=gcc

CFLAGS=-I\$(IDIR)

ODIR=obj → directory where object files will be created

LDIR =../lib → you can put library files in this directory

LIBS=-lm → looks for library libm.so or libm.a

_DEPS = hellomake.h

DEPS = \$(patsubst %,\$(IDIR)/%,\$(_DEPS)) → create string with ../include/<header filename> in our case ../include/hellomake.h

_OBJ = hellomake.o hellofunc.o

OBJ = \$(patsubst %,\$(ODIR)/%,\$(_OBJ)) → create string with obj/<object filename> in our case obj/hellomake.o and obj/hellofunc.o (obj folder will be created under current i.e. source directory)

Make – Format generally used for professional software

```
$(ODIR)/%.o: %.c $(DEPS)
```

```
    $(CC) -c -o $@ $< $(CFLAGS)
```

```
hellomake: $(OBJ)
```

```
    gcc -o $@ $^ $(CFLAGS) $(LIBS)
```

.PHONY: clean

create dummy label to run make file

```
clean:
```

```
    rm -f $(ODIR)/*.o *~ core $(INCDIR)/*~ → removes object files
```

and temporary files created with ~ at the end of file name in current
i.e. source and include directory

Quiz

- Consider an example of creating a dynamic library from previous slides.
- Assume directory structure as

project

src

 obj → all object file should be created here

 lib → dynamic library should be created here

 include

 bin → application executable should be created here

- Develop “makefile” to:

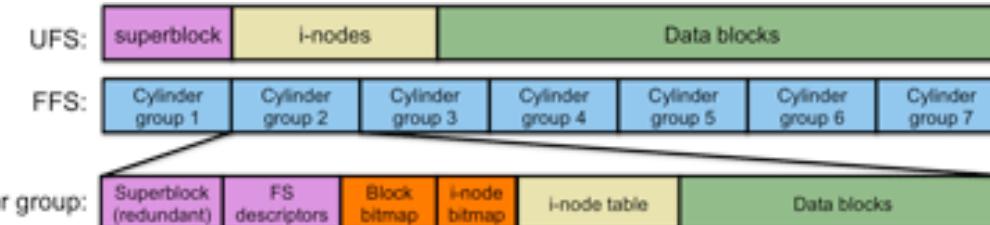
- Build dynamic library in lib folder using add.c and sub.c functions
- Build application binary using library in bin folder

Systems
Software/Programming
File IO

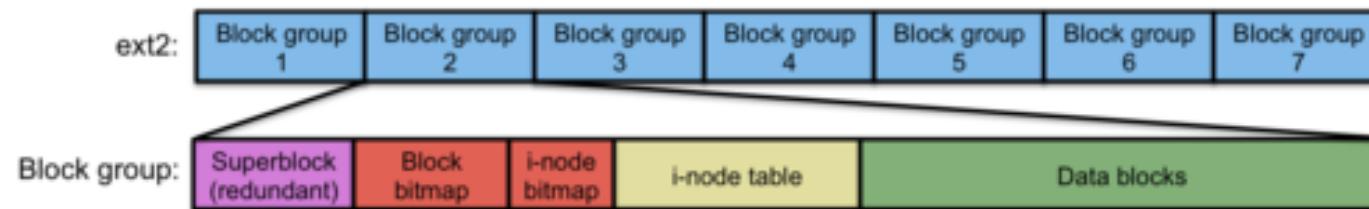
Files and Directories in Unix based systems

Todays File Systems Extended from UFS (Unix File System)

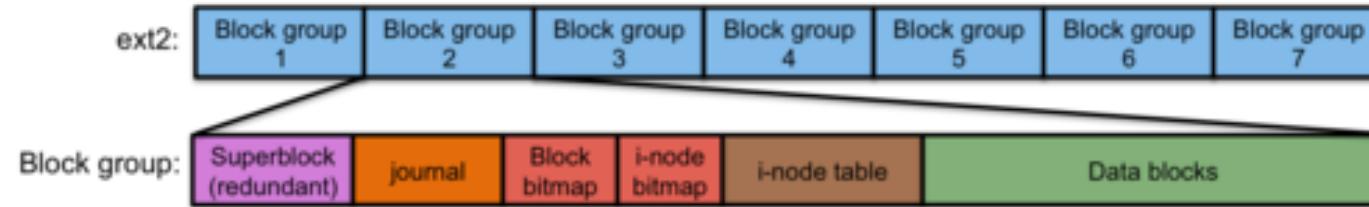
- BSD Fast File System



- Linux ext2

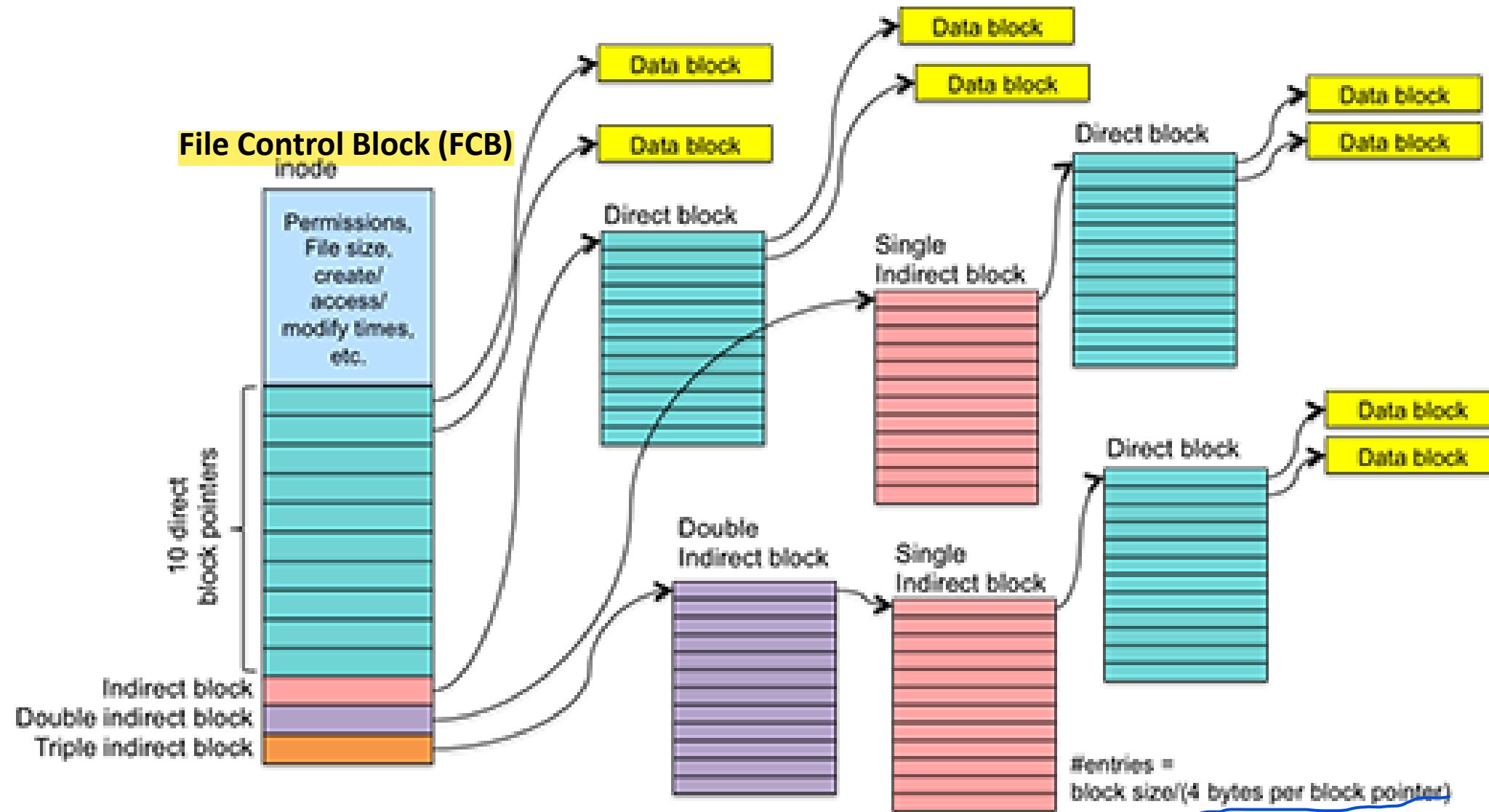


- Linux ext3



- Linux ext4 – Structure is same as ext3 with some addition (only 2 are listed here)
 - Large file system (disks upto 1 exabytes vs 32 TB in ext3 and file size upto 16TB vs 2TB in ext3)
 - Larger Directories 64000 entries vs 32000 in ext3

Unix File System (UFS) inode layout in disks



UFS vs EXT Superblock

- UFS superblock (Volume Control Block) contains
 - Size of file system
 - Number of free blocks
 - list of free blocks (including a pointer to free block lists)
 - index of the next free block in the free block list
 - Size of the inode list
 - Number of free inodes in the file system
 - Index of the next free inode in the free inode list
 - Modified flag (clean/dirty)
- FFS/EXT superblock (Volume Control Block) contains
 - Blocks in the file system
 - No of free blocks in the file system
 - Inodes per block group
 - Blocks per block group
 - No of times the file system was mounted since last fsck.
 - Mount time
 - UUID of the file system
 - Write time
 - File System State (i.e.: was it cleanly unmounted, errors detected etc.)
 - The file system type etc.(i.e.: whether its ext2,3 or 4).
 - The operating system in which the file system was formatted

inode File Control Block (FCB)

- UFS inode contains (aka File Control Block FCB)
 - file owner, group owner
 - file type (regular file, directory, character device, block device, FIFO)
 - access permissions
 - times: create, last access, and last modification
 - number of links to the file, which is the number of file names in the directory that refer to this inode
 - file size or major/minor numbers if the file is a device
 - file data location: ten direct blocks, one indirect block, one double indirect block, and one triple indirect block

See the inode information

For Current Directory

```
faculty@faculty-OptiPlex-3040:~$ ls -lia
```

- total 134204
- 9699330 drwxr-xr-x 57 faculty faculty 4096 Dec 21 10:59 .
- 9699329 drwxr-xr-x 3 root root 4096 Feb 24 2017 ..
- 9699466 -rw-rw-r-- 1 faculty faculty 94319943 Mar 21 2017 2.4.13
- 12323363 drwxrwxr-x 3 faculty faculty 4096 Aug 29 13:29 ApacheSpark
- 10094452 drwxrwxr-x 3 faculty faculty 4096 Jun 28 12:03 BackupUSB
- 9699526 -rw----- 1 faculty faculty 49233 Dec 21 13:23 .bash_history
- 9699331 -rw-r--r-- 1 faculty faculty 220 Feb 24 2017 .bash_logout
- 9705098 -rw-r--r-- 1 faculty faculty 4330 Nov 24 14:17 .bashrc
- 9704737 -rw-r--r-- 1 faculty faculty 4198 Aug 23 13:30 .bashrc~

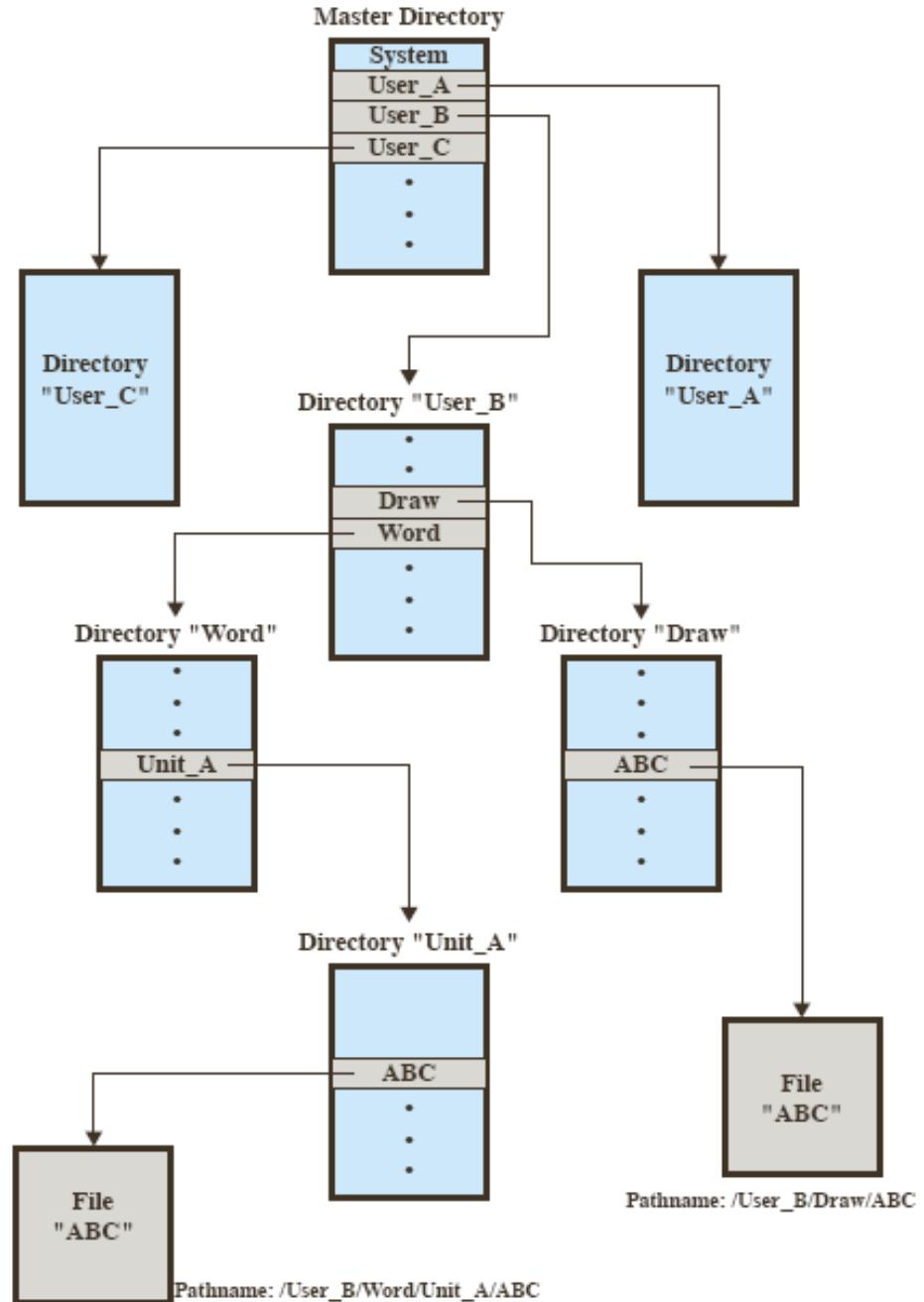
Recursively for current and all subdirectories

```
faculty@faculty-OptiPlex-3040:~$ ls -iaR virtualenv/ | more
```

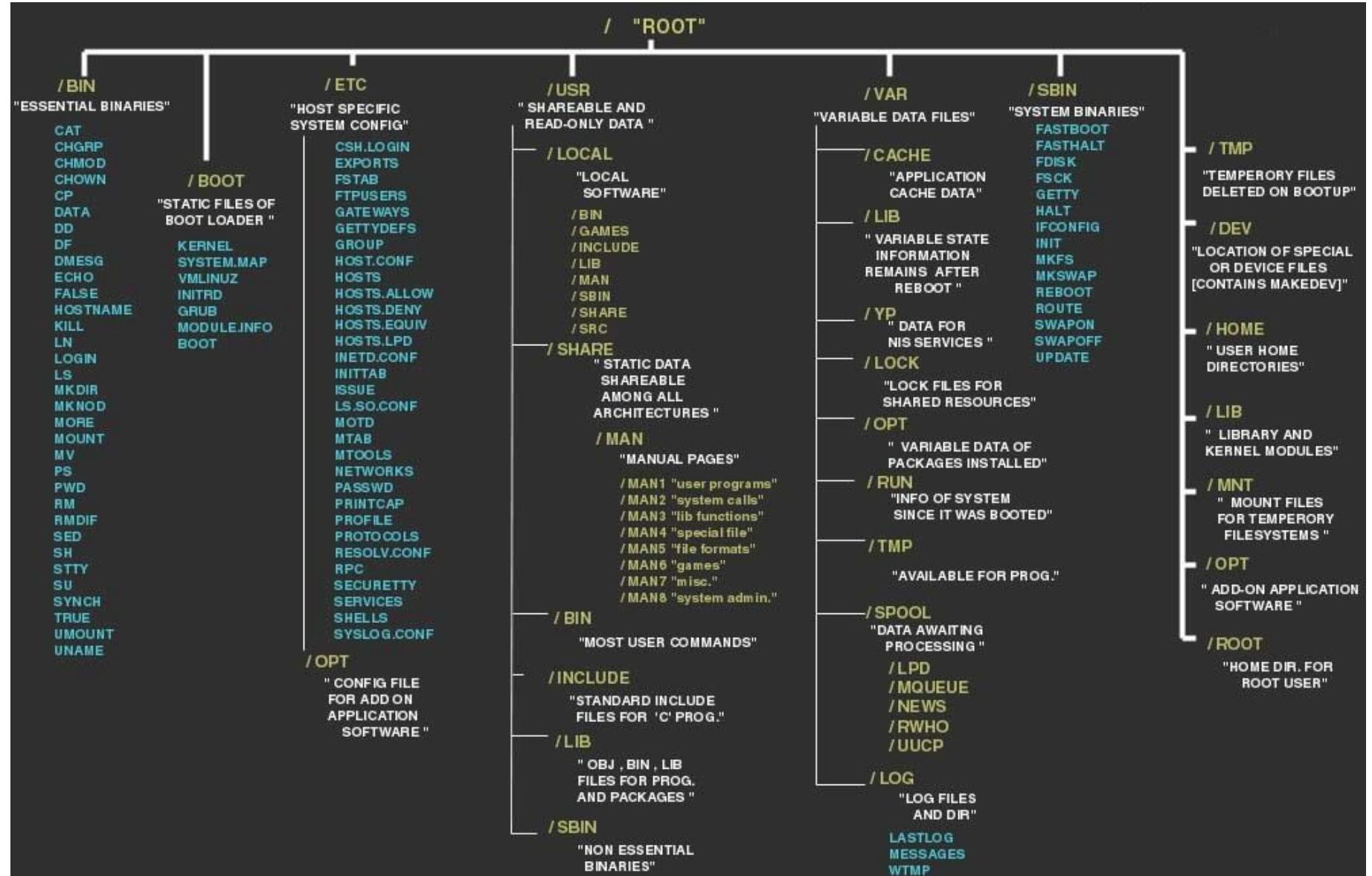
- virtualenv/:
- 9704509 .
- 9699330 ..
- 11540615 architectureanalysis
- 9704475 bashrcn
- 9704543 bashrcn~
- 9705036 hosts
- 9704703 hosts~
- 9699741 ml
- 9704714 mpi_pi.c
- 9704716 mpi_pi.out
- virtualenv/architectureanalysis:
- 11540615 .
- 9704509 ..
- 11407369 archanalyze_decisiontree.ipynb
- 11540737 archanalyze_linearregression.ipynb
- 11541142 archanalyze_mlp-Copy1.ipynb
- 11540670 archanalyze_mlp.ipynb
- 9312887 archanalyze_mlp.py
- 11540730 archanalyze_partialleastsquare.ipynb

Directories

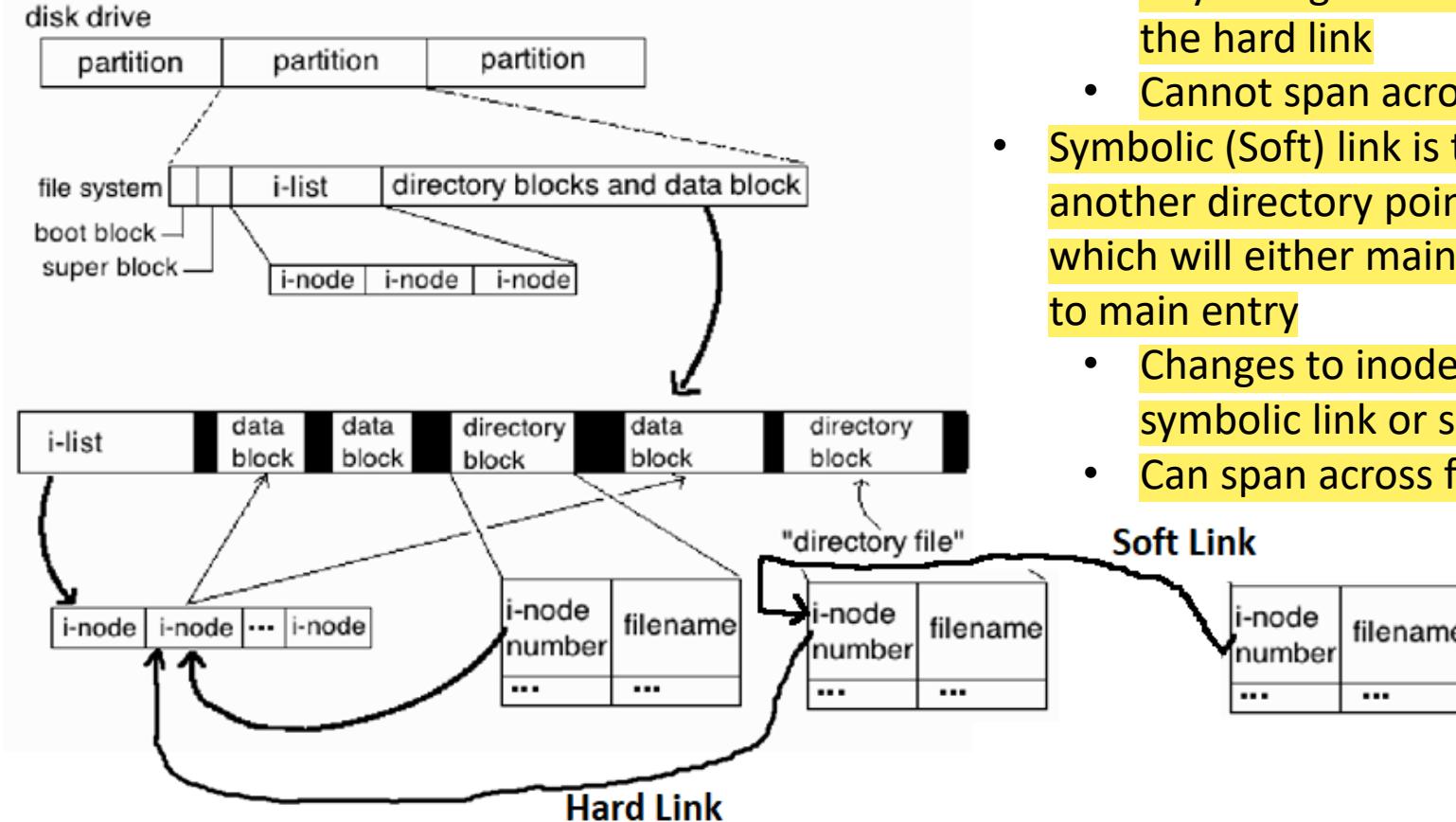
- Directories are special files that keep track of other files
 - the collection of files is systematically organized
 - first, disks are split into partitions that create logical volumes (can be thought of as “virtual disks”)
 - second, each partition contains information about the files within
 - this information is kept in entries in a **device directory** (or volume table of contents)
 - the directory is a symbol table that translates file names into their entries in the directory
 - it has a logical structure
 - it has an implementation structure (linked list, table, etc.)



Linux File System Hierarchy



Directory entries pointing to file with link - Two directory entries refer to the same file but with different names



- Hard link is the directory entry of a file or another directory pointing to inode directly
 - Any changes to inode of original file does not affect the hard link
 - Cannot span across file systems
- Symbolic (Soft) link is the directory entry of a file or another directory pointing to another directory entry which will either main entry or will have hard or soft link to main entry
 - Changes to inode in original file will affect the symbolic link or soft link
 - Can span across file system

Create Symbolic Links (ln command)

Hard Link

```
ln mount_HostFiles.sh mnt
```

```
$ ls -lia m*
```

```
138633 -rwxrwxrwx 2 ubuntu18  
ubuntu18 92 Jan 4 14:34 mnt
```

```
138633 -rwxrwxrwx 2 ubuntu18  
ubuntu18 92 Jan 4 14:34  
mount_HostFiles.sh
```

inode is same for hard link as original file

Soft Link

```
ln -s mount_HostFiles.sh mnt_soft
```

```
$ ls -lia m*
```

```
133913 lrwxrwxrwx 1 ubuntu18  
ubuntu18 18 Jan 28 09:37 mnt_soft -> mount_HostFiles.sh
```

```
138633 -rwxrwxrwx 2 ubuntu18  
ubuntu18 92 Jan 4 14:34  
mount_HostFiles.sh
```

inode is different for soft link than original file

File Properties

```
Struct stat {  
    mode_t st_mode; // file type & mode (permissions)  
    ino_t st_ino; // i-node number (serial number)  
    dev_t st_dev; // device number (file system)  
    dev_t st_rdev; // device number for special files  
    nlink_t st_nlink; // number of links  
    uid_t st_uid; // user ID of owner  
    gid_t st_gid; // group ID of owner  
    off_t st_size; // size in bytes for regular files  
    time_t st_atime; // time of last access  
    time_t st_mtime; // time of last modification  
    time_t st_ctime; // time of last file status change  
    blksize_t st_blksize; // best I/O block size  
    blkcnt_t st_blocks; // number of disk blocks allocated  
}
```

- Stat Functions

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

```
#include <sys/stat.h>
```

- Returns information about a named file

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

- Returns information about already opened file

```
int fstat(int filedes, struct stat *buf);
```

- Returns information about symbolic link, not the referenced file

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

st_ino and st_nlinks

- st_nlinks – provides number of hardlinks
- Create hard link for a file and run this code by passing that filename or hardlink
- [File IO\stat_links.c](#)

st_mode bits

- File Types: S_IFREG – regular file, S_IFDIR – directory, S_IFLINK – symbolic link
- Macro to Check for File Types: S_ISREG(mode) – check if mode value indicates regular file, S_ISDIR(mode) – check if mode indicates directory, S_ISLNK(mode) – check if mode indicates symbolic link
- Permission: Owner
 - S_IRWXU - read, write, execute/search by owner (bitwise OR of S_IRUSR, S_IWUSR, S_IXUSR)
 - S_IRUSR - read permission, owner
 - S_IWUSR - write permission, owner
 - S_IXUSR - execute/search permission, owner

Similarly GRP/G instead of USR/U indicates permission for Group and OTH/O indicates permission for Other Users

[File IO\stat example.c](#)

Useful Library Functions – The same are also available as commands for shell

```
#include <unistd.h>
int access(const char *pathname, int mode) // get accessibility i.e. file permissions
int chown(const char *pathname, uid_t owner, gid_t group) // change owner of file to provided values in owner an group parameters
int truncate(const char *pathname, off_t length) // truncate the file size to length parameter
int link(const char *existingpath, const char *newpath) // creates hard link newpath from the existing file at existingpath
int unlink(const char *pathname) // remove hard link
int symlink(const char *actualpath, const char *sympath) // create symbolic (soft) link
int readlink(const char *restrict pathname, char *restrict buf, size_t bufsize) // read the content of symbolic (soft) link, performs open(), read() and close() function all together
int rmdir(const char *pathname) // removes directory
```

Useful Library Functions – The same are also available as commands for shell

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

```
int remove(const char *pathname) // delete file or directory or unlink  
symbolic link
```

```
int rename(const char *oldname, const char *newname) // rename file  
or directory
```

```
#include <sys/stat.h>
```

```
int chmod(const char *pathname, mod_t mode) // change file access  
permission
```

```
int mkdir(const char *pathname, mode_t mode) // creates new  
directory
```

dirent.h – format of directory entries

- The internal format of directories is unspecified.
- The `<dirent.h>` header defines the following data type through `typedef`:
- `DIR` : A type representing a directory stream.
- It also defines the structure `dirent` which includes the following members:
 - `ino_t d_ino` file inode (aka serial) number
 - `char d_name[]` name of entry
 - The type `ino_t` is defined as described in `<sys/types.h>`.
 - The character array `d_name` is of unspecified size, but the number of bytes preceding the terminating null byte will not exceed `{NAME_MAX}`.

Directory Functions

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

```
DIR *opendir(const char *pathname); // open directory stream and return  
pointer to DIR stream object (DIR is similar to FILE stream object)
```

```
struct dirent *readdir(DIR *dp); // read directory entries from DIR stream  
into pointer to (array of) structure dirent which has inode numbers and  
filenames
```

```
void rewinddir(DIR *dp); // rewind DIR stream to beginning of directory
```

```
int closedir(DIR *dp); // close DIR stream
```

```
long telldir(DIR *dp); // return the current position of DIR stream
```

```
void seekdir(DIR *dp, long loc); //seek to location/position mentioned by loc
```

Implementation of pwd (present working directory) using Directory Functions

Implement present working directory functionality of shell command that displays the path of the current directory.

[File IO\mypwd.c](#)

Buffered vs Unbuffered IO

- Unbuffered I/O: each read/write invokes a system call in the kernel.
 - read, write, open, close, lseek
 - Data Unit: raw byte
- Buffered I/O: data is read/written in optimal-sized chunks from/to disk --> streams
 - standard I/O library written by Dennis Ritchie
 - Data Unit: C data type

Unbuffered IO

Open a File

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h> or <unistd.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);

• Parameters
    • pathname : name of the file with complete path
    • flags:
        • O_RDONLY : read-only access
        • O_WRONLY : write-only access
        • O_RDWR : read-write access
        • O_CREAT: if file doesn't exists then create it
        • O_APPEND : in write mode, don't overwrite but append the new content
        • O_TRUNC : in write mode, truncate the file before writing new content
    • mode:
        • 0600 (i.e. -rwx-----) : read-write access for current user, no access for group users or other users
        • 0644 (i.e. -rw-r--r--) : read-write access for current user, read-only access for group users and other users
```

Open a File

- `open()` returns an integer:
 - `-1` means error i.e. file could not be opened
 - `>= 0` : this is a “file descriptor” of a open file. Save it in a variable, you will need to pass it to all subsequent file related functions such as `read`, `write` etc
- You don’t need to specify file mode unless you will be creating a new file

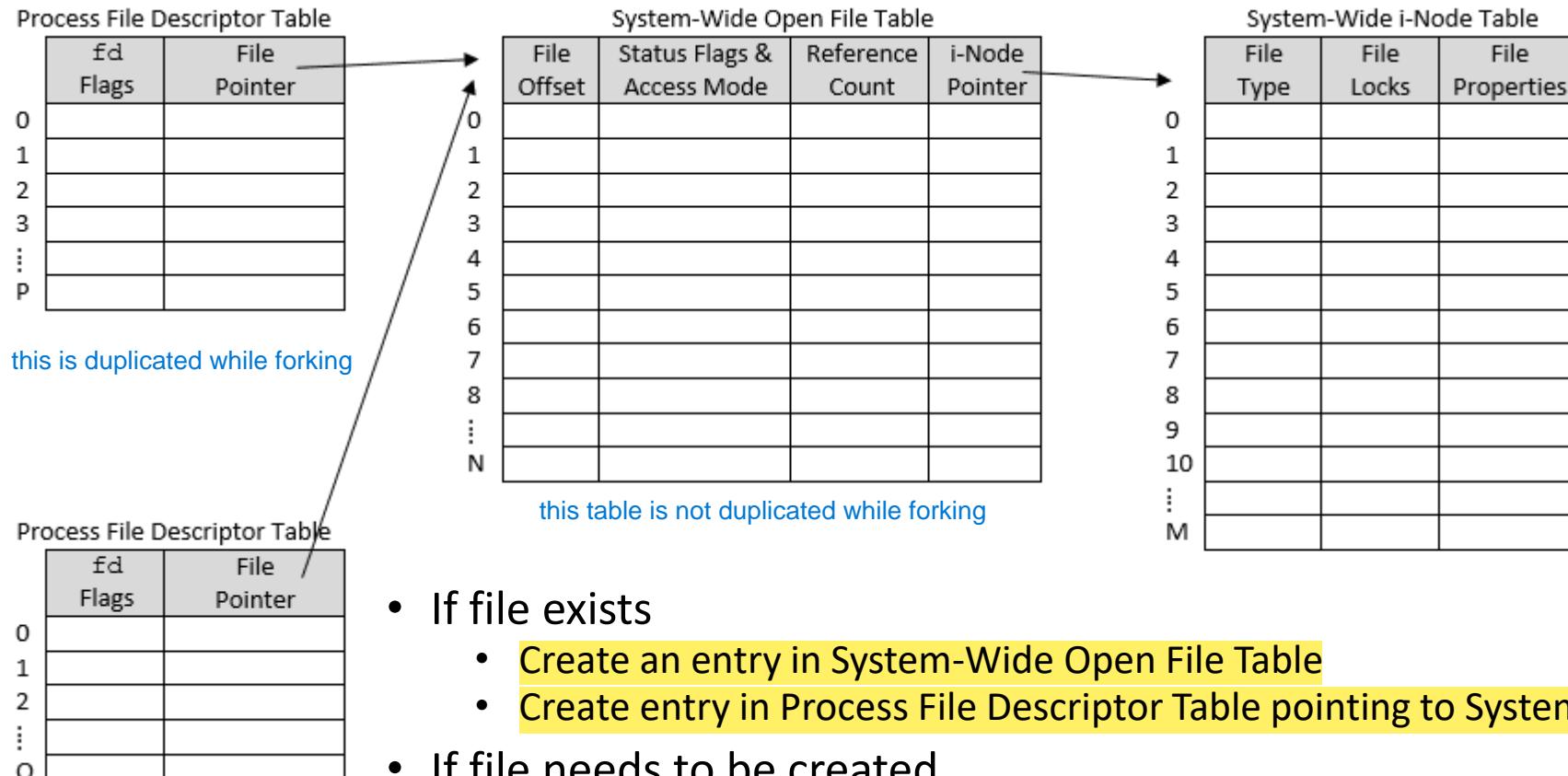
• `fd = open("this_file_already_exists", O_RDONLY);`

- You can combine multiple flags together

• `fd = open("foo", O_RDWR | O_CREAT, 644);`

DR

Operating System Tables



- If file exists
 - Create an entry in System-Wide Open File Table
 - Create entry in Process File Descriptor Table pointing to System-Wide Table
- If file needs to be created
 - Disk FCB (i-node) is created first and then loaded it in System-Wide i-node Table
 - Then same two steps from above

dup and dup2

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

int dup(int oldfd); → creates a copy of the file descriptor oldfd, using the lowest-numbered unused file descriptor for the new descriptor

i.e. row number of process file descriptor table
int dup2(int oldfd, int newfd); → create a new file descriptor newfd pointing to the same physical file as oldfd

Returns:

New file descriptor on success

-1 on error with **errno** variable set to check for exact error

Read From a File

```
#include <sys/types.h>
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count); → difference betn size_t &
ssize_t?
```

- Parameters:
 - fd : file descriptor of the file which you want to read from
 - buf : buffer where the file content will be stored after reading
 - count : number of bytes to read
- **read()** returns an integer:
 - -1 means error reading the file
 - ≥ 0 : number of bytes that were actually read from the file. If return value is less than the value in count (i.e. number of bytes to be read) then it is inferred that End of File has reached

Difference between open and dup

Process File Descriptor Table

fd	File Flags	File Pointer
0		
1		
2		
3	6	
4	2	
5	8	
6	2	
7		

System-Wide Open File Table

File Offset	Status Flags & Access Mode	Reference Count	i-Node Pointer
0			
1			
2	0 ₁ 2	2	7
3			
4			
5			
6	0 ₁	1	3
7			
8	0	1	3
⋮			
N			

System-Wide i-Node Table

File Type	File Locks	File Properties
0		
1		
2		
3	Regular	...
4		
5		
6		
7	Regular	...
8		
9		
10		
⋮		
M		

```
fdA1 = open("fileA.txt", O_RDONLY);
read(fdA1, &c, 1);
fdB = open("fileB.txt", O_RDONLY);
read(fdB, &c, 1);
fdA2 = open("fileA.txt", O_RDONLY);
fdBdup = dup(fdB);
read(fdBdup, &c, 1);
```

Write Into a File

```
#include <sys/types.h>
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

- Parameters:
 - fd : file descriptor of the file which you want to write to
 - buf : buffer from where the content will be written to the file
 - count : number of bytes to write
- **write()** returns an integer:
 - -1 means error writing the file
 - ≥ 0 : number of bytes were actually written into the file which should be the same as value in count. If return value is less than the value in count then you may have encountered error like not sufficient disk space etc.

Close a File

```
#include <unistd.h>  
int close(int fd);
```

- Parameters:
 - fd : file descriptor of the file which you want to close
- close() returns an integer:
 - -1 means error closing the file
 - 0 : OK (i.e. file successfully closed)
- Don't forget to close the file once you have finished using the file otherwise you leave orphan file descriptors in the system. (Normally OS will check all the open files at the time when program execution ends and will closes them but we must close them in our program.)

Difference between open and dup

Process File Descriptor Table

fd	File Flags	File Pointer
0		
1		
2		
3	6	
4	2	
5	8	
6	2	
7		

System-Wide Open File Table

File Offset	Status Flags & Access Mode	Reference Count	i-Node Pointer
0			
1			
2	0 ₁ 2	2	7
3			
4			
5			
6	0 ₁	1	3
7			
8	0	1	3
⋮			
N			

System-Wide i-Node Table

File Type	File Locks	File Properties
0		
1		
2		
3	Regular	...
4		
5		
6		
7	Regular	...
8		
9		
10		
⋮		
M		

```
fdA1 = open("fileA.txt", O_RDONLY);
read(fdA1, &c, 1);
fdB = open("fileB.txt", O_RDONLY);
read(fdB, &c, 1);
fdA2 = open("fileA.txt", O_RDONLY);
fdBdup = dup(fdB);
read(fdBdup, &c, 1);
```

What happens when we do close(fdA1)?
 How about close(fdB)?

Special Files

- When any program starts executing, 3 file descriptors are created automatically
 - You can use them as you like
 - You are not obliged to close them after you finish using them
- These file descriptors are:
 - 0: it represents standard input (generally keyboard) for your program. When user types input on keyboard it is read using file descriptor 0. It is read-only file descriptor which can't be written
 - 1: it represents standard output (generally display screen) for your program. When any message is written file descriptor 1, it is displayed on the screen
 - 2: it represents standard error (generally display screen) for your program. You can use this to display any error messages to screen.

Seeking a File

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

Returns: new file offset if OK, -1 on error

whence: where to start the offset i.e. SEEK_SET, SEEK_CUR, SEEK_END

SEEK_SET – file offset set to offset

SEEK_CUR – file offset set to Current location + offset

SEEK_END – file offset set to file size + offset (i.e. increasing the file size)

Examples

- Example with lseek : [File IO\FileIO Example1.c](#)
- Example with multiple File Descriptor to the same file and using STDOUT file descriptor with write() system call :
[File IO\FileIO Example2.c](#)

Buffered IO

Standard I/O Library

- Difference from Unbuffered File I/O
 - File Pointers (`FILE *`) vs File Descriptors (`int`)
 - `fopen` vs `open`
 - When file is opened or created, a stream is associated with the file.
 - `FILE` object
 - File descriptor, buffer size, # of remaining chars, an error flag
 - `stdin`, `stdout`, `stderr` defined in `<stdio.h>`
 - `STDIN_FILENO`, `STDOUT_FILENO`, `STDERR_FILENO` (unbuffered file descriptors are defined in `<unistd.h>`) as IEEE POSIX (Portable Operating System Interface) standard

Buffering

- Goal
 - Use minimum number of read and write calls
- Types
 - Fully Buffered
 - Actual I/O occurs when the buffer is filled up
 - A buffer is automatically allocated when first I/O operation is performed on a stream
 - flush: standard I/O library vs terminal driver
 - Line Buffered
 - Perform I/O when a newline character is encountered – usually for terminals
 - Caveat
 - Filling of buffer could trigger I/O
 - Flushing all line buffered output if input requested
 - Unbuffered
 - Expect to output ASAP e.g. when write() is called
 - Or when using stdout or stderr

Buffering

By default, buffering is done automatically when file is opened but in case you need a specific buffer implementation you can use this functions

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

```
void setbuf(FILE *fp, char *buf);
```

```
void setvbuf(FILE *fp, char *buf, int mode, size_t size);
```

- Full/line buffering if buf is NOT NULL (BUFSIZ)
 - Terminals
- Mode _IOFBF, _IOLBF, _IONBF (<stdio.h>)
 - Optional size → st_blksize (stat())
- #define BUFSIZ 1024 (<stdio.h>)
- They must be called before any operation is performed on the stream

Buffering

- ANSI C requirements
 - Fully buffered for stdin and stout unless interactive devices are referred to.
 - 4.3+BSD – line buffered
 - Stderr is never fully buffered

```
#include <stdio.h>  
int fflush(FILE *fp);  
• All output stream are flushed if fp == NULL
```

Opening a Stream

- `#include <stdio.h>`
- `FILE *fopen(const char *pathname, const char *type);`
- opens a specified file
- types:
 - r : open for reading
 - w : create for writing or truncate to 0
 - a : open or create for writing at the end of file
 - r+ : open for reading and writing
 - w+: create for reading and writing or truncate to 0
 - a+ :open or create for reading and writing at the end of file
- use b to differentiate text vs binary , e.g. rb, wb ..etc

fopen() vs open() flags

fopen() flags	open() flags
r	O_RDONLY
w	O_WRONLY O_CREAT O_TRUNC
a	O_WRONLY O_CREAT O_APPEND
r+	O_RDWR
w+	O_RDWR O_CREAT O_TRUNC
a+	O_RDWR O_CREAT O_APPEND

Restrictions

Type	r	w	a	r+	w+	a+
File exists?	Y			Y		
Truncate		Y			Y	
R	Y			Y	Y	Y
W		Y	Y	Y	Y	Y
W only at end			Y			Y

- When a file is opened for reading and writing:
 - Output cannot be directly followed by input without an intervening *fseek, fsetpos, or rewind*
 - Input cannot be directly followed by output without an intervening *fseek, fsetpos, or rewind*

Setting File Position

- **long ftell(FILE * stream)**
 - obtains the current value of the file position indicator for the stream pointed to by stream.
- **int fseek (FILE * stream, long int offset, int whence);**
 - stream: Pointer to a FILE object that identifies the stream.
 - offset : Binary files- Number of bytes to offset from *whence*. Text files- Either zero, or a value returned by *ftell()*.
 - whence: SEEK_SET - Beginning of file, SEEK_CUR - Current position of the file pointer, SEEK_END - End of file *
- **int rewind(FILE * stream);**
 - Rewind function sets the file position indicator for the stream pointed to by stream to the beginning of the file. It is equivalent to
`(void)fseek(stream, 0L, SEEK_SET)`

Setting File Position

- `fgetpos()` and `fsetpost()` alternate interfaces equivalent to `ftell()` and `fseek()` with whence set to `SEEK_SET`
- `int fgetpos(FILE *stream, fpos_t *pos);`
- `int fsetpos (FILE * stream, const fpos_t * pos);`
 - Stream: pointer to a `FILE` object that identifies the stream
 - pos: Pointer to a `fpos_t` object containing a position previously obtained with `fgetpos`.

Closing a Stream

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

```
int fclose(FILE *fp);
```

- Flush buffered output
- Discard buffered input
- All I/O streams are closed after the process exists

- setbuf or setvbuf to change the buffering of a file before any operation on the stream

Reading and Writing from/to Streams

- Unformatted I/O
 - Character-at-a-time I/O e.g. `getc()`
 - Buffering handled by standard I/O library
 - Line-at-a-time I/O e.g. `fgets()`
 - Buffer limit might need to be specified
 - Direct I/O e.g. `fread()`
 - Read/write a number of objects of a specified size
 - An ANSI C term e.g. = object-at-a-time I/O

Reading a Char

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

```
int getc(FILE *fp); - Can be used as a macro or a function
```

```
int fgetc(FILE * fp); - Can be used only as a function
```

```
int getchar(void);
```

- `Getchar() == getc(stdin)`
- unsigned char converted to int in returning

Error/EOF Check

```
#include <stdio.h>

int ferror(FILE *fp); - test error indicator

int feof(FILE *fp); - test if end of file

void clearerr(FILE *fp);

int ungetc(int c, FILE *fp); - puts the char back in the stream which was
read using getc

    • An error flag and EOF flag for each FILE
    • No pushing back of EOF (i.e. -1)
        • No need to be same char read!
```

Writing a char

```
#include <stdio.h>

int putc(int c, FILE *fp);

int fputc(int c, FILE *fp);

int putchar(int c);
    • putchar(c) == putc(c, stdout)
    • Differences between putc and fputc
        • putc() can be implemented as a macro hence can't be used in function pointer
```

Example

getchar(), putchar() example : [File IO\FileIO Example3.c](#)

Copy File Example: [File IO\copy_file.c](#)

./copy_file.out abc.txt def.txt 2>&1

2>&1 can be used to redirect both stderr and stdout to stdout

Line-at-a-Time I/O

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

```
char *fgets(char *buf, int n, FILE *fp);
```

- Include '\n' and be terminated by null
- Could return a partial line if the line is too long

```
char *gets(char *buf);
```

- Read from stdin
- No buffer size is specified → overflow
- *buf does not include '\n' and is terminated by null

Line-at-a-Time I/O

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

```
char *fputs(const char *str, FILE *fp);
```

- Include '\n' and be terminated by null
- No need for line-at-a-time output

```
char *puts(const char *str);
```

- *str does not include '\n' and is terminated by null
- puts then writes '\n' to stdout

Example 2

fgets() fputs() example: [File IO\FileIO Example4.c](#)

Copy file example: [File IO\copy_file.c](#)

Standard I/O Efficiency

- Measure time spent in user mode and kernel mode using : time <program>
- Copy file [File IO\IOefficiency.txt](#) of size 2,100,000 bytes (@2MB) to [File IO\IOefficiency1.txt](#) by redirecting stdin and stdout:
[File IO\IOefficiency.c](#)

time ./IOefficiency.out <IOefficiency.txt >IOefficiency1.txt

Functions	Total (Real)	User Time	Kernel Time	
fgets(), fputs() (5MB buffer)	0m0.144s	0m0.006s	0m0.057s	it uses system buffer
fgetc(), fputc()	0m0.190s	0m0.008s	0m0.100s	
read(), write()	6m27.971s	0m0.687s	2m35.339s	as it is unbuffered it needs to go to disk

Reading fixed sized items

```
#include <stdio.h>
size_t fread(void *restrict ptr, size_t size, size_t nitems, FILE *restrict
stream);
```

- ptr – pointer to buffer to which data will be read from stream
- size – size of individual item
- nitems – number of items
- stream – FILE stream object

What is the difference in the below calls?

```
elements_read = fread(buf, sizeof(buf), 1, fp);
bytes_read = fread(buf, 1, sizeof(buf), fp);
```

Writing fixed size items

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

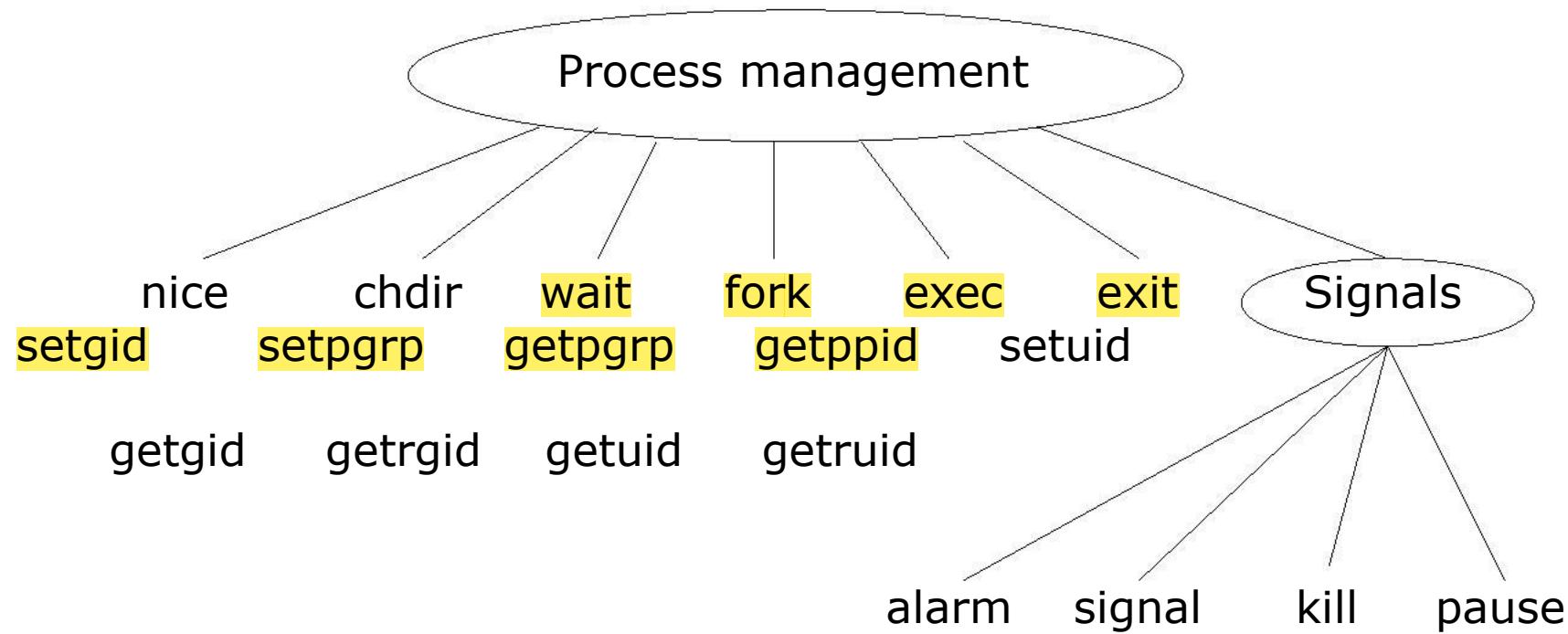
- ptr – pointer to buffer from which data has to be written to stream
- size – size of individual item
- nitems – number of items
- stream – FILE stream object

Reading and Writing C Data types

- `int fscanf(FILE *stream, const char *format, ...);`
- `int fprintf(FILE *stream, const char *format, ...);`
- Same as scanf and printf except operations are performed on a given file stream

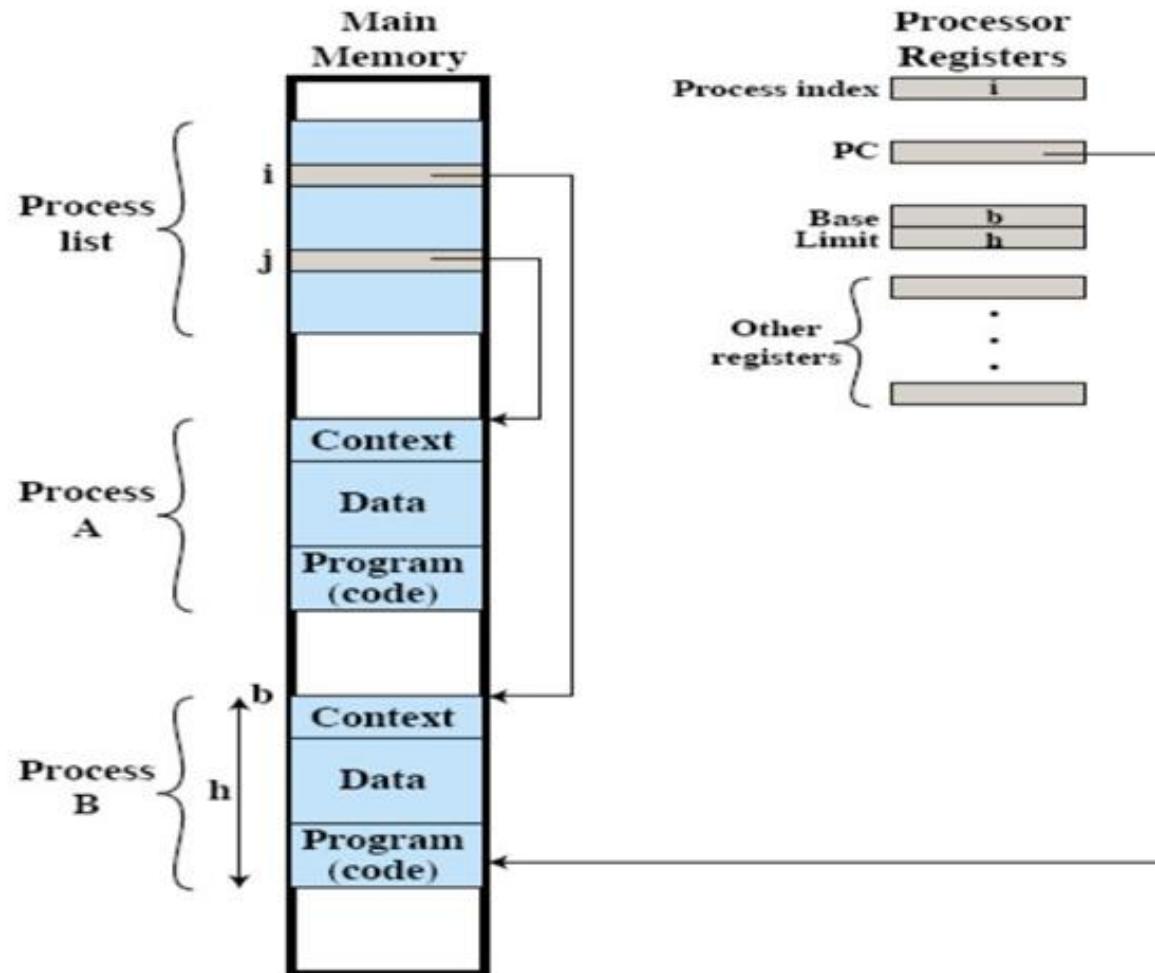
Systems
Software/Programming
Process Management

Process Management System Call Hierarchy



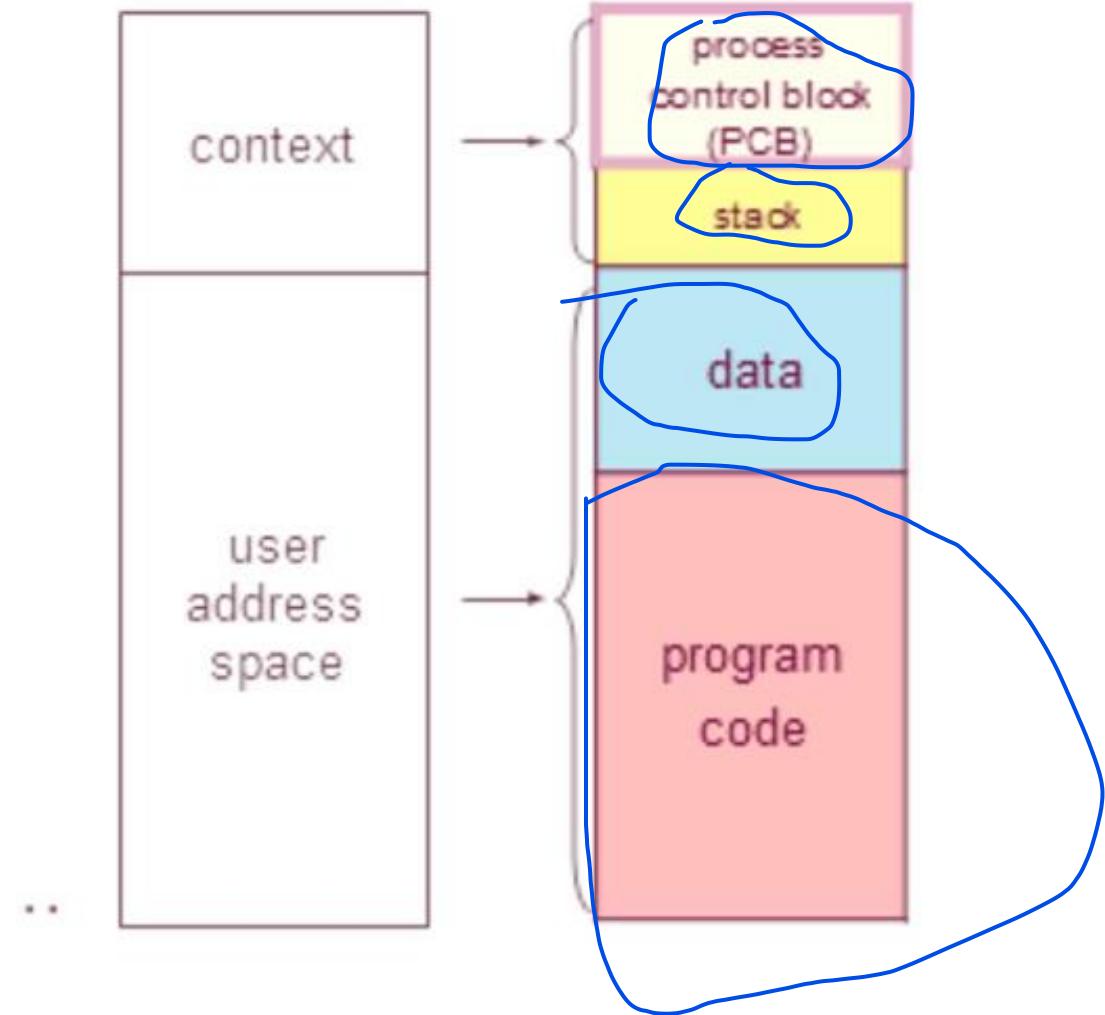
Process Concept

- A process is program in execution. **Multiple instances of the same program are different processes**
- Process memory image contains 3 components in 2 address spaces:
 - User Address Space
 - An executable program code
 - Associated data needed by program
 - Kernel Address Space
 - Execution context needed by OS to manage the process (Process ID, CPU registers, CPU time, stack, open files, memory space for code data and stack, signals etc)

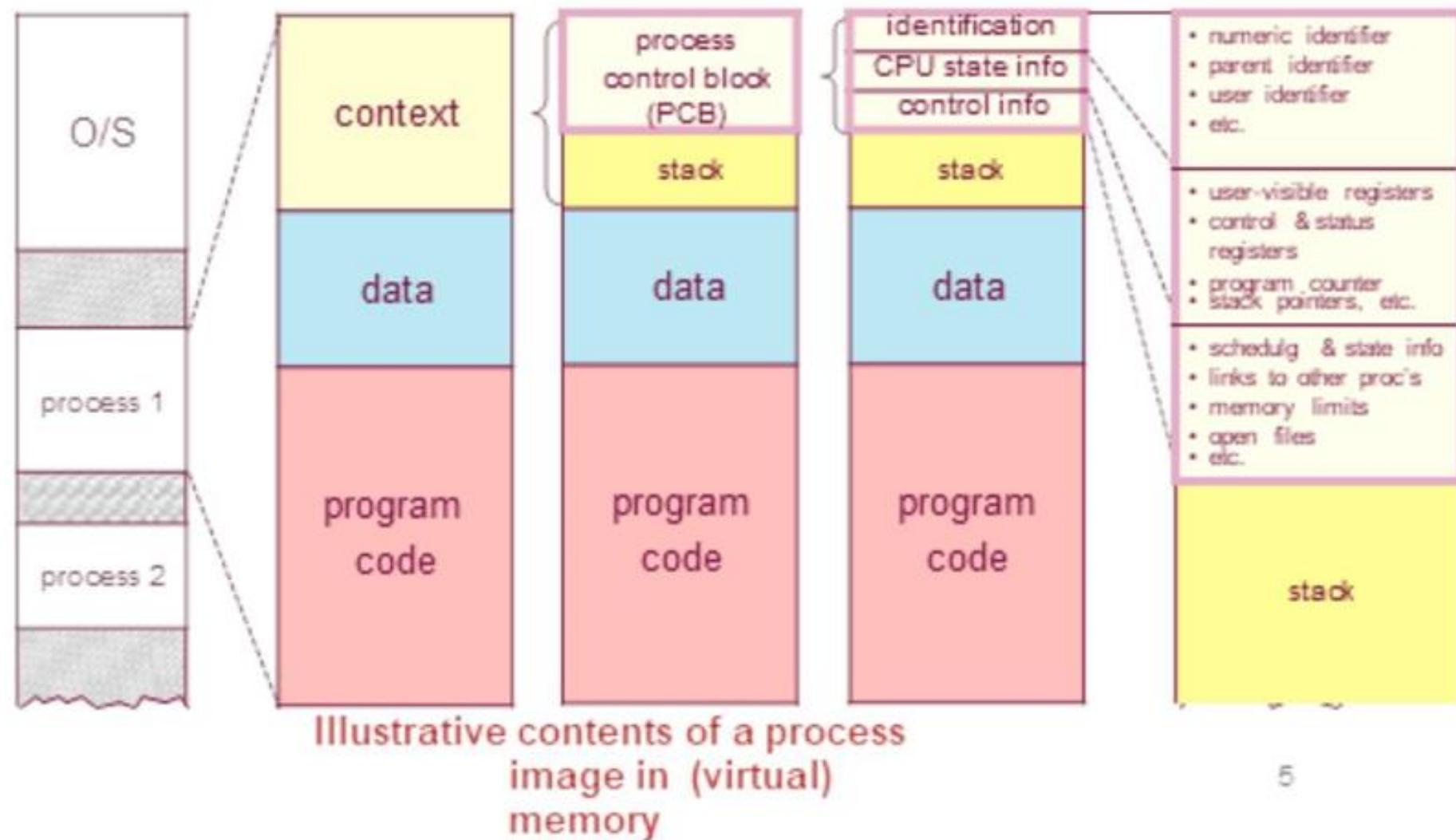


Process Control Block (PCB)

- For each process PCB is maintained in system's process table array or linked list
- PCB is included in context along with stack
- Typical PCB contains:
 - Process ID, Parent Process ID
 - Process State
 - CPU State: CPU register contents, PSW (Process Status Word)
 - Priority and other Scheduling Info
 - Pointers to different memory areas
 - Open file information
 - Signals and signal handler info
 - Various accounting info like CPU time etc
 - Many other OS specific fields can be there e.g. Linux PCB has 100+ fields



Example of process and PCB in memory



Operations on Process

- **Process Creation**
 - Data structures like PCB set up and initialized
 - Initial resources allocated and initialized if needed
 - Process added to ready queue (queue of processes ready to run)
- **Process Scheduling**
 - CPU is allotted to process and process start running
- **Process Termination**
 - Process is removed
 - Resources are reclaimed
 - Some data may be passed to parent process such as exit status
 - Parent process may be informed (e.g. **SIGCHLD signal in Unix**)

Events Leading to Process Creation

- System boot – During system booting several background processes or daemons are started
- User Request – When command is executed on CLI shell or double-click on application
- A process can spawn another i.e. child process using fork() e.g. server can create new process for each client request or init daemon waits for login and spawns new shell
- Batch system takes next job to process

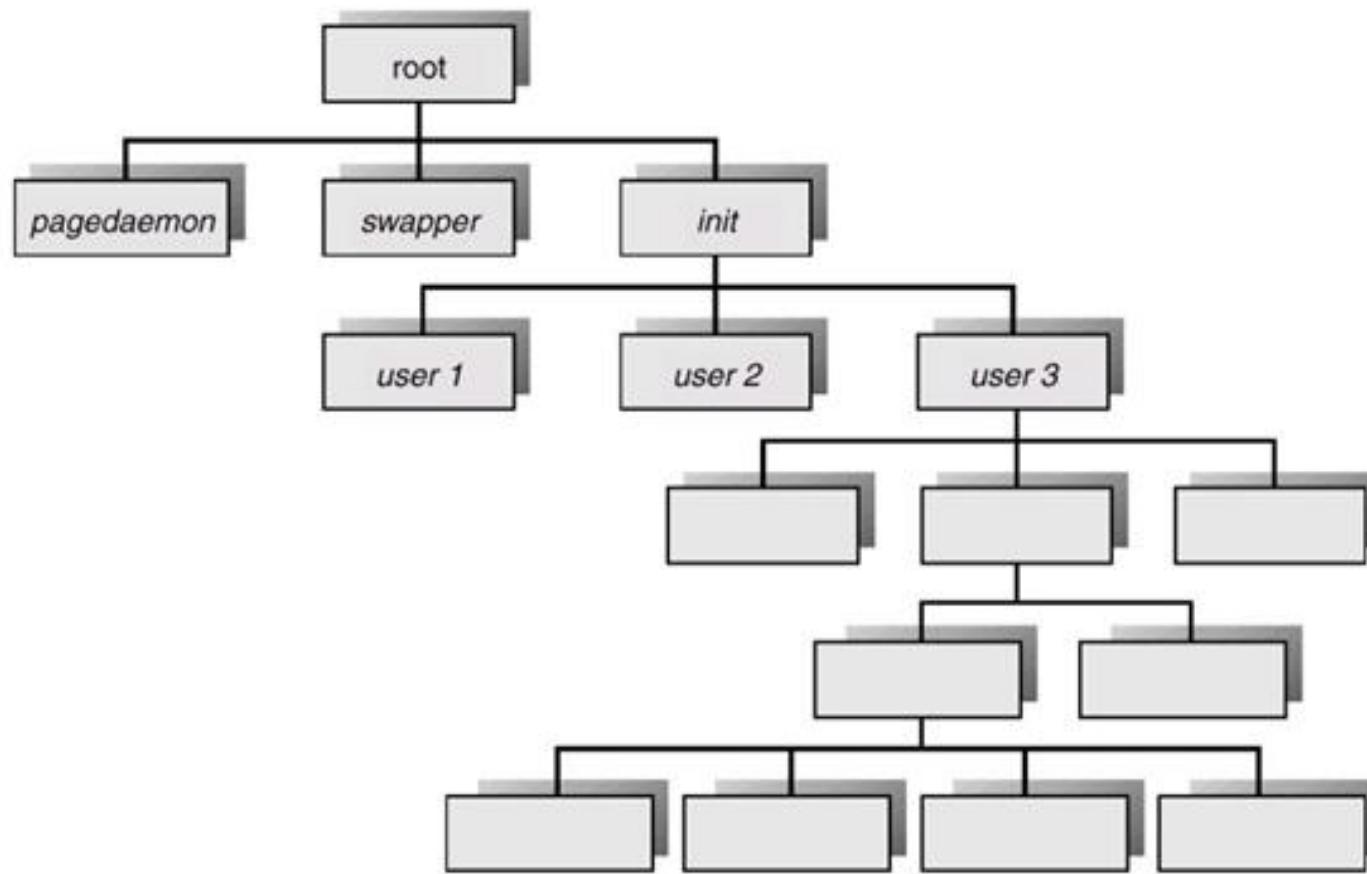
Stages of Linux Boot Process

- BIOS: Basic Input/output System → performs system integrity checks, search boot loader on cd-rom or disk, and executes MBR boot loader
- MBR: Master Boot Record → available in 1st sector of bootable disk, executes GRUB boot loader
- GRUB: GRand Unified Bootloader → if multiple kernel images are installed you can select using GRUB , executes selected Kernel
- Kernel: Kernel → Mounts the root file system, executes Init (init is the first program to run)
- Init: Init → executes run level programs depending on runlevel (runlevels: halt, single user, multiuser without NFS, Full multiuser etc)
- Runlevel → Runlevel programs are executed from etc/rc.d/rc*.d

Runlevel Definition

- 0 - System halt; no activity, the system can be safely powered down.
- 1 - Single user; rarely used.
- 2 - Multiple users, no NFS (network [filesystem](#)); also used rarely.
- 3 - Multiple users, [command line](#) (i.e., all-text mode) [interface](#); the standard runlevel for most Linux-based [server](#) hardware.
- 4 - User-definable
- 5 - Multiple users, [GUI](#) (graphical user interface); the standard runlevel for most Linux-based desktop systems.
- 6 - [Reboot](#); used when restarting the system.

Process Tree on Unix System



- Swapper is scheduler
- Init is the root of all user processes
- Pagedaemon is responsible for virtual page management

Events Leading to Process Termination

- Process executes last statement and asks operating system to terminate it using `exit()` function
- Process encounter fatal error like divide by zero, I/O error, memory allocation errors etc
- Parent may terminate execution of child process by executing `kill` (`SIGKILL` signal) function for some specific reason
 - Task assigned to child is no longer needed
 - Child has exceeded the allocated resources
- Parent is exiting
 - Some OS may not allow child to continue if parent terminates

Process Relation Between Parent and Child Processes

- Resource sharing possibilities
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
- Execution possibilities
 - Parent and children execute concurrently
 - Parent waits until children terminate
- Memory address space possibilities
 - Address space of child duplicate of parent
 - Child has a new program loaded into it

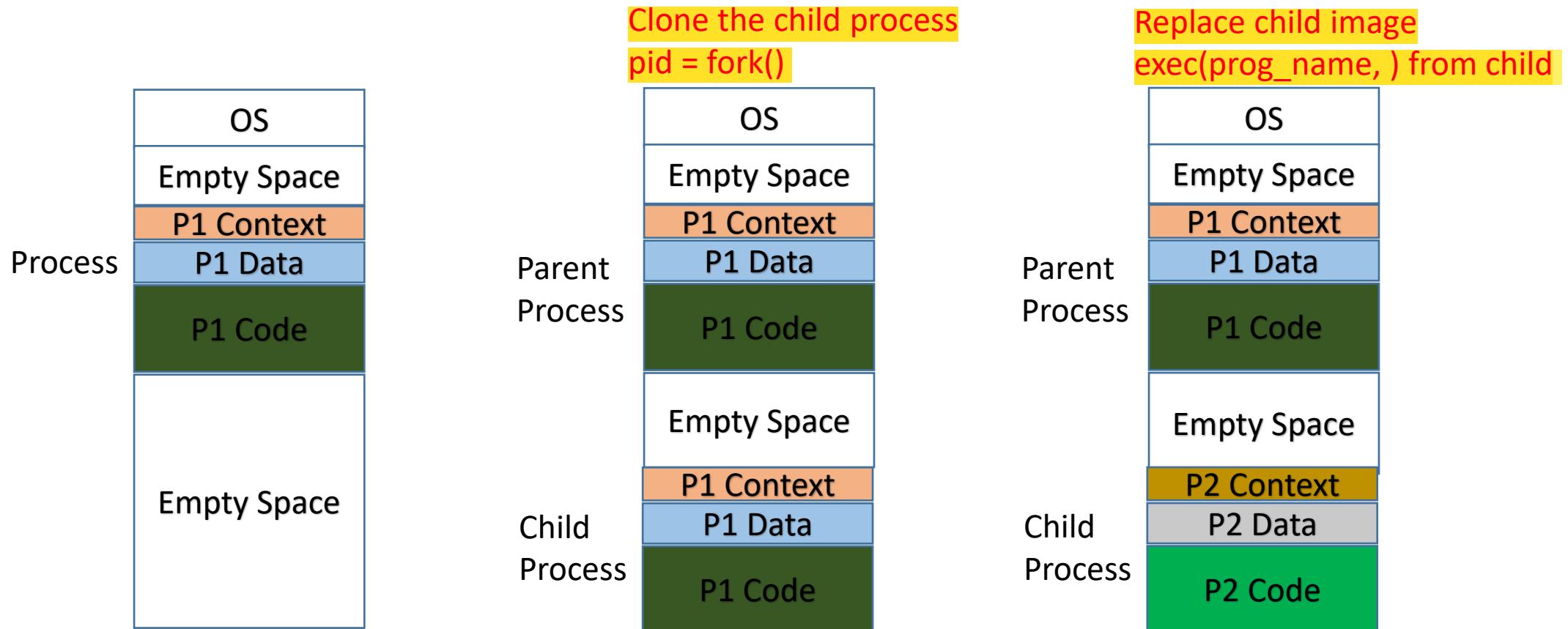
Process Management System Call

System Call Signature	Purpose of System Call	Return Values
pid_t getpid(void)	Get process ID	Returns process ID of calling process
pid_t getppid(void)	Get parent's process ID	Returns parent process ID of calling process
pid_t fork(void)	Create child process by duplicating memory of parent process. Child gets copy of data space etc. Now uses Copy-On-Write (COW)	Returns 0 in child, process ID of child in parent, -1 on error
pid_t vfork(void)	Now Obsolete: Earlier fork() used to copy a complete memory of parent to child so vfork was used as optimized method	Returns 0 in child, process ID of child in parent, -1 on error
void exit(int)	Terminated the process	Parent of process will receive the success value passed as int

Process Management System Call

System Call Signature	Purpose of System Call	Return Values
pid_t wait(int *statloc)	Wait for a child process to Terminate	Returns process ID of child that terminated and statloc indicates returned status
pid_t wait(pid_t pid, int *statloc..)	Wait for child with given pid	
int exec???(const char pathname, const char *arg0, ...)	Replaces the current process memory with new process to be executed in pathname	Return -1 on error and no return on success

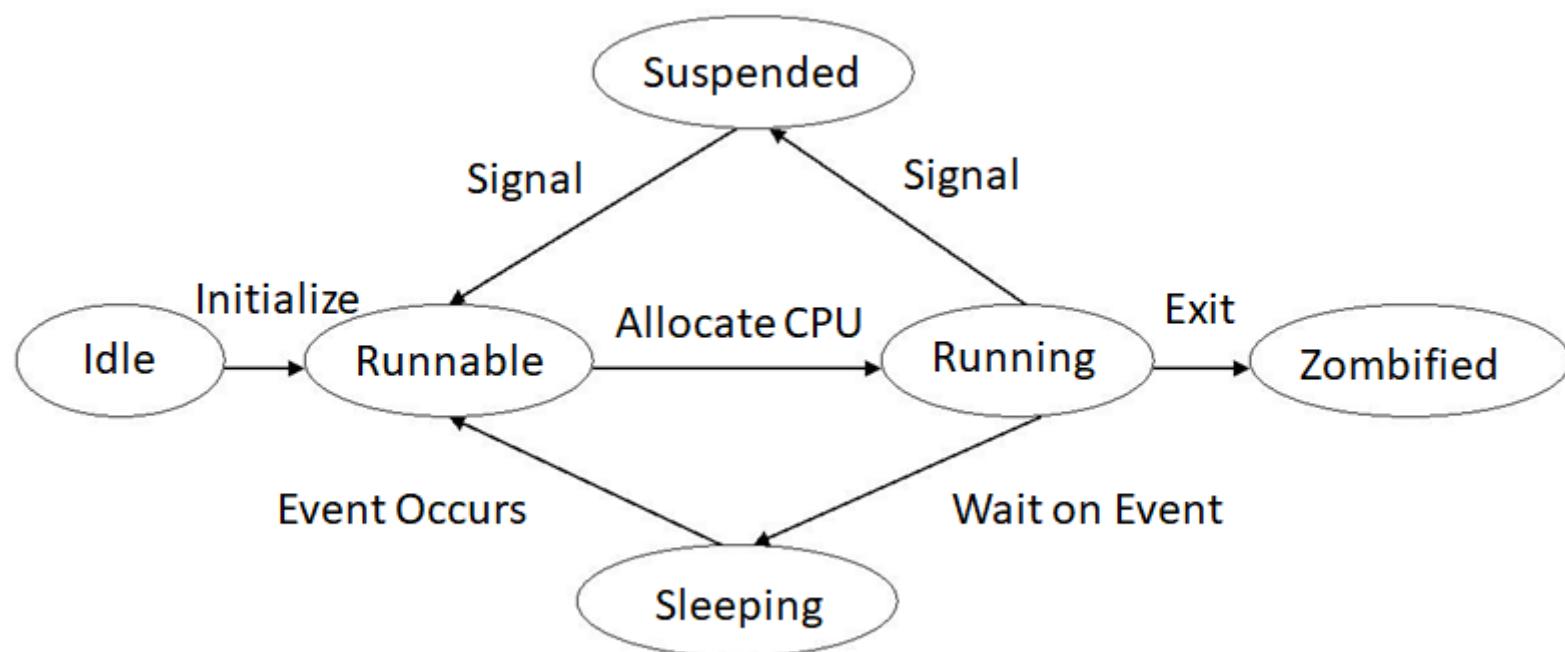
fork() and exec() memory changes



Process States (6 State Model)

- Process changes state as it executes:
 - Idle → process is being created by fork() system call and is not yet runnable
 - Runnable → process is waiting for CPU to start running
 - Running → process is currently running executing instructions
 - Sleeping → process is waiting for an event to occur e.g. if process has executed read() system calls, it will go to sleep until I/O request is complete
 - Suspended → process has been “frozen” by signal such as SIGSTOP, it will resume when SIGCONT signal is received e.g. Ctrl-Z suspends all the processes of foreground job
 - Zombified → process has terminated but has not yet returned its exit code to its parent. The process remains in zombie state until parent accepts return code using wait() system call

Process State Transition



System Calls fork(), getpid() and getppid() example

ProcessManagement\fork example.c

```
#include <stdio.h>
main()
{ int pid;
  printf("I'm the original process with PID %d and PPID %d. \n", getpid(), getppid() );
  pid = fork(); /* Duplicate. Child and parent continue from here */
  if ( pid!= 0 ) /* pid is non-zero, so I must be the parent */ ← Parent and Child execute
from this point
  {
    printf("I'm the parent process with PID %d and PPID %d. \n", getpid(), getppid() );
    printf("My child's PID is %d \n", pid );
  }
```

System Calls fork(), getpid() and getppid() example

```
else /* pid is zero, so I must be the child */  
{  
    printf("I'm the child process with PID %d and PPID %d. \n",  
getpid(), getppid() );  
}  
printf("PID %d terminates. \n", getpid() ); /* Both processes execute  
this */  
}
```

System Calls fork(), getpid() and getppid() example

\$ fork_example

I'm the original process with PID 13292 and PPID 13273.

I'm the parent process with PID 13292 and PPID 13273.

My child's PID is 13293.

I'm the child process with PID 13293 and PPID 13292.

PID 13293 terminates. ---> child terminates.

PID 13292 terminates. ---> parent terminates.

WARNING:

it is dangerous for a parent process to terminate without waiting for the death of its child.

The only reason our program doesn't wait for its child to terminate is because we haven't yet used the "wait()" system call!.

Parent and Child Process Variable scope

ProcessManagement\fork var scope example.c

- Before fork there is only one process so only one copy of variable
- Immediately after fork, two copies of the variables are created (one for parent and the other for child).
 - After fork any changes to the variable is done in the local copy of the respective process, hence the other process variable value will be different

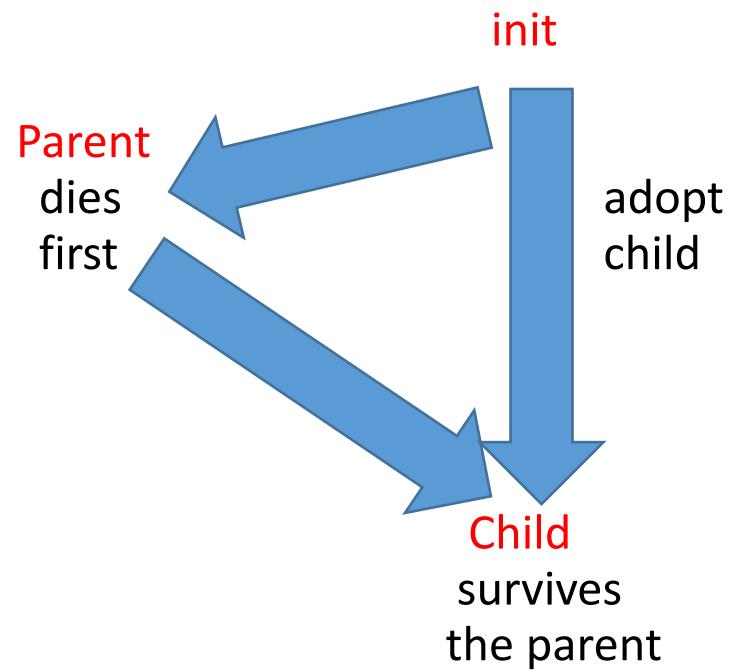
What happens with this code?

```
int main(void)
{
    fork();
    fork();
    fork();
}
```

How many total processes will be created?

Orphan Process

- If parent process does not wait for child and it first terminates leaving child process orphan
 - Orphan processes are adopted by init process which started the parent (i.e. parent of parent)



Orphan Process Example

ProcessManagement\orphan child.c

```
else /* pid is zero, so I must be the child
*/
{
    sleep(10); // add sleep so child process
    will terminate later than parent

    printf("I'm the child process with • PID
13364 terminates. PID %d and PPID %d. \n",
getpid(), getppid() );

}

printf("PID %d terminates. \n", getpid() );
/* Both processes execute this */
}
```

\$ orphan ---> run the program.
I'm the original process with PID 13364 and PPID 13346.
I'm the parent process with PID 13364 and PPID 13346.
PID 13364 terminates.
I'm the child process with PID 13365 and PPID 1. ---> orphaned!
PID 13365 terminates.
\$

Note the change in PPID for child processes

System Call wait() to avoid orphans

ProcessManagement\wait example.c

```
#include <stdio.h>
main()
{ int pid, status;
  printf("I'm the original process with PID %d and PPID %d. \n", getpid(), getppid() );
  pid = fork(); /* Duplicate. Child and parent continue from here */
  if ( pid!= 0 ) /* pid is non-zero, so I must be the parent */
  {
    printf("I'm the parent process with PID %d and PPID %d. \n", getpid(), getppid() );
    printf("My child's PID is %d \n", pid );
    childPid = wait( &status ); // add wait in parent process to wait for child process, child will not
    // become orphan
  }
  else.....
```

Zombie Process

ProcessManagement\zambi example.c

```
#include <stdio.h>
main()
{
    int pid;
    pid = fork(); /* Duplicate */
    if ( pid!= 0 ) /* Branch based on return value from fork() */
    {
        while (1) /* Never terminate, and never execute a wait() */
            sleep(1000);
    }
    else
    {
        exit(42); /* Exit with a silly number */
    }
}
```

\$ zombie & ---> execute the program in the background using &
[1] 13545

\$ ps ---> obtain process status.
PID TT STAT TIME COMMAND
13535 p2 S 0:00 -ksh (ksh) ---> the shell.
13545 p2 S 0:00 zombie ---> the parent process.
13546 p2 Z 0:00 <defunct> ---> the zombie child.
13547 p2 R 0:00 ps

\$ kill 13545 ---> kill the parent process.
[1] Terminated zombie

\$ ps ---> notice that the zombie is gone now.
PID TT STAT TIME COMMAND
13535 p2 S 0:00 -ksh (ksh)
13547 p2 R 0:00 ps

Race Condition

ProcessManagement\racecondition example.c

You can call this as critical section

```
static void charatertime(char *str)
{
char *ptr;
int c;
setbuf(stdout, NULL);
for (ptr=str;c=*ptr++;) putc(c,stdout);
}
main()
{
pid_t pid;
if ((pid = fork())<0) printf("fork error!\n");
else if (pid ==0) charatertime("12345678901234567890\n");
else charatertime("abcdefghijklmnopqrstuvwxyz\n");
}
```

```
$ test_fork
```

```
12345678901234567890
```

```
abcdefghijklmнопqrstuvwxyz
```

```
$ test_fork
```

```
12a3bc4d5e6f78901g23hi4567jk890
```

```
lmnopqrstuvwxyz
```

Need to have parent wait for child or child wait for parent to complete the critical section code. This can be done using signals which will study in next chapter

Additional Status Info from wait() System Call

ProcessManagement\child exit reason example.

C

childpid = wait(&wstatus); → returns the exit status from child which can further be inspected using these macros

WIFEXITED(wstatus) → returns true if child terminated normally

WEXITSTATUS(wstatus) → returns exit status (least significant 8 bits)

WIFSIGNALED(wstatus) → returns true if child process was terminated by a signal

WTERMSIG(wstatus) → returns the number of signal

WCOREDUMP(wstatus) → returns true if child produced a core dump

WIFSTOPPED(wstatus) → returns true if child was stopped by a signal

WSTOPSIG(wstatus) → returns the signal number which caused child to stop

WIFCONTINUED(wstatus) → returns true if child was resumed with SIGCONT signal

exec() family of System Calls

When fork() creates a child process with a copy of same code, data etc as parent process but if you need to run another process as child process then →

A process may replace its current code, data, and stack with those of another executable by using one of the “exec()” family of system calls

When a process executes an “exec()” system call, its PID and PPID numbers stay the same - only the code that the process is executing changes.

System Call:

int execl(const char* path, const char* arg0, const char* arg1,..., const char* argn, NULL)

int execv(const char* path, const char* argv[])

int execlp(const char* path, const char* arg0, const char* arg1, ..., const char* argn, NULL)

int execvp(const char* path, const char* argv[])

The “exec()” family of system calls replaces the calling process’ code, data, and stack with those of the executable whose pathname is stored in path.

Difference in exec() System Calls

- “execp()” and “execvp()” use the \$ PATH environment variable to find path.
 - If the executable is not found, the system call returns a value of -1; otherwise, the calling process replaces its code, data, and stack with those of the executable and starts to execute the new code.
- “execl()” and “execlp()” invoke the executable with the string arguments pointed to by arg1 through argv.
 - argv[0] must be the name of the executable file itself, and the list of arguments must be null terminated.
- “execv()” and “execvp()” invoke the executable with the string arguments pointed to by argv[1] to argv[n], where argv[n+1] is NULL.
 - argv[0] must be the name of the executable file itself.

System Call exec() example

ProcessManagement\exec example.c

the program displays a small message and then replaces its code with that of the “ls”.

```
#include <stdio.h>
main()
{
    printf("I'm process %d and I'm about to exec an ls -l \n", getpid());
    execl( "/bin/ls", "ls", "-l", NULL ); /* Execute ls */
    printf("This line should never be executed \n");
}
```

\$ myexec ---> run the program.

I'm process 13623 and I'm about to exec an ls -l

total 125

-rw-r--r-- 1 glass 277 Feb 15 00:47 myexec.c

-rwxr-xr-x 1 glass 24576 Feb 15 00:48 myexec

Background Process using fork() and exec() System Calls

[ProcessManagement\run exec in child.c](#)

```
#include <stdio.h>
int main( int argc, char* argv[] )
{
if ( fork() == 0 ) /* Child */
{
execvp( argv[1], &argv[1] ); /* Execute other program */
fprintf( stderr, "Could not execute %s
\n", argv[1] );
}
}
```

\$./background ls -R -ltr /usr/ ---> run the program.

Confirm that “ls” command is showing up in ps listing

```
faculty@faculty-OptiPlex-3040:~$ ps
2579
```

PID	TTY	STAT	TIME	COMMAND
2579	pts/22	D	0:24	ls -R -ltr /usr/

Master-Slave implementation

- Master accepts n values from command line
- Set of 2 values to be passed to each child process i.e. argv[1] and argv[2] pass to child 1, argv[2] and argv[3] pass to child 2 and so on
- forks as many child processes as required i.e. $n/2$
- Reads the value returned using exit() call using WEXISTSTATUS macro
- [ProcessManagement\process_parent.c](#)
- Child process accepts the 2 command line arguments as argv[1] and argv[2]
- Perform the required processing
- Return the result back as exit status using exit() system call
- [ProcessManagement\process_child.c](#)

System Call : system()

```
int system(const char *command);
```

- Implemented using fork(), exec() and waitpid()
- Used to execute the command passed as parameter

e.g. system("ls -ltr"); → runs “ls –ltr” command

system("date > tempfile"); → create tempfile with output redirected from date

How to get additional information about process which is running

- When a process starts it creates a directory with process ID under /proc for per process information

First check the pid using ps command

```
faculty@faculty-OptiPlex-3040:/proc$ ps -aux
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
------	-----	------	------	-----	-----	-----	------	-------	------	---------

...

faculty	29593	0.7	3.9	2194648	156540	?	S	Jan17	20:48	/usr/lib/firefox/firefox -contentproc -chil
---------	-------	-----	-----	---------	--------	---	---	-------	-------	---

....

Process ID for this process is 29593

How to get additional information about process which is running – Collect System Information

Now change directory to /proc/29593 which is the directory for this process. List the content of the that directory

```
faculty@faculty-OptiPlex-3040:/proc/29593$ ls -l
total 0
dr-xr-xr-x 2 faculty faculty 0 Jan 19 11:17 attr
-rw-r--r-- 1 faculty faculty 0 Jan 19 11:17 autogroup
-r----- 1 faculty faculty 0 Jan 19 11:17 auxv
-r--r--r-- 1 faculty faculty 0 Jan 19 11:17 cgroup
--w----- 1 faculty faculty 0 Jan 19 11:17 clear_refs
-r--r--r-- 1 faculty faculty 0 Jan 18 17:48 cmdline
-rw-r--r-- 1 faculty faculty 0 Jan 19 11:17 comm
-rw-r--r-- 1 faculty faculty 0 Jan 19 11:17 coredump_filter
-r--r--r-- 1 faculty faculty 0 Jan 19 11:17 cpuset
```

```
lrwxrwxrwx 1 faculty faculty 0 Jan 19 02:46 cwd -> /home/faculty
-r----- 1 faculty faculty 0 Jan 19 11:08 environ
lrwxrwxrwx 1 faculty faculty 0 Jan 19 02:44 exe -> /usr/lib/firefox/firefox
dr-x----- 2 faculty faculty 0 Jan 19 02:46 fd
dr-x----- 2 faculty faculty 0 Jan 19 11:17 fdinfo
-rw-r--r-- 1 faculty faculty 0 Jan 19 11:17 gid_map
-r----- 1 faculty faculty 0 Jan 19 11:17 io
-r--r--r-- 1 faculty faculty 0 Jan 19 11:17 limits
-rw-r--r-- 1 faculty faculty 0 Jan 19 11:17 loginuid
dr-x----- 2 faculty faculty 0 Jan 19 11:17 map_files
-r--r--r-- 1 faculty faculty 0 Jan 19 02:46 maps
-rw----- 1 faculty faculty 0 Jan 19 11:17 mem
```

How to get additional information about process which is running – Collect System Information

```
-r--r--r-- 1 faculty faculty 0 Jan 19 11:17 mountinfo  
-r--r--r-- 1 faculty faculty 0 Jan 19 11:17 mounts  
-r----- 1 faculty faculty 0 Jan 19 11:17 mountstats  
dr-xr-xr-x 6 faculty faculty 0 Jan 19 11:17 net  
dr-x--x--x 2 faculty faculty 0 Jan 19 11:17 ns  
-r--r--r-- 1 faculty faculty 0 Jan 19 11:17 numa_maps  
-rw-r--r-- 1 faculty faculty 0 Jan 19 11:17 oom_adj  
-r--r--r-- 1 faculty faculty 0 Jan 19 11:17 oom_score  
-rw-r--r-- 1 faculty faculty 0 Jan 19 11:17 oom_score_adj  
-r----- 1 faculty faculty 0 Jan 19 11:17 pagemap  
-r----- 1 faculty faculty 0 Jan 19 11:17 personality  
-rw-r--r-- 1 faculty faculty 0 Jan 19 11:17 projid_map  
lrwxrwxrwx 1 faculty faculty 0 Jan 19 02:46 root -> /  
-rw-r--r-- 1 faculty faculty 0 Jan 19 11:17 sched
```

```
-r--r--r-- 1 faculty faculty 0 Jan 19 11:17 schedstat  
-r--r--r-- 1 faculty faculty 0 Jan 19 11:17 sessionid  
-rw-r--r-- 1 faculty faculty 0 Jan 19 11:17 setgroups  
-r--r--r-- 1 faculty faculty 0 Jan 19 11:17 smaps  
-r----- 1 faculty faculty 0 Jan 19 11:17 stack  
-r--r--r-- 1 faculty faculty 0 Jan 18 17:48 stat  
-r--r--r-- 1 faculty faculty 0 Jan 19 11:17 statm  
-r--r--r-- 1 faculty faculty 0 Jan 18 17:48 status  
-r----- 1 faculty faculty 0 Jan 19 11:17 syscall  
dr-xr-xr-x 33 faculty faculty 0 Jan 19 11:17 task  
-r--r--r-- 1 faculty faculty 0 Jan 19 11:17 timers  
-rw-r--r-- 1 faculty faculty 0 Jan 19 11:17 uid_map  
-r--r--r-- 1 faculty faculty 0 Jan 19 11:17 wchan  
faculty@faculty-OptiPlex-3040:/proc/29593$
```

How to get additional information about process which is running – Collect System Information

- Most files with its content is described here

File	Content
clear_refs	Clears page referenced bits shown in smaps output
cmdline	Command line arguments
cpu	Current and last cpu in which it was executed
cwd	Link to the current working directory
environ	Values of environment variables
exe	Link to the executable of this process
fd	Directory, which contains all file descriptors
maps	Memory maps to executables and library files
mem	Memory held by this process

How to get additional information about process which is running – Collect System Information

File	Content
root	Link to the root directory of this process
stat	Process status
statm	Process memory status information
status	Process status in human readable form
wchan	Present with CONFIG_KALLSYMS=y: it shows the kernel function symbol the task is blocked in - or "0" if not blocked
Pagemap	Page table
stack	Report full stack trace, enable via CONFIG_STACKTRACE
Smaps	an extension based on maps, showing the memory consumption of each mapping and flags associated with it
numa_maps	an extension based on maps, showing the memory locality and binding policy as well as mem usage (in pages) of each mapping

How to get additional information about process which is running – Collect System Information

File	Content
oom_adj & oom_score_adj	Adjust the oom-killer (Out-Of_Memory) score
oom_score	Display current oom-killer score
io	Display the IO accounting fields
coredump_filter	Core dump filtering settings
mountinfo	Information about mounts
comm & /task/<tid>/comm	common content info about the parent task and each of the child task e.g. on web browser there may be multiple windows each of which is child task
/task/<tid>/children	Information about task children. For each process there may be multiple child task, hence for each of the child task a subdirectory with process ID of child i.e. <tid> is created user /proc/<pid>/task/<tid>

How to get additional information about process which is running – Collect System Information

File	Content
/fdinfo/<fd>	Information about opened file (fd is file descriptor)
map_files	Information about memory mapped files
timerslack_ns	Task timerslack value
patch_state	Livepatch patch operation state

- How each file is structure with field names and use can be found at:
<https://www.kernel.org/doc/Documentation/filesystems/proc.txt>

Environment Variables in C

```
$ env  
SESSION=ubuntu  
TERM=xterm-256color  
SHELL=/bin/bash  
USER=faculty  
LD_LIBRARY_PATH=/home/faculty/torch/install/lib:  
PATH=/home/faculty/ApacheSpark/spark/bin:/home/faculty/tor  
ch/install/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/s  
bin:/bin:/usr/games:/usr/local/games:/snap/bin  
PWD=/home/faculty/SystemProgramming  
HOME=/home/faculty  
LANGUAGE=en_IN:en  
LOGNAME=faculty  
OLDPWD=/home/faculty  
_=~/bin/env  
$
```

Set User Defined Environment Variable:

```
$ course=it628  
$ export course  
$ env | grep course  
course=it628  
$
```

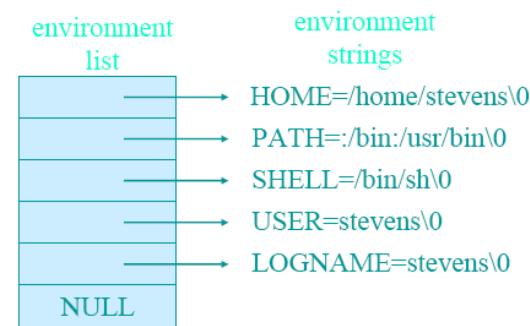
Environment Variables in C

ProcessManagement\disp env vars.c

■ Environment Variables

- int main(int argc, char **argv, char **envp);

extern char **environ;



■ getenv/putenv

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

```
int main(int argc, char*argv[],  
char*env[])
```

```
{
```

```
    char **ptr;
```

```
    for(ptr=env; *ptr != 0; ptr++)  
        printf("%s\n",*ptr);
```

```
}
```

Environment Variables in C

ProcessManagement\getenv setenv example.c

```
char *getenv(const char *name);
```

name – search for this name in environment of current process

Returns pointer to value

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

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

```
main()
```

```
{
```

```
printf("SHELL = %s\n", getenv("SHELL"));
```

```
printf("HOST = %s\n", getenv("HOST"));
```

```
}
```

```
int setenv(const char *name, const char *value, int overwrite);
```

Adds or change name parameter and its value

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

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

```
main()
```

```
{
```

```
setenv("COURSE", "Systems Programming", 1);
```

```
printf(" COURSE = %s\n", getenv(" COURSE "));
```

```
}
```

Systems
Software/Programming
Inter-Process
Communication

Inter-Process Communication

- Inter-Process Communication(IPC) is the generic term describing how two processes may exchange information with each other.
- In general, the two processes may be running on the same machine or on different machines
- This communication may be an exchange of data for which two or more processes are cooperatively processing the data or synchronization information to help two independent, but related, processes schedule work so that they do not destructively overlap.

Inter-Process Communication Methods

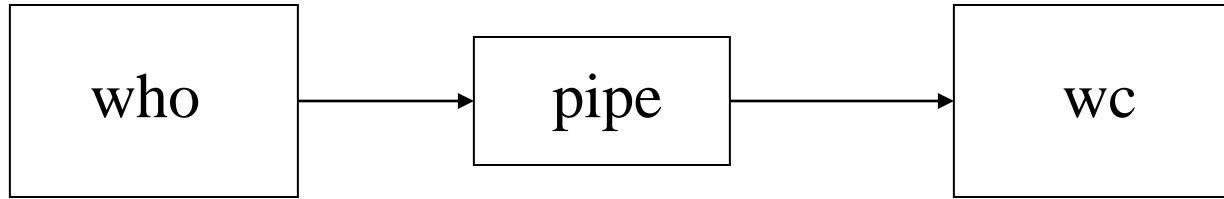
- Local → Processes running on the same machine
 - Pipe
 - Signal
 - MultiProcessing (MP) in multi-core/multi-processor architecture
- Distributed → Processes running on different machines
 - Message Passing Interface (MPI)

Inter-process Communication using Pipe

Pipes

- Pipes are an inter-process communication mechanism that allow two or more processes to send information to each other.
 - commonly used from within shells to connect the standard output of one utility to the standard input of another.
 - For example, here's a simple shell command that determines how many users there are on the system:
`$ who | wc -l`
 - The who utility generates one line of output per user. This output is then “piped” into the wc utility, which, when invoked with the “-l” option, outputs the total number of lines in its input.

Pipes



Bytes from “who” flow through the pipe to “wc”

A simple pipe

PIPs are more powerful constructs

- Child process exit code send to parent process → limit of only 1 int
- What if child process wants to send larger data of different data types to parent process?
- Also what if parent process wants to send some information to different child processes?
 - if information is static it can be initialized in parent before fork() to share a content of a variable with the respective child.
 - But what if parent generates information along the way after fork()?

Pipes

- It's important to realize that both the writer process and the reader process of a pipeline execute concurrently;
 - a pipe automatically buffers the output of the writer and suspends the writer if the pipe gets too full.
 - Similarly, if a pipe empties, the reader is suspended until some more output becomes available.
- All versions of UNIX support unnamed pipes, which are the kind of pipes that shells use.
- System V also supports a more powerful kind of pipe called a named pipe.

Unnamed Pipes: `pipe()` System Call

- An unnamed pipe is a unidirectional communications link that automatically buffers its input (the maximum size of the input varies with different versions of UNIX, but is approximately 5K) and may be created using the `pipe()` system call.
- Each end of a pipe has an associated file descriptor:
 - The “write” end of the pipe may be written to using `write()`
 - The “read” end may be read from using `read()`
- When a process has finished with a pipe’s file descriptor. it should close it using `close()`
- Note `read()`, `write` and `close()` are unbuffered I/O System Calls that we have studied earlier

Unnamed Pipes: pipe() System Call

```
int pipe( int fd[2] )
```

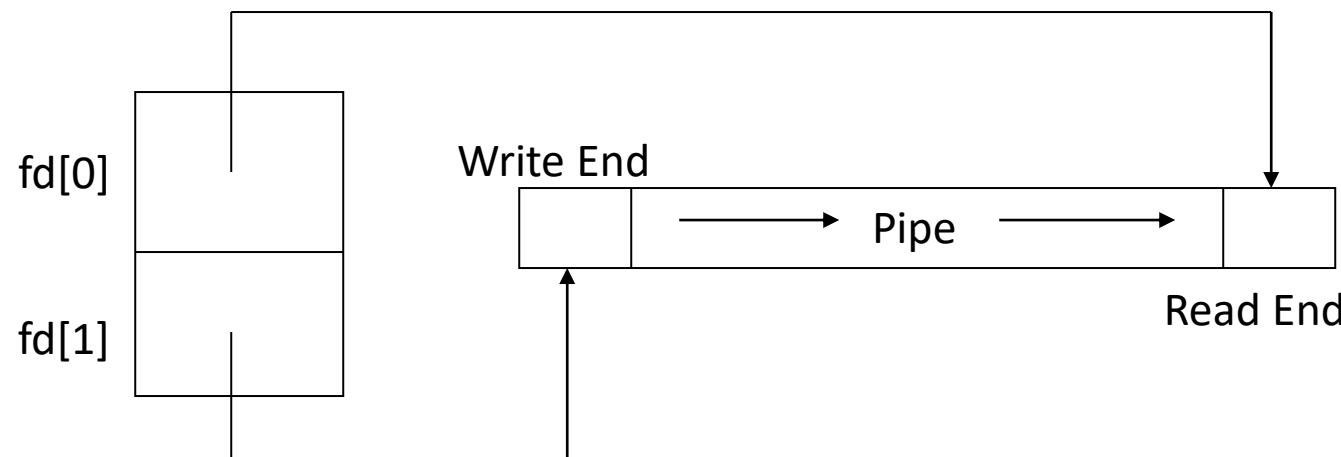
- pipe() creates an unnamed pipe and returns two file descriptors:
 - The descriptor associated with the “read” end of the pipe is stored in fd[0],
 - The descriptor associated with the “write” end of the pipe is stored in fd[1].

PIEs are generally Unidirectional

- Assume that the following code was executed:

```
int fd[2];  
pipe(fd);
```

data structure as shown below will be created



READ and WRITE conditions for PIPE for OS Implementation

- READ Conditions:

- If a process reads from a pipe whose “write” end has been closed, the “read()” call returns a value of zero, indicating the end of input.
- If a process reads from an empty pipe whose “write” end is still open, it sleeps until some input becomes available.
- If a process tries to read more bytes from a pipe that are present, all of the current contents are returned and read() returns the number of bytes actually read.

over reading

✓ 0

1 ✗

✓ 0

2 ✓

READ and WRITE conditions for PIPE for OS Implementation

- WRITE Conditions:

X 0 1 ✓

- if a process writes to a pipe whose “read” end has been closed, the write fails and the writer process is sent a SIGPIPE signal. the default action of this signal is to terminate the process.
- If a process writes fewer bytes to a pipe than the pipe can hold, the write() is guaranteed to be atomic; that is, the writer process will complete its system call without being preempted by another process.
- If the kernel cannot allocate enough space for a new pipe, pipe() returns a value of -1; otherwise, it returns a value of 0.

How OS would implement READ and WRITE for PIPE?

Variable to be used in read and write algos.

```
READ_END=0      // pipefd read index i.e. pipefd[READ_END]
WRITE_END=1     // pipefd write index i.e. pipefd[WRITE_END]
read_offset=0    // offset for the next read from pipe
write_offset=0   // offset for the next write to pipe
pipebufsize=64*1024 // maximum size of pipebuf
String pipebuf  // pipe wrap-around buffer to store pipe data
pipefilled      // number (count) of bytes already in pipe
```

How OS would implement READ for PIPE?

```
int READ(int pipefd[2], String appbuf, int appbufsz):  
    IF pipefd[WRITE_END] IS CLOSED THEN  
        return 0  
    ELSE  
        IF pipefilled IS 0 THEN READER PROCESS SLEEPS  
            FOR DATA TO WRITTEN TO PIPE (BLOCKING READ)  
        IF pipefilled <= appbufsize THEN  
            COPY ALL BYTES FROM pipebuf TO appbuf  
            pipefilled_ret = pipefilled  
            pipefilled = 0  
            read_offset = (read_offset + appbufsize) %  
            pipebufsize;  
        return pipefilled_ret  
    ELSE // pipefilled > appbufsize  
        COPY appbufsize BYTES FROM pipebuf TO appbuf  
        pipefilled_ret = appbufsize  
        pipefilled = pipefilled - appbufsize  
        read_offset = (read_offset + appbufsize) %  
        pipebufsize  
    return pipefilled_ret
```

How OS would implement WRITE for PIPE?

```
int WRITE(int pipefd[2], String appbuf, int appbufsz):
    IF pipefd[READ_END] IS CLOSED THEN
        SEND SIGPIPE SIGNAL TO WRITER PROCESS
    ELSE
        IF pipebufsize == pipefilled return 0
        IF (pipebufsize - pipefilled) >= appbufsize THEN
            COPY ALL BYTES FROM appbuf TO pipebuf
            pipefilled = pipefilled + appbufsize
            write_offset = (write_offset + appbufsize) %
                           pipebufsize
            pipefilled_ret = appbufsize
        ELSE // (pipebufsize - pipefilled < appbufsize)
            COPY (pipebufsize - pipefilled) BYTES FROM
                appbuf TO pipebuf
            pipefilled_ret = pipebufsize - pipefilled
            pipefilled = pipebufsize
            read_offset = (read_offset + appbufsize) %
                           pipebufsize
        return pipefilled_ret
```

Application Flow to Use PIPE(s)

- Unnamed pipes are usually used for communication between a parent process and its child, with one process writing and the other process reading.
- The typical sequence of events for such a communication is as follows:
 1. The main process creates an unnamed pipe using `pipe()`.
 2. The main process `forks` creating parent and child processes (sharing `pipfd`).
 3. The writer (parent/child) closes its “read” end of the pipe, and the designated reader (child/parent) closes its “write” end of the pipe.
 4. The processes communicate by using `write()` and `read()` system calls.
 5. Each process closes its active pipe descriptor when it is finished using them.

Example of Unidirectional Communication

- Here's a small program that uses a pipe to allow the child to send message to parent which is read by parent and display it:
- [ProcessManagement\talk_unidirectional.c](#)

Each message must have “End of Message” indication

- The child included the phrase’s **NULL terminator** as part of the message so that the parent could easily display it.

`write(fd[WRITE], phrase, strlen(phrase)+1); → length+1`

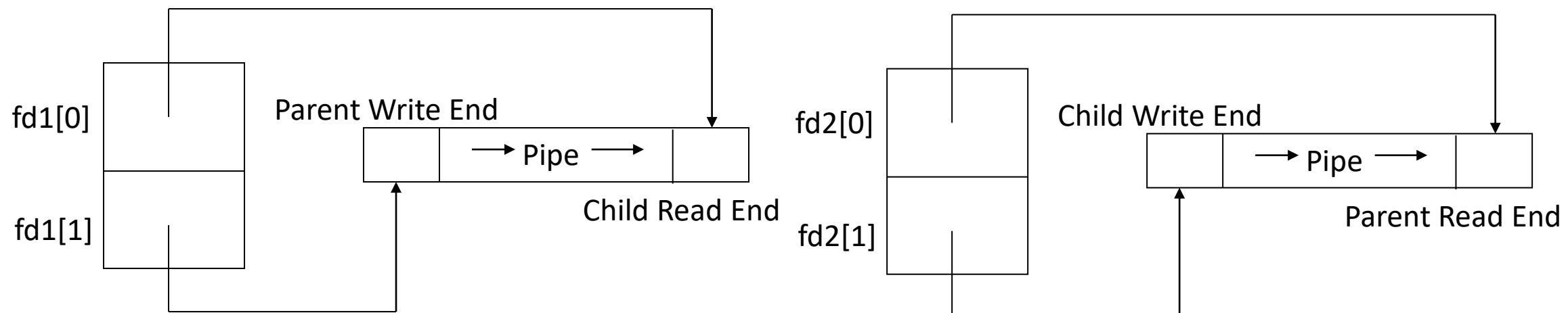
- When a writer process sends more than one variable-length message into a pipe, it must use a protocol to indicate to the reader the location for the end of the message.
- Methods for such indication include :
 - sending the length of a message(in bytes) before sending the message itself
 - ending a message with a special character such as a new line or a NULL

Bidirectional PIPE with 2 PIPEs

- Pipe are 1-way communication between parent and child that means for bidirectional (2-way) communication you need 2 pipes

```
int fd1[2], fd2[2];
```

```
pipe(fd1); pipe(fd2);
```



Example of Bidirectional Communication

- Please note : Bidirectional communication is only possible by using two pipes.
- [ProcessManagement\talk_bidirectional.c](#)

Our own implementation of Shell pipe operator (|)

- UNIX shells use unnamed pipes to build pipelines, connecting the standard output of the first to the standard input of the second.
- For example we can pass two arguments to [ProcessManagement\connect.c](#) first argument generating output on STDOUT and the other taking input which will be redirected from STDOUT.
- Code uses dup2 system call: duplicate a file descriptor and we can use the file descriptors interchangeably to refer to the file

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

Unnamed Pipes: pipe() System Call

\$ who ---> execute "who" by itself.

ubuntu18 :0 2019-02-13 10:11 (:0)

\$ who | wc ---> pipe "who" through "wc".

1 5 44 ...1 line, 5 words, 44 chars.

\$ connect who wc ---> pipe "who" through "wc".

1 5 44 ...1 line, 5 words, 44 chars.

Systems
Software/Programming
Inter-Process
Communication

Inter-Process Communications using Signals

What is a signal?

- Program must sometimes deal with unexpected or unpredictable events, such as :
 - a floating point error
 - a power failure
 - an alarm clock “ring”
 - the death of a child process
 - a termination request from a user (i.e., Control+C)
 - a suspend request from a user (i.e., Control+Z)
- These kind of events are sometimes called **interrupts**, as they must **interrupt the regular flow of a program** in order to be processed.
- When UNIX recognizes that such an event has occurred, it sends the corresponding process a signal.

Therefore a signal is:

- a software interrupt delivered to a process by the OS because:
 - it did something (segmentation fault, FPE)
 - the user did something (pressed Ctrl+C)
 - another process wants to tell it something (SIGUSR?)
- one way a process can communicate with other processes
- Signal Types:
 - Some signals are **asynchronous**, they may be raised at any time (user pressing **Ctrl+C**)
 - Some signals are directly related to hardware/process code (illegal instruction, arithmetic exception, such as attempt to divide by 0) - **synchronous** signals
- **Signals functions** are defined in **signal.h** and **signal numbers** are defined in **signal.h** or **signum.h** (**signum.h** is used inside **signal.h** if used)

Actions when Signal is received

- The default handler usually performs one of the following actions:
 - Abort: terminate the process and generate a core file (dump)
 - Quit/Exit: terminate the process without generating a core image file (quit)
 - Ignore: ignore and discard the signal (ignore)
 - Stop: suspend the process (suspend)
 - Continue: resume the process (resume)

List of Signals (Linux)

Macro	Signal Code	Default Action	Description
SIGHUP	1	Quit	Hang up - sent to a process when its controlling terminal has disconnected
SIGINT	2	Quit	Interrupt – Ctrl+C pressed by user, terminate the process after saving the work, Signal can be trapped
SIGQUIT	3	Dump	Quit
SIGILL	4	Dump	Illegal instruction (bad opcode)
SIGTRAP	5	Dump	Trace trap (used by debuggers)
SIGABRT	6	Dump	Abort process – Ctrl+\ pressed by user, terminate immediately
SIGBUS	7	Dump	bus error (bad format address or unaligned memory access)
SIGFPE	8	Dump	Floating Point (Arithmetic) execution bad argument
SIGKILL	9	Quit	shell command Kill –SIGKILL or kill -9 or system call kill() to send SIGKILL signal by another process (unblockable)

List of Signals (BSD)

Macro	Signal Code	Default Action	Description
SIGUSR1	10	Quit	user signal 1
SIGSEGV	11	Dump	segmentation violation (out-of-range address)
SIGUSR2	12	Quit	user signal 2
SIGPIPE	13	Quit	write on a pipe or other socket with no one to read it.
SIGALRM	14	Quit	alarm clock timeout
SIGTERM	15	Quit	software termination signal(default signal sent by kill)
SIGSTKFLT	16	Quit	Stack fault
SIGCHLD	17	Ignore	child status changed
SIGCONT	18	Ignore	continued after suspension
SIGSTOP	19	Quit	Suspend process by signal (unblockable)
SIGTSTP	20	Quit	Keyboard store, Stopped by user (Ctrl+Z pressed which suspend/pause the process)

List of Signals (BSD)

Macro	Signal Code	Default Action	Description
SIGTTIN	21	Quit	Background read from tty
SIGTTOU	22	Quit	Background write from tty output
SIGURG	23	Ignore	Urgent condition on socket
SIGXCPU	24	Dump	CPU time limit exceeded
SIGXFSZ	25	Dump	file size limit exceeded
SIGVTALRM	26	Quit	virtual timer expired
SIGPROF	27	Quit	profiling timer expired
SIGWINCH	28	Ignore	window size change
SIGIO	29	Ignore	I/O is now possible
SIGINFO	29	Ignore	Status request from keyboard
SIGPWR	30	Quit	Power failure restart
SIGSYS	31	Dump	bad argument to system call or non-existence system call

To run any process in background: ./process.out &

Sending signals from keyboard

`fg` brings last background process in foreground

- Process suspended using `Ctrl+Z` (`SIGTSTP`) can be brought back to life using `fg` command which sends (`SIGCONT`) signal to resume
- For background process can't use `Ctrl+C`, `Ctrl+Z` etc hence `kill` command is used (how one can run process in background ?)

`kill [options] pid`

to kill any background process: `kill -9/SIGKILL pid`

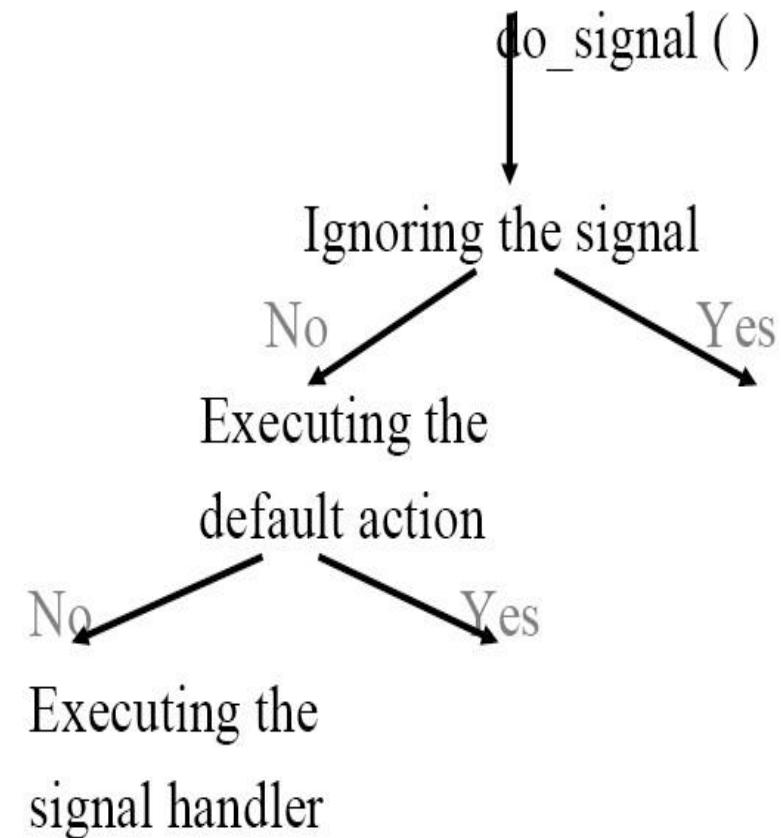
`--l` lists all signals you can send

`--signal number`

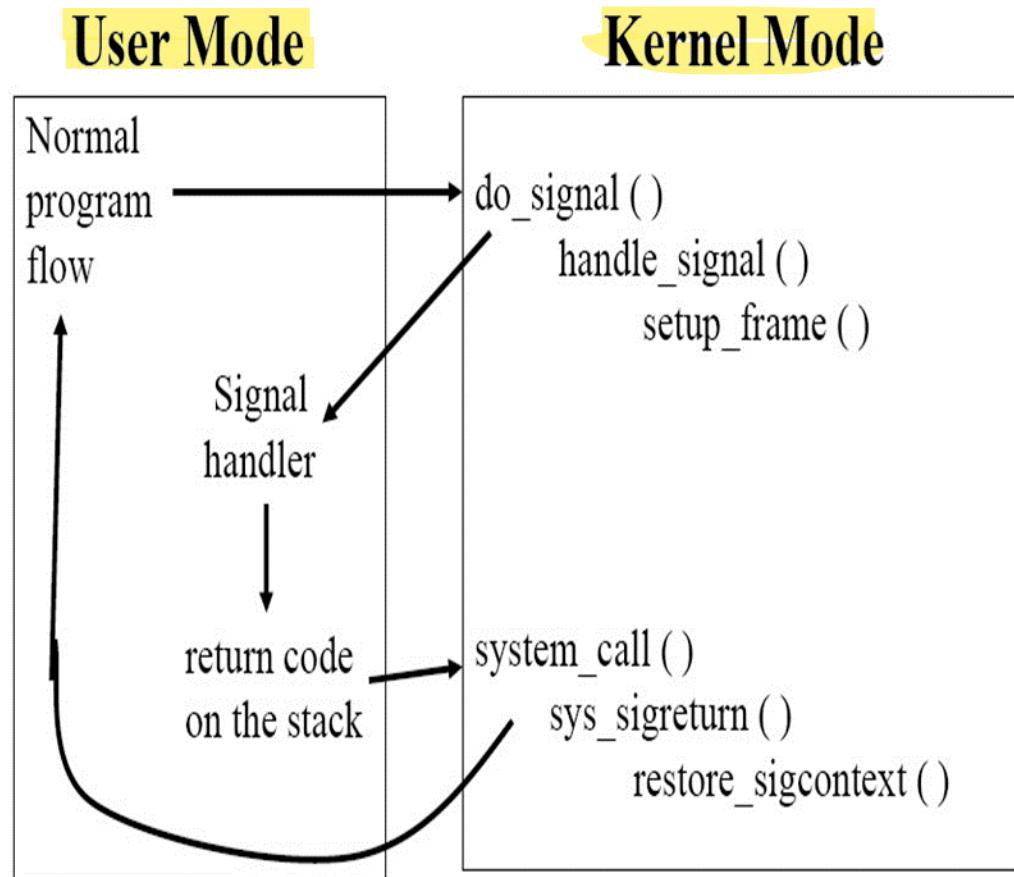
kill command can send `SIGKILL` signal upon pressing `Ctrl+C` to the foreground process which cannot be ignored

Signal Handling

- Most signals can be ignored except **SIGKILL** and **SIGSTOP**
- Application can trap the signal (e.g. **SIGINT** when Ctrl+C pressed) and handle signal disposition via signal handler
- Signal handler can perform default action (usually process termination) or customized actions such as blocking the signal



Catching Signal



Requesting an Alarm Signal SIGALRM using alarm() System Call

System Call : unsigned int `alarm(unsigned int count)`

- `alarm()` instructs the kernel to send the `SIGALRM` signal to the calling process after `counting seconds.`
 - If an alarm had already been scheduled, than an alarm is overwritten.
 - If `count` is 0, any pending alarm requests are cancelled.
- The `default handler` for this signal **displays the message “Alarm clock”** and terminates the process.
- `alarm()` returns the number of seconds that remain until the alarm signal is sent

System Call alarm() example

- Set Alarm for 3 second with default action handler which will display a default message “**Alarm clock**” and exit the program

[Signals\alarm_test.c](#)

Handling Signals (Overriding Default Action)

System Call: `void(*signal(int sigCode, void (*func)(int))) (int)`

- `signal()` allows a process to specify the action that it will take when a particular signal is received.
 - `sigCode` specifies the signal number (as per the table shown in earlier slides) that is to be reprogrammed
 - `func` may be one of several values:
 - `SIG_IGN`, which indicates that the specified signal should be ignored and discarded (i.e. No Action to be taken)
 - `SIG_DFL`, which indicates that the kernel's default handler should be used.
 - an address of a user-defined function, which indicates that the function should be executed when the specified signal arrives.

Handling Signals

- The valid signal numbers are stored in `/usr/include/signal.h` or `/usr/include/bits/signum.h`
 - The signals `SIGKILL` and `SIGSTP` may not be reprogrammed.
- `signal()` returns the previous func value associated with `sigCode` if successful; otherwise, it returns a value of -1
- `signal()` system call can be used to override the default action
- A child process inherits the signal settings from its parent during a `fork()`

System Call pause()

System Call: int pause(void)

- pause() suspends the calling process and returns when a calling process receives a signal.
- It is most often used to wait efficiently for a signal.
- pause() doesn't return anything useful.

Difference between sleep(int) and pause:

pause() suspends process for indefinite time whereas sleep(int secs) suspends processes for specified number of seconds passed as parameter secs.

Catch **SIGALRM** of alarm() system call using signal() system call

- Write your own handler to handle alarm signal SIGALRM
- Override default action using signal() system call
- [Signals\alarm_override.c](#)

Protecting Critical Code from Ctrl-C attack

- Overriding may be used to protect critical pieces of code against Control-C attacks and other such signals.
- it can be restored after the critical code has executed.
- Here's the source code of a program that protects itself against SIGINT signals [Signals\ctrl_c_override.c](#)

Process Groups

- When you execute a program in terminal window (shell) that creates several children, a single Control-C from the keyboard will normally terminate the foreground parent process and all its children processes and then return to the shell.
- In order to support this kind of behavior, UNIX introduced a few new concepts.
 - In addition to having a unique process ID number, every process is also a member of a process group. Each process has process id (pid) and process group id (pgid).
 - Several processes can be members of the same process group (pgid).
 - When a process forks, the child inherits its process group from its parent.
 - A process may change its process group to a new value by using setpgid()
 - When a process runs exec???() system call, its process group remains the same.

Process Groups and Control Terminals (i.e. Which processes have control of the terminal windows)

- Every process can have an associated control terminal, which is typically the terminal where the process was started.
 - When a process forks, the child inherits its control terminal from its parent.
 - When a process execs, its control terminal stays the same.
- Every terminal can be associated with a single control process.
 - When a metacharacter such as a Control-C is detected, the terminal sends the appropriate signal to all of the processes in the process group of its control process.
- If a process attempts to read from its control terminal and is not a member of the same process group as the terminal's control process,
 - the process is sent a SIGTTIN signal, which normally suspends the process.

Process Groups and Control Terminals (i.e. Which processes have control of the terminal windows)

- When an interactive shell begins (i.e. terminal window), it is the control process of a terminal and has that terminal as its control terminal.
- When a shell executes a foreground process:
 - the child shell places itself in a different process group before exec'ing the command and takes control of the terminal by executing “int tcsetpgrp(int fd, pid_t pgrp)” system call
 - Any signals generated from the terminal thus go to the foreground command rather than to the original parent shell.
 - When the foreground command terminates, the original parent shell takes back control of the terminal again using tcsetpgrp system call

terminal control set process group

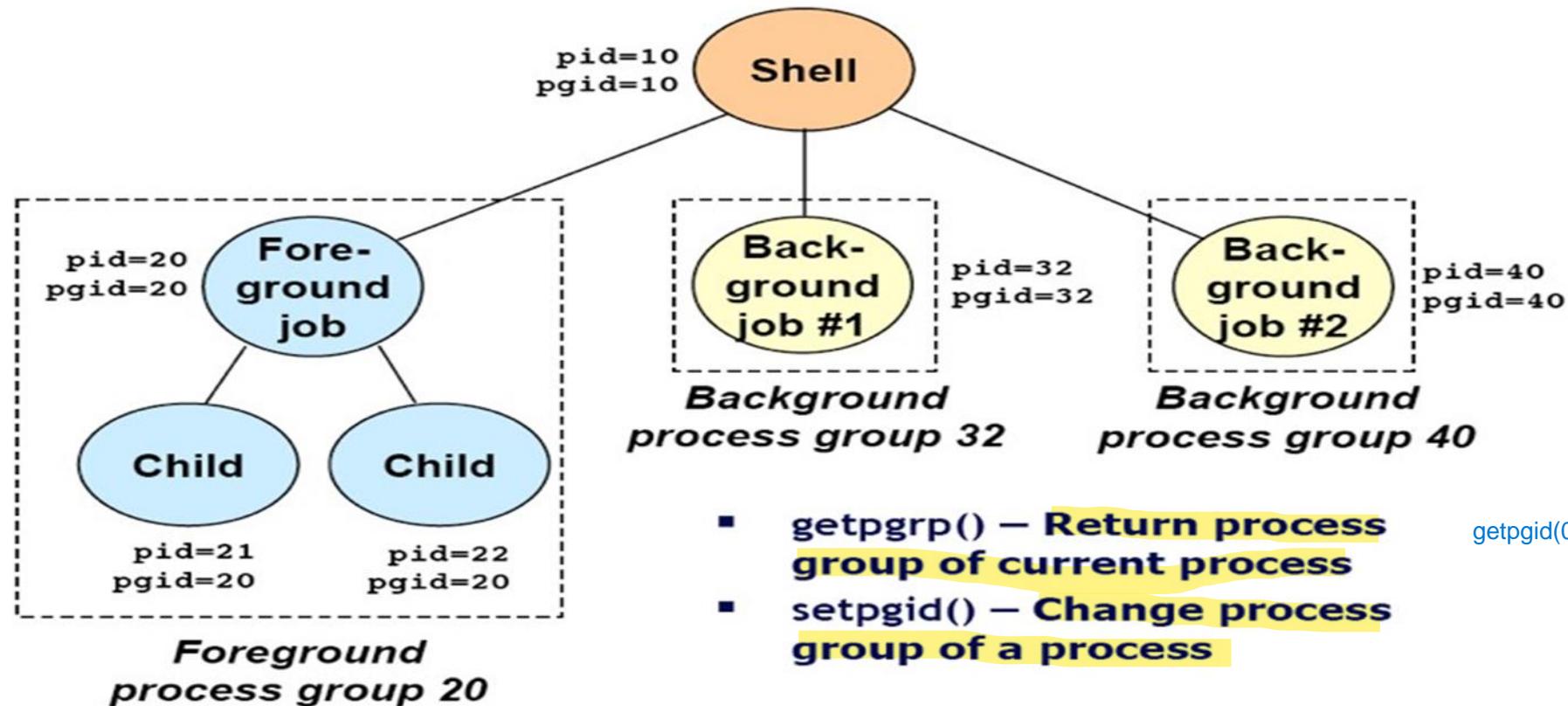
Process Groups and Control Terminals

Signals\process group background.c

- When a shell executes a background process:
 - the child shell places itself in a different process group before executing, but does not take control of the terminal.
 - Any signals generated from the terminal continue to go to the shell.
 - If the background process tries to read from its control terminal, it is suspended by a SIGTTIN signal.

Process Groups and Control Terminals

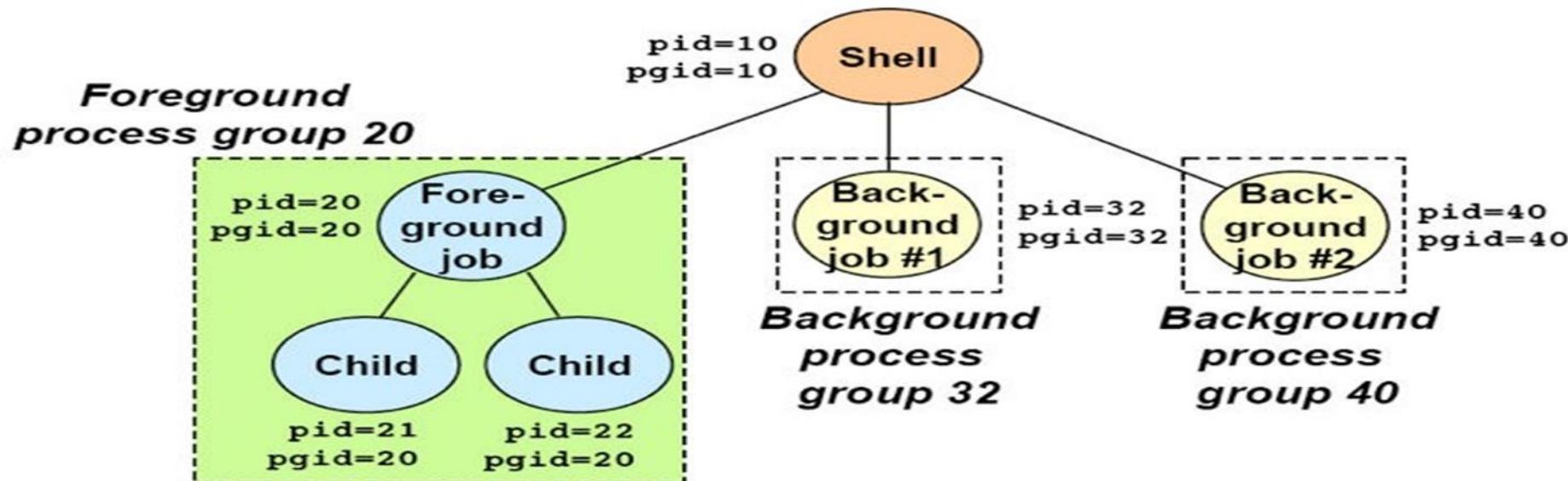
- Every process belongs to exactly one process group.



Process Groups and Control Terminals

- **Sending signals from the keyboard**

- Typing **ctrl-c** (**ctrl-z**) sends a **SIGINT** (**SIGTSTP**) to every job in the foreground process group.
 - **SIGINT**: default action is to terminate each process.
 - **SIGTSTP**: default action is to stop (suspend) each process.



group Id is nothing but ppid

System Call setpgid() and getpgid()

setpgid(getpid(),0) : set getpid() as the group id of current process

System Call : pid_t setpgid(pid_t pid, pid_t pgrpId)

- setpgid() sets the process-group ID of the process with PID pid to pgrpId.
 - if pid is zero, the caller's process group ID is set to pgrpId.
 - In order for setpgid() to succeed and set the process group ID, at least one of the following conditions must be met:
 - The caller and the specified process must have the same owner.
 - The caller must be owned by a super-user.
- If setpgid() fails, it returns a value of -1.

System Call: pid_t getpgid(pid_t pid)

- getpgid()" returns the process group ID of the process with PID pid.
 - If pid is zero, the process group ID of the caller is returned.

Parent and Child in Same Process Groups, both gets the Signal

- example illustrates the fact that a terminal distributes signals to all of the processes in its control process' process group.
- Since the child inherited its process group from its parent, both the parent and child caught the SIGINT signal.

[Signals\process_group_same.c](#)

```
$ ./process_group_same.out parent  
pid 19354 and parent group 19354  
waits
```

```
child pid 19355 and child group  
19354 waits
```

```
^C
```

```
process 19355 got a SIGINT  
process 19354 got a SIGINT
```

```
$
```

Parent and Child in Different Process Groups, Only one gets the Signal

- example illustrates the fact that a terminal distributes signals to all of the processes in its control process' process group.
- Since the child is in different process group SIGINT signal was caught only by parent and not child. We had to explicitly kill child since it was orphan.

[Signals\process_group_diff.c](#)

```
$ ./process_group_diff.out
parent pid 19460 and parent group 19460 waits
child pid 19461 and child group 19461 waits
^C
process 19460 got a SIGINT
$ ps 19461
PID TTY  STAT  TIME COMMAND
19461 pts/1  S  0:00 ./process_group_diff.out
$ kill 19461
$ ps 19461
PID TTY  STAT  TIME COMMAND
$
```

How parent can get additional info about child terminated or stopped by signal?

`childpid = wait(&wstatus);` → returns the exit status from child which can further be inspected using these macros

`WIFEXITED(wstatus)` → returns true if child terminated normally

`WEXITSTATUS(wstatus)` → returns exit status (least significant 8 bits)

`WIFSIGNALED(wstatus)` → returns true if child process was terminated by a signal

`WTERMSIG(wstatus)` → returns the number of signal

`WCOREDUMP(wstatus)` → returns true if child produced a core dump

`WIFSTOPPED(wstatus)` → returns true if child was stopped by a signal

`WSTOPSIG(wstatus)` → returns the signal number which caused child to stop

`WIFCONTINUED(wstatus)` → returns true if child was resumed with SIGCONT signal

Sending Signal from another Process: System Call kill()

- A process may send a signal to another process by using the kill() system call
- kill() is a misnomer, since many of the signals that it can send do not terminate a process
- Its call kill() because it was originally designed to terminate a process which is still one of its function

System Call kill()

System Call: `int kill(pid_t pid, int sigCode)`

- `kill()` sends the signal with value `sigCode` to the process with PID `pid`.
- `kill()` succeeds and the signal is sent as long as at least one of the following conditions is satisfied:
 - The sending process and the receiving process have the same owner.
 - The sending process is owned by a super-user.
- There are a few variations on the way that “`kill()`” works:
 - If `pid` is positive, then `signal` is sent to the process with the ID specified by `pid`
 - If the `pid` is negative, but not -1, then `signal` is sent to every process in the process group whose ID is `-pid`.
 - If `pid` is zero, then `signal` is sent to every process in the same process group as of the calling process.
 - If `pid` is -1 and the sender is owned by a super-user, the signal is sent to all processes, including the sender.
 - If `pid` is -1 and the sender is not owned by a super-user, the signal is sent to all of the processes owned by the same owner as that of the sender, excluding the sending process.

Detecting death of a Child Process

- When a child terminates, the OS sends parent of the terminated child a SIGCHLD signal.
- A parent process often installs a handler to deal with this signal; this handler typically executes a wait() system call to accept the child's termination code and let the childdezombify.
- the parent can choose to ignore SIGCHLD signals.
- The program works by performing the following steps:
 - The parent process installs a SIGCHLD handler that is executed when its child process terminates.
 - The parent process forks a child process to execute the command.
 - The parent process goes on and do other activities.
 - If the child terminates before its parent finishes, the parent's SIGCHLD handler is executed for that child.
 - Note: If parent has multiple children for each child SIGCHLD handler will be executed upon termination the child.

Master-Slave Implementation with SIGCHLD

- In earlier parent and child processes which implemented master-slave relationship, parent was not doing anything except waiting for child process, hence child was not zambified
- But if parent process is also performing some tasks and has to make sure that child is not zambified, it should override SIGCHLD signal and execute wait system call todezambify child
- [Signals\process parent sigchld.c](#)
- [Signals\process child sigchld.c](#)

Detecting death of a Child Process

Signals\child death detection.c

```
#include <stdio.h>
#include <signal.h>
int delay;
void childHandler();
int main( int argc, char* argv[] )
{
    int pid;
    signal( SIGCHLD, childHandler ); /* Install death-of-child
handler */
    pid = fork(); /* Duplicate */
    if ( pid == 0 ) /* Child */
    {
        execvp( argv[2], &argv[2] ); /* Execute command */
        perror("limit"); /* Should never execute */
    }
    else /* Parent */
    {
        sscanf( argv[1], "%d", &delay ); /* Read delay from
command-line */
        sleep(delay); /* Sleep for the specified number of seconds
*/
        printf("Child %d exceeded limit and is being killed \n", pid );
        kill( pid, SIGINT ); /* Kill the child */
    }
}
```

Detecting death of a Child Process

```
void childHandler() /* Executed if the child dies before the parent */
{
    int childPid, childStatus;
    childPid = wait(&childStatus); /* Accept child's termination code */
    printf("Child %d terminated within %d seconds \n", childPid, delay);
    exit(/* EXIT SUCCESS */ 0);
}
```

```
$ limit 5 ls  ---> run the program; command finishes OK.  
a.out      alarm      critical      handler      limit  
alarm.c    critical.c  handler.c   limit.c  
Child 4030 terminated within 5 seconds.
```

```
$ limit 4 sleep 100  ---> run it again; command takes too long  
Child 4032 exceeded limit and is being killed
```

Suspending and Resuming Process using SIGSTOP and SIGCONT signals

- The **SIGSTOP** and **SIGCONT** signals **suspend and resume** a process, respectively.
- They are used by the UNIX shells that support job control to implement built-in commands like stop, fg, and bg.
- Program in next slide performs these steps:
 - The main program creates two children; they both enter an infinite loop and display a message every second.
 - The main program waits for three seconds and then **suspend** the first child.
 - After another **three seconds**, the parent restarts the first child, waits a little while longer, and then terminated both children.

Suspending and Resuming Process using SIGSTOP and SIGCONT signals

Signals\sigstop sigcont example.c

```
#include <signal.h>
#include <stdio.h>
int main(void)
{ int pid1;
  int pid2;
  pid1 = fork();
  if (pid1== 0) /* First child */
  {
    while(1) /* Infinite loop */
    {
      printf("pid1 is alive \n");
      sleep(1);
    }
  }
  pid2 = fork(); /* Second child */
  if ( pid2 == 0 )
  {
    while(1) /* Infinite loop */
    {
      printf("pid2 is alive \n");
      sleep(1);
    }
    sleep(3);
    kill( pid1, SIGSTOP ); /* Suspend first child */
    sleep(3);
    kill( pid1, SIGCONT ); /* Resume first child */
    sleep(3);
    kill( pid1, SIGINT ); /* Kill first child */
    kill( pid2, SIGINT ); /* Kill second child */
  }
}
```

Suspending and Resuming Process using SIGSTOP and SIGCONT signals

- \$ pulse ---> run the program.
- pid1 is alive ---> both run in first three seconds.
- pid2 is alive
- pid1 is alive
- pid2 is alive
- pid1 is alive
- pid2 is alive
- pid2 is alive ---> just the second child runs now since first child is suspended
- pid2 is alive
- pid2 is alive
- pid1 is alive ---> the first child is resumed.
- pid2 is alive
- pid1 is alive
- pid2 is alive
- pid1 is alive
- pid2 is alive

SIGUSR1 and SIGUSR2 → User Defined Signals

- Developer can use SIGUSR1 and/or SIGUSR2 to create own signals
- Steps for implementing SIGUSR1/2:
 - Process designated as recipient of SIGUSR1/2:
 - First create a signal handler function → void my_signal_handler(int signum)
 - signum will be passed as SIGUSR1 or SIGUSR2 depending on which signal was sent
 - Register signal handler function for each signal so that it can be called upon receipt of that signal → signal(SIGUSR1, my_signal_handler)
 - Process designated for sending signals SIGUSR1/2:
 - Uses kill() system call to generate/send signal to a specific process using pid → kill(pid, SIGUSR1)
 - As we studied earlier kill() system call is misnomer and you can use it to send any signal to a process, in this case it is sending SIGUSR1 signal to pid1

Synchronize the common activity between all children

- At a particular time interval parent and child all need to do certain activity
- Parent can set alarm for a specified time and upon receipt of the SIGALRM it sends SIGUSR1 to the group
- [Signals\siguser sync example.c](#)

Synchronization between two processes

- Inter Process Communication between parent and child using SIGUSR1 and SIGUSR2
- This example shows
- Register signal handler for SIGUSR1 and SIGUSR2 for parent process which inherited by child process because they belong to the same process group
- Child first sends SIGUSR1 to parent
- Parent upon receiving SIGUSR1 sends SIGUSR1 to child
- Child upon receiving SIGUSR1 sends SIGUSR2 to parent
- Parent upon receiving SIGUSR2 sends SIGUSR2 to child
- Child upon receiving SIGUSR2 terminates itself by sending SIGINT signal to itself
- Parent when detect child has died it terminates
- [Signals\siguser_test.c](#)

Ignore Other Signals inside Signal Handler

Signals\ignore signals inside signalhandler.c

```
int main(int ac, char *av[])
{
    void inthandler(int);
    void quithandler(int);
    char input[100];
    signal( SIGINT, inthandler ); // set trap
    signal( SIGQUIT, quithandler ); // set trap
    int i=1;
    while (i<5) {
        sleep(1);
        printf("main:%d\n",i++);
    }
}

void inthandler(int s)
{
    printf(" Received SIGINT signal %d .. waiting\n", s );
    printf(" Leaving inthandler \n");
}

void quithandler(int s)
{
    signal(SIGINT, SIG_IGN);
    printf(" Received SIGQUIT signal %d .. waiting\n", s );
    sleep(5);
    printf(" Leaving quithandler \n");
    signal( SIGINT, inthandler );
}
```

System Call: sigaction()

Left from Here

System Call : int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)

- It is more advance system call then signal()
- It uses struct sigaction for additional controls

struct sigaction {

void (*sa_handler)(int); → SIG_DFL or SIG_IGN or pointer to signal handler

void(*sa_sigaction)(int, siginfo_t *, void *); → Used only of SA_SIGINFO value in sa_flags, passes additional information

sigset_t sa_mask; → set of additional signals to be blocked during execution of sa_handler function

int sa_flags; → special flags affecting signal behavior e.g. SA_SIGINFO value in this flag indicates handler is sa_sigaction rather than sa_handler

void (*sa_restore)(void); → used for POSIX real time signal only not used by application
}

Manipulating Signal Sets

sa_mask of sigaction structure has set of signal to block.

Set can be modified using these functions:

sigemptyset(sigset_t *set) → init to no signals

sigfillset(sigset_t *set) → init to all signals

sigaddset(sigset_t *set, int signum) → add signal to set

sigdelset(sigset_t *set, int signum) → remove signal from set

sigismember(sigset_t *set, int signum) → check if signal is member of set

System Call : sigprocmask()

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

oldset: oldset is set with currently block set of signals before making any changes (this will be used to restore the block state before the call to sigprocmask)

set : set of signals that will follow action of “how”

how:

SIG_BLOCK: add set of signals in “set” to current blocked signal set

SIG_UNBLOCK: remove set of signals in “set” from current blocked signal set

SIG_SETMASK: set of blocked signal is set to “set”

Block Other Signal inside Signal Handler using segprocmask()

```
void inthandler(int s)
{
sigset_t sigset, sigoldset;
sigemptyset(&sigset);
sigaddset(&sigset, SIGQUIT);
sigprocmask(SIG_SETMASK, &sigset,
&sigoldset);
printf(" Received signal %d .. waiting\n", s );
.....
printf(" Leaving SIGINT Handler \n");
sigprocmask(SIG_SETMASK, &sigoldset, NULL);
}
```

```
int main(int ac, char *av[])
{
void inthandler(int);
signal( SIGINT, inthandler ); // set trap

int i=1;
while (i<5) {
    sleep(1);
    printf("main:%d\n",i++);
}
}
```

sigaction() example

```
main()
{
    struct sigaction newhandler; /* new settings */
    sigset_t blocked; /* set of blocked sigs */
    void inthandler(); /* the handler */
    newhandler.sa_handler = inthandler; /* handler function */
    sigfillset(&blocked); /* mask all signals */
    newhandler.sa_mask = blocked; /* store blockmask */
    int i;
    for (i=1; i<31;i++)
        if (i!=9 && i!=17) /* catch all except these signals */
            if ( sigaction(i, &newhandler, NULL) == -1 )
                printf("error with signal %d\n", i);
    while(1){}
}
```

Systems
Software/Programming
Concurrent Programming

Motivation Concurrent Programming

- Improve the application performance by running multiple tasks simultaneously (Divide the task into n smaller pieces, using n processors run it n times faster) → mainly used in computationally intensive processes
 - addition of 100,000 elements available in memory
 - convert RGB image to grayscale by applying $0.2126R + 0.7152G + 0.0722B$ to each pixel.
- To cope with independent slower devices → mainly used in I/O bound processes e.g. running multiple database queries since database is mostly disk I/O intensive application
 - Do not wait for slower devices such as disk, network etc. instead perform other tasks
 - During I/O perform computation
 - During continuous visualization, handle key strokes and I/O e.g. video games
 - While listening to network, perform other operations e.g. listening to multiple sockets at the same time
 - Concurrent I/O, concurrent transfers e.g. Web browsers

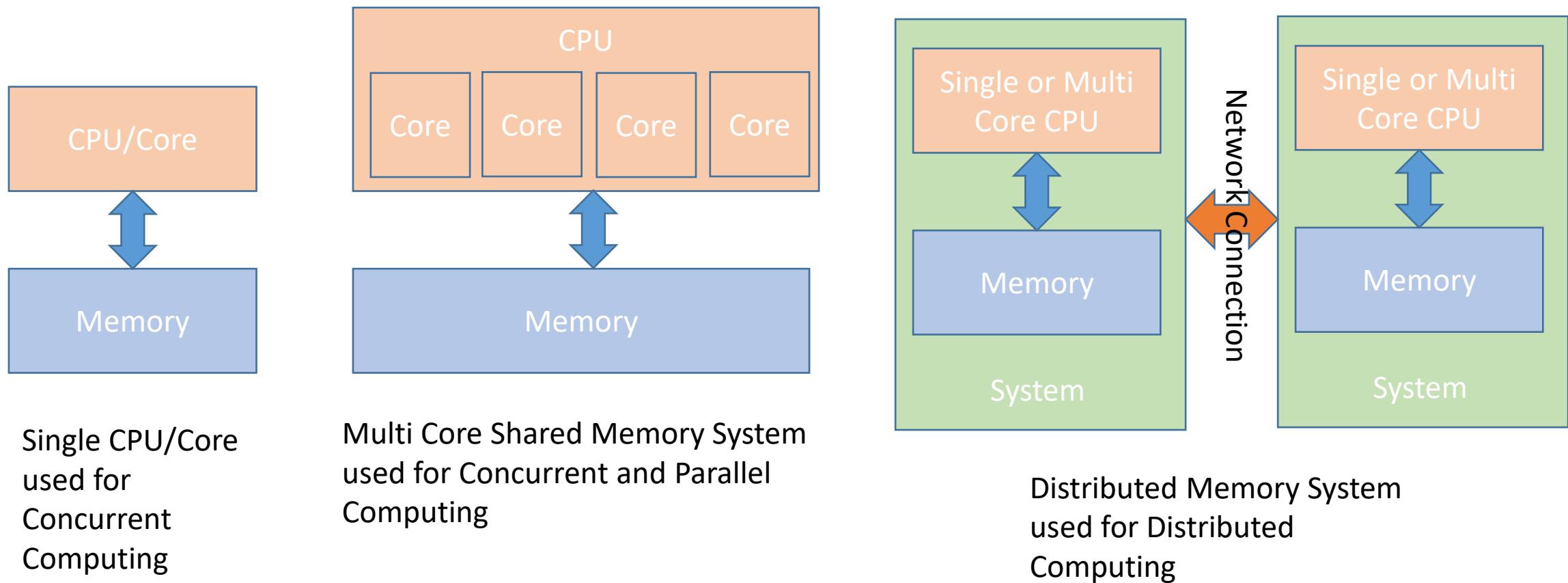
Sequential versus Concurrent Programming

- Sequential Programming : Multiple computational tasks are performed in sequential order
- Concurrent Programming : Multiple computational tasks are performed simultaneously
- Concurrent tasks can be implemented using
 - Multiple processes using parent-child model using fork() that we studied earlier OR
 - Multiple threads within the same process
- Execution of concurrent tasks can use:
 - Single processor (core) : multi-process or multithreaded programming where multiple processes or threads running concurrently
 - Several processors (cores) part of the same system i.e. in close proximity : parallel computing where multiple threads run in parallel
 - Several processors (cores) distributed across a network i.e. part of different systems : distributed computing

Interaction between Concurrent Tasks

- multi-process or multithreaded programming and parallel computing is meant for single systems → Shared memory architecture
 - All tasks has access to the same physical memory
 - Communication between tasks is done by changing content of the shared memory
- Distributed computing is meant for multiple systems connected through network → Distributed memory architecture
 - No sharing of physical memory
 - Tasks must communicate by exchanging messages

Shared Memory vs Distributed Memory Architectures



Serial versus Parallel Tasks (Divide and Conquer)

$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8$

Step1: sum=x₁ + x₂

Step2: sum=sum + x₃

Step3: sum=sum + x₄

Step4: sum=sum + x₅

Step5: sum=sum + x₆

Step6: sum=sum + x₇

Step7: sum=sum + x₈

How many total steps? 7 ~ 8

$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8$

With 4 processors divide tasks in 4

Step1: parallelism = 4

a=x₁ + x₂

b=x₃ + x₄

c=x₅ + x₆

d=x₇ + x₈

Now using 2 processors calculate

Step2: parallelism = 2

e=a+b

f=c+d

Now using 1 processor calculate

Step3: parallelism = 1

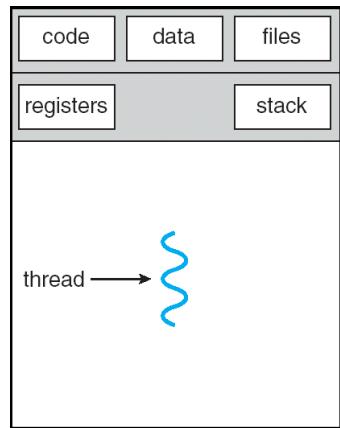
sum=e+f

How many total steps? 3 = log(8)

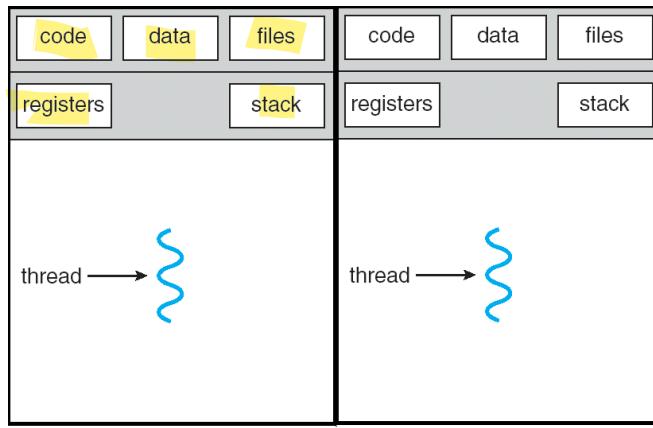
Gain from Parallelism

- In theory,
 - Dividing program into p small parts and running them in parallel on p processors should result in p times speed up (means it should run p times faster), because
 - Time taken by parallel version algorithm (T_p) = Time taken by serial algorithms (T_s) / Number of processors (p)
 - Speedup (s) = $T_s / T_p \rightarrow$ this should be equal to number of processors (p)
- In practice, this is the case due to
 - Inter-task dependencies
 - In addition example $T_s=n$, $T_p=\log(n)$, therefore $\text{speedup}(s)=n/\log(n)$, for $n=8$; $T_s=8$, $T_p=\log(8)=3$ therefore $\text{speedup}(s)=8/3=2.67$. Ideally it should have been equal to processors $p=4$.
 - Communication Costs
 - In case data is very large which need to send across network, it will incur communication cost e.g. reading 100,000 elements from use in one main process and then sending partial results of additions to other processors

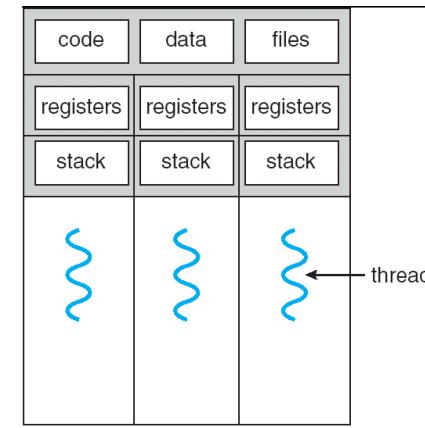
Memory Map of Multi-process vs Multithreading



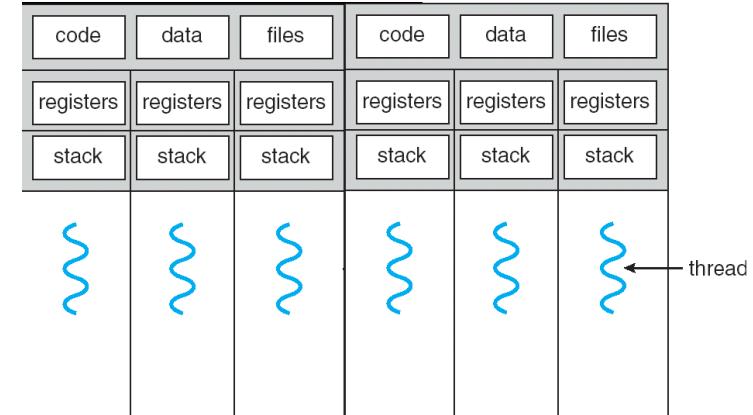
Single Process
Single Thread



Multi-Process Single Thread



Single Process
Multithreaded



Multi-Process Multithreaded

Multi-process Model vs Multithreaded Model

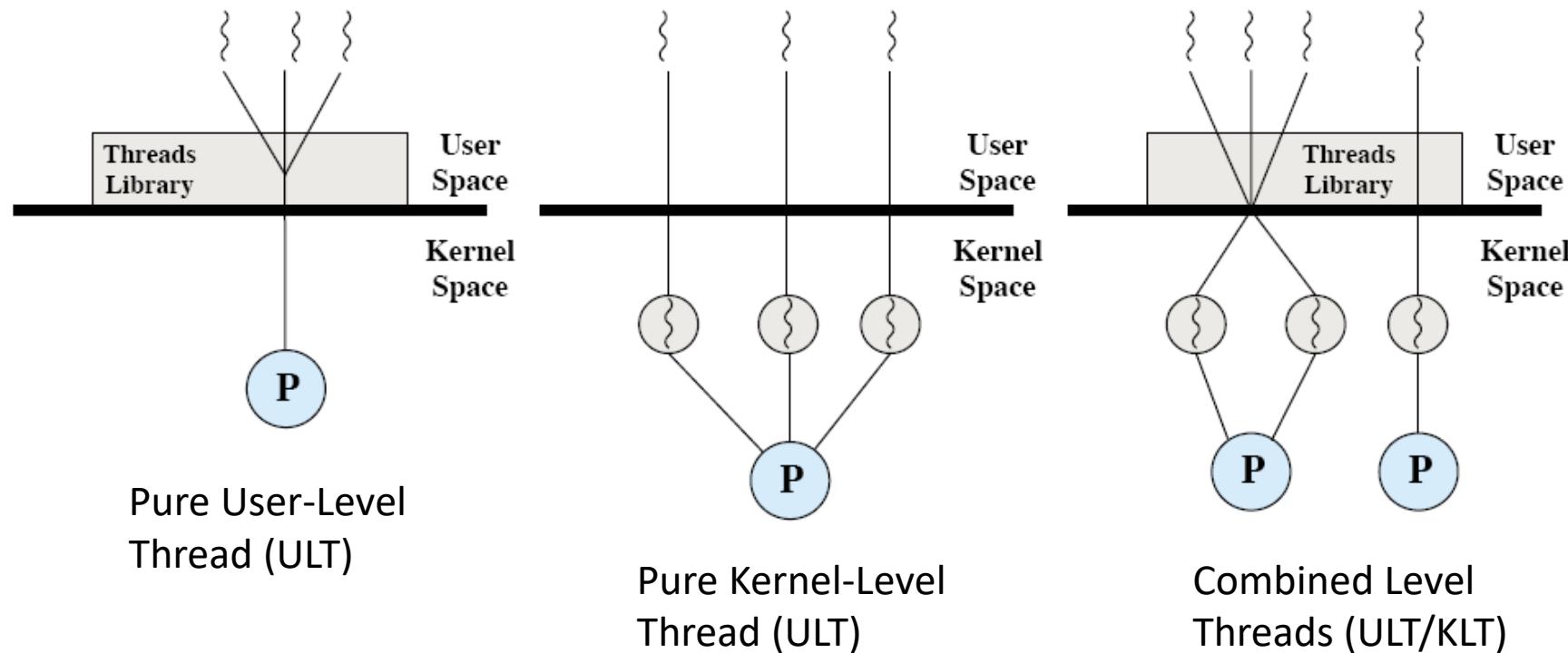
- Process Spawning
- Process creation involve following actions:
- Setting up process control block (PCB)
- Allocation of memory address space
- Loading program into allocated memory address space
- Passing PCB to scheduler to queue up the process to run
- Thread Spawning
- Threads are created within and belonging to processes
- All the threads created within one process share the resources of the process including the address space
- Scheduling is performed on a per-thread basis.
- The thread model is a finer grain scheduling model than the process model
- Threads have a similar lifecycle as the processes and will be managed mainly in the same way as processes are

Why use multi-threaded model over multi-process model?

- A common terminology:
 - Heavyweight Process = Process
 - Lightweight Process = Thread
- Advantages (Thread vs. Process):
 - Much quicker to create a thread than a process
 - spawning a new thread only involves allocating a new stack and a new CPU state block
 - Much quicker to switch between threads than to switch between processes
 - Threads share data easily
- Disadvantages (Thread vs. Process):
 - Processes are more flexible – because they use different memory map
 - No security between threads: One thread can stomp on another thread's data
 - For threads which are supported by user thread package instead of the kernel:
 - If one thread blocks, all threads in task block.

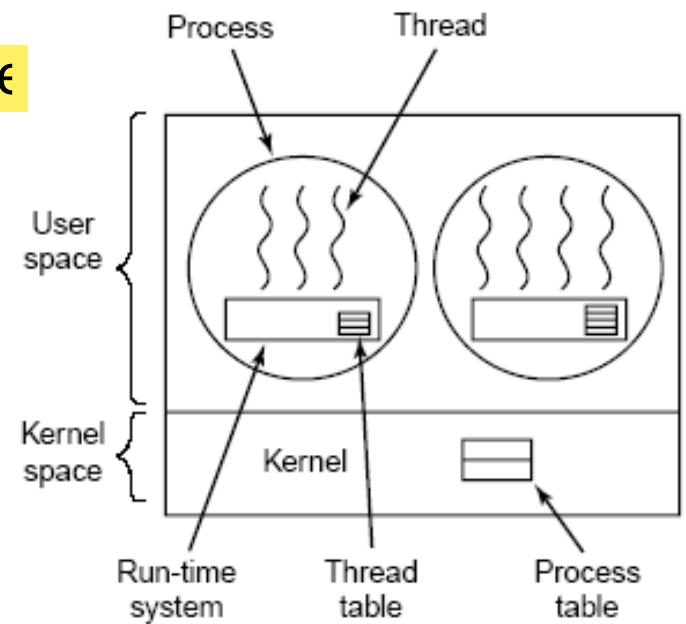
Thread Implementation

- Two broad categories of thread implementation
 - User-Level Threads (ULT)
 - Kernel-Level Threads (KLT)



Thread Implementation

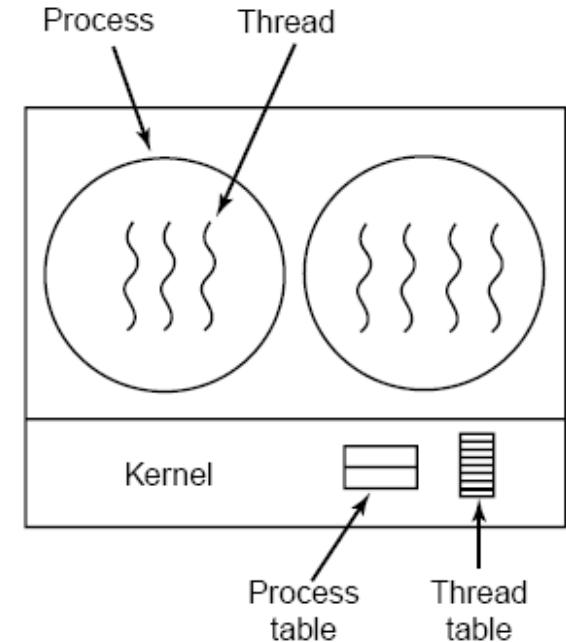
- User-Level Threads (ULT)
 - Kernel is not aware of existence of threads, it knows only processes with one thread of execution
 - Each user process manages its own private thread table
 - Advantages:
 - light thread switching: does not need kernel mode privileges
 - cross-platform: ULT can run on any underlying O/S
 - Disadvantage:
 - if a thread blocks, the entire process is blocked, including all other threads in it



User-Level Thread Package

Thread Implementation

- Kernel-Level Thread (KLT)
 - the kernel knows about and manages the threads: creating and destroying threads are system calls
 - Advantage:
 - fine-grain scheduling, done on a thread basis
 - if a thread blocks, another one can be scheduled without blocking the whole process
 - Disadvantage:
 - heavy thread switching involving mode switch



Kernel-Level Thread Package

POSIX Threads (pthreads) Library

- OS Support: FreeBSD, NetBSD, OpenBSD, Linux, Mac OS X, Android and Solaris. DR-DOS and Microsoft Windows implementation also exists
- Library : libpthread
- Header file : `pthread.h`
- More than 100 threads functions categorized in to 4 category
 - Thread Management
 - Mutexes
 - Condition Variables
 - Synchronization between Threads using read/write locks and barriers
- To compile with gcc you need to use `-pthread` option of gcc to link libpthread

Thread Identification

ConcurrentProgramming\pthread self example.c

```
#include <pthread.h>  
pthread_t pthread_self(void);
```

Returns thread ID of the calling thread

Note: Each process has at least a single thread running by default even if you have not created thread explicitly

Thread Creation

```
#include <pthread.h>  
int pthread_create(pthread_t *tidp, const pthread_attr_t *attr, void  
*(*start_routine)(void*), void *arg);
```

Returns 0 on success and error number in case of failure

Parameters:

- tidp : pointer to thread ID variable which will be set with thread ID for newly created thread
- attr : specific attributes to control the thread (default value to pass is NULL)
- start_routine : pointer to function which will be executed upon creation of a new thread
- arg: pointer to argument(s) to be passed to start_routine function. Note if more than one parameter need to be passed than you need to use structure instead of primitive type

Thread Creation Examples

- Create thread with function with no parameters.

[ConcurrentProgramming\thread example no param.c](#)

- Create thread with function with only one parameter with primitive data type.

[ConcurrentProgramming\thread example one param.c](#)

- Create thread with function with multiple parameters with primitive type or user-defined type.

[ConcurrentProgramming\thread example multiple params.c](#)

Thread Termination

- If thread calls `exit()` system call, complete process will be terminated which is generally not desirable because the goal is to terminate a thread and not the process, hence we use `pthread_exit()` as below

```
#include <pthread.h>
```

```
void pthread_exit(void * retval);
```

Does not return anything

Different between using exit(), return or pthread_exit()

- `exit()` – exit the complete process so if you call from one of the thread function, it will end the process without waiting for main or any other thread in the process to continue → **process ends**
- [ConcurrentProgramming\thread_example1.c](#)

```
$ ./thread_example1.out
```

Main thread 521172736 is starting

child thread id 513017600 is starting

child thread id 513017600 is calling exit

```
$
```

Note: Message “Main thread 521172736 is finished” is never displayed because parent also have terminated when child called `exit()`

Different between using exit(), return or pthread_exit()

- return() – when called from a thread then it will not wait for any of child thread to complete → parent and all the child threads ends

[ConcurrentProgramming\thread_example2.c](#)

\$./thread_example2.out

Main thread 1203812096 is starting

child thread id 1195656960 is starting

Main thread 1203812096 is finished

\$

Note: Message “child thread id 1195656960 is calling exit” is never displayed because child is terminated by parent calling return

Different between using exit(), return or pthread_exit()

- `pthread_exit()` – when called from a thread then it will only end that thread but all other threads in the same process or child threads can continue → **this thread ends but all the child threads or any other threads in the same process continues**

[ConcurrentProgramming\thread_example3.c](#)

```
$ ./thread_example3.out
```

```
Main thread 1709131520 is starting
```

```
child thread id 1700976384 is starting
```

```
Main thread 1709131520 is finished but let the child thread continue
```

```
child thread id 1700976384 is calling exit
```

```
$
```

Note: `pthread_exit()` is called by parent/main thread hence it has terminated but child thread continues to execute until it calls `exit()`

Thread Synchronization

```
#include <pthread.h>  
int pthread_join(pthread_t tid, void **retval);
```

thread calling `pthread_join` waits for target thread with `tid` to terminate.
Target thread is terminated when it calls `pthread_exit()`

Returns 0 on success and error number in case of failure

Parameters:

`tid` : thread ID of target thread for which calling thread is waiting to terminate

`retval` : when target thread called `pthread_exit()`, exit status of the same is returned in `retval`

Complete Thread Example with Synchronization

[ConcurrentProgramming\thread_sync_example.c](#)

```
$ ./thread_sync_example.out
```

```
Main thread is 2739095296
```

```
child thread 2730940160 is created
```

```
child thread 2722547456 is created
```

```
main thread 2739095296 will wait for child thread 2730940160
```

```
child thread 2730940160 exiting
```

```
child thread 2722547456 exiting
```

```
child thread 2730940160 exit code 1
```

```
main thread 2739095296 will wait for child thread 2722547456
```

```
child thread 2722547456 exit code 2
```

```
Main thread 2739095296 exiting
```

```
$
```

Thread Co-Operation

[ConcurrentProgramming\thread_coo
p_example.c](#)

Multiple runs may produce different results

```
$ ./thread_coop_example.out
```

```
counter = 0
```

```
counter = 2
```

```
counter = 3
```

```
counter = 4
```

```
counter = 5
```

```
$ ./thread_coop_example.out
```

```
counter = 1
```

```
counter = 2
```

```
counter = 3
```

```
counter = 3
```

```
counter = 4
```

```
$
```

Thread Synchronization

- One thread can request another thread belonging to the same process to terminate by calling `pthread_cancel()`

```
#include <pthread.h>
```

```
int pthread_cancel(pthread_t tid);
```

Returns 0 on success and error number in case of failure

Parameters:

`tid` : thread ID of target thread which is requested to be cancelled

Note: Behavior of the target thread will be the same as if it had called `pthread_exit()`

Process vs Thread System Calls

Process Primitive	Thread Primitive	Description
fork	pthread_create	Create a new flow of control
exit	pthread_exit	Exit from an existing flow of control
waitpid	pthread_join	Get exit status from flow of control
atexit	pthread_cancel_push	Register function to be called at exit from flow of control
getpid	pthread_self	Get ID of flow of control
abort	pthread_cancel	Request abnormal termination of flow of control

Concurrency Issues

- Most common way of communication between multiple processes or threads on the same system is using shared variables in shared memory
- Let's say two tasks are running a instruction $x = x + 1$. if processes are different and not sharing memory the result produces is the same. But what happens when x is a shared variable.
- Consider $x = x + 1$ is a non-atomic operation (since memory is non-atomic) and two process runs below steps:
 - Read value of x from memory to register ($\text{Reg} \leftarrow x$)
 - Increment value of x in register ($\text{Reg} = \text{Reg} + 1$)
 - Write back the updated value in memory ($\text{Reg} \rightarrow x$)

Concurrency Issue

- Consider $x = 8$ initially and two concurrent Threads execute $x = x+1$

T1	T2	Output
Reg <- x		Reg=8
Reg=Reg+1		Reg=9
Reg -> x		X=9
	Reg <- x	Reg=9
	Reg=Reg+1	Reg=10
	Reg -> x	x=10

Output from serial execution of T1 and T2 which should be generated by concurrent processes as well

T1	T2	Output
Reg <- x		Reg=8
Reg=Reg+1		Reg=9
	Reg <- x	Reg=8
	Reg=Reg+1	Reg=9
Reg -> x		x=9
	Reg -> x	x=9

Output from concurrent (interleaved) execution of T1 and T2

T1	T2	Output
Reg <- x		Reg=8
	Reg <- x	Reg=8
Reg=Reg+1		Reg=9
Reg -> x		x=9
	Reg=Reg+1	Reg=9
	Reg -> x	x=9

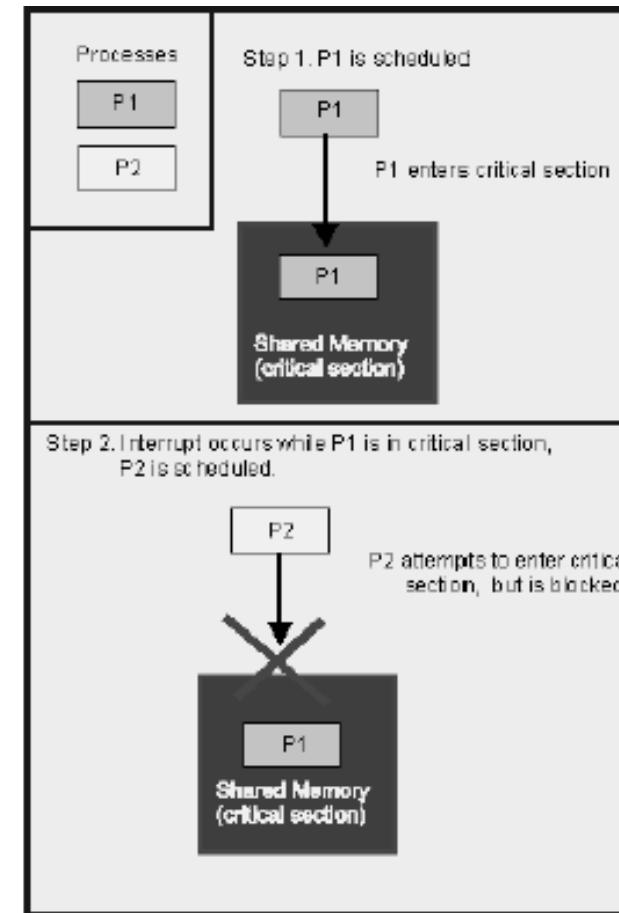
Output from concurrent (interleaved) execution of T1 and T2

Thread Synchronization

- Mechanism that allows programmer to control relative order of operation occurrence in different threads or processes
- How thread synchronization works:
 - Programmer identifies **critical section** in the code
 - Implements **mutual exclusion** to ensure that critical section is mutually exclusive i.e. atomic

Critical Section and Mutual Exclusion

- A section or consecutive lines of code which should be seen as atomic is called critical section
- Synchronization which is needed for critical section is called mutual exclusion



Busy Waiting – Solution for Mutual Exclusion

- Busy waiting is one of the solution to implement synchronization (and a mutual exclusion)
 - Threads writes and reads shared variable flag
 - Thread intent to enter critical section must check flag. If it is false then it can't enter critical section and continuously checks flag to turn true in loop (busy waiting)
 - When it finds flags as turned to true and set flag to false and enters the critical section
 - Upon finishing the critical section code thread turns the flag to true again

Threads T1 and T2 execute the following code:

```
While flag == false  
    do nothing  
flag=false  
enter critical section  
...  
finish critical section  
flag = true
```

Problems with Busy Waiting

- Occupies CPU resources and waste precious CPU cycles
- Generates excessive traffic on bus or internal network
- Race Condition (**Why?**)
 - Think what would happen if both threads T1 and T2 intent to check the flag at exactly the same time and found that the flag is false. Both will be allowed to enter critical section which violate mutual exclusion rule

Suspend and Resume - – Solution for Busy Waiting

- Instead of waiting the loop continuously thread can be suspended when cant enter the critical section
- It can be resumed (woken up) when it can enter the critical section (**Generally done through sending signal**)
- For example, assume that T2 has already entered critical section hence the flag is false. When T1 tries to enter critical section and checks flag which is false it is suspended. After finishing critical section T2 will resume T1 by sending signal
- **Disadvantage:** Still leads to race condition
Why ?
 - Flag is a shared variable which is used to control the action
 - Testing and action (suspend) should be done atomically

Threads T1 and T2 execute this code:
if flag == false
 suspend T?
flag=false
enter critical section
...
finish critical section
flag = true
resume T?

Mutual Exclusion – Solution – Not possible in multi-core environment Why?

Thread T1

```
loop  
  flag1 := up  
  turn := 2  
  while flag2 = up and turn  
= 2 do  
    null  
  end  
  critical section  
  flag1 := down  
  non-critical section  
  end  
end T1
```

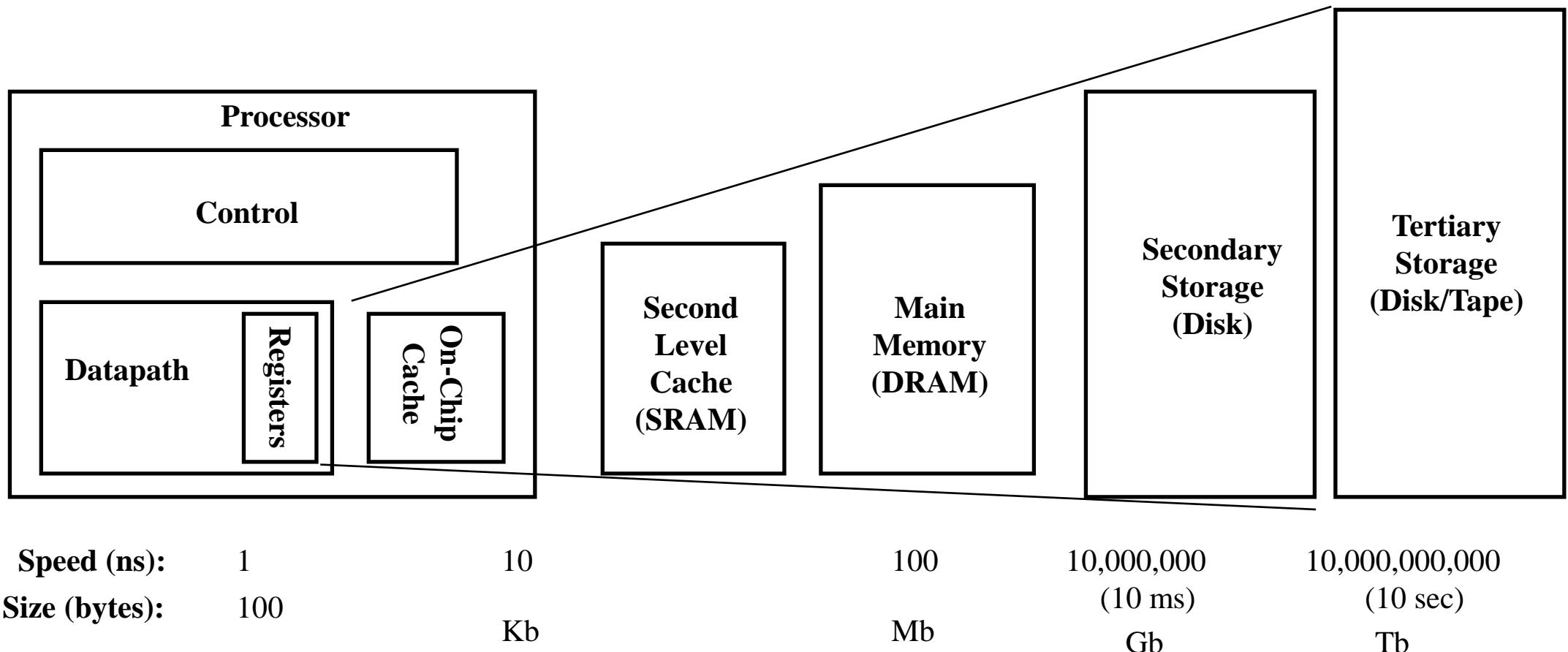
Thread T2

```
loop  
  flag2 := up  
  turn := 1  
  while flag1 = up and  
turn = 1 do  
    null  
  end  
  critical section  
  flag2 := down  
  non-critical section  
  end  
end T2
```

- Now we can guarantee that both the threads never enters the critical section for an infinite time
- And thread which starts its pre-protocol will eventually enter the critical section regardless of behavior of other thread

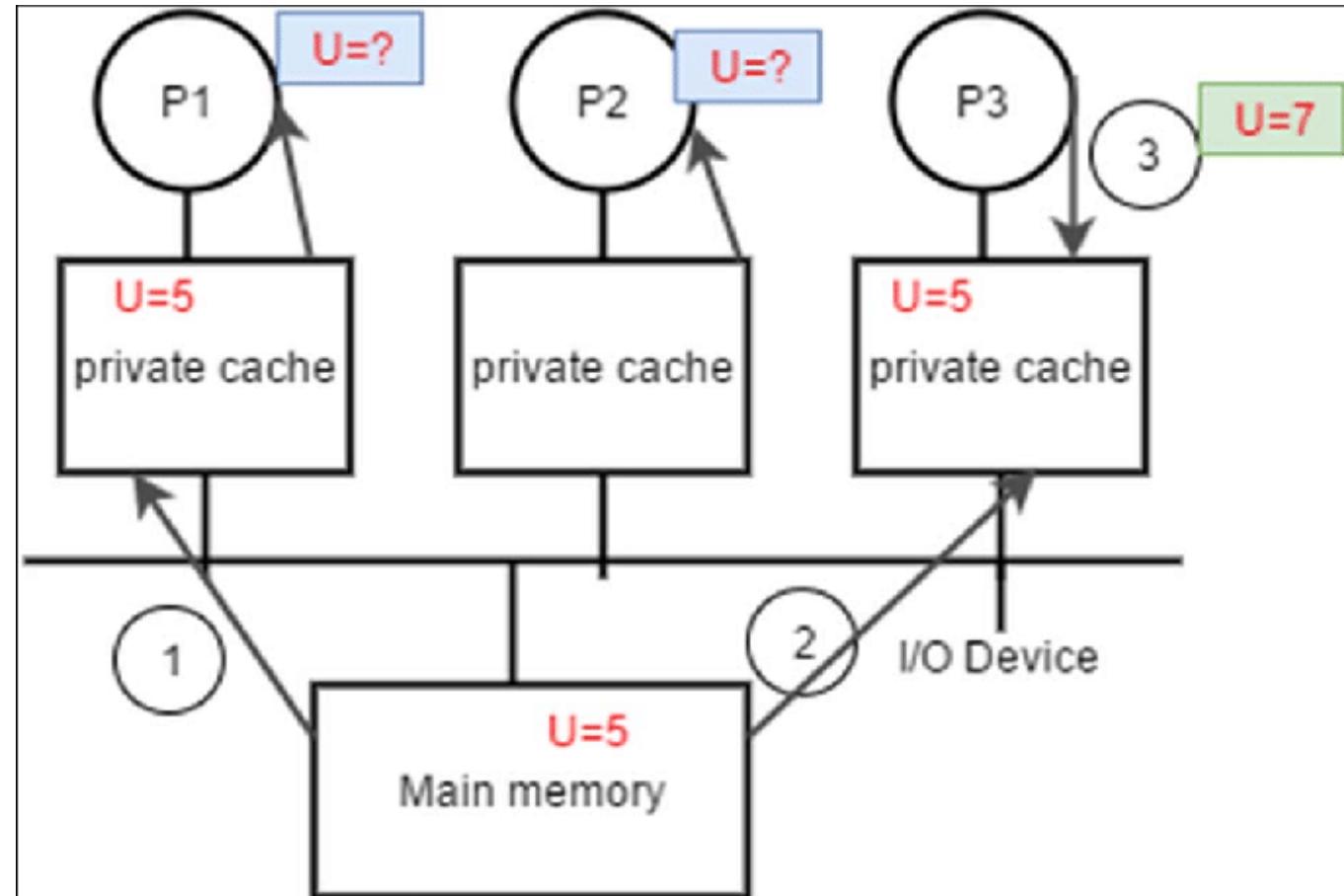
A Modern Memory Hierarchy

- By taking advantage of the principle of locality:
 - Present the user with as much memory as is available in the cheapest technology.
 - Provide access at the speed offered by the fastest technology.



Cache Coherence Problem

1. P1 reads value of U variable from main memory to private cache
2. P3 reads value of U variable from main memory to private cache
3. P3 updates value of U to 7 in private cache
4. Just before P3 gets a chance to update U in main memory, P2 reads from main memory and P1 from private cache has value of 5 but current value should have been 7



Cache Coherency Problem Solution

- Processor (P3) updating the value performs atomic update to cache and memory at the same time.
- Lock the system bus for all other processors (P1 and P2) so they cannot read or write until the data is written by processor (P3) who is updating the value.
- Make the private cache values as dirty for all other processors (P1 and P2) so that subsequent read must come from main memory only

MUTEX implementation in today's Multi-core System

```
static __always_inline bool __mutex_trylock_fast(struct mutex *lock) { unsigned long curr = (unsigned long)current; unsigned long zero = 0UL;  
if (__atomic_long_try_cpxchg_acquire(&lock->owner, &zero, curr)) return true; return false; }  
  
#define __atomic_long_try_cpxchg_acquire(l, old, new) \  
__ATOMIC_LONG_PFX(__try_cpxchg_acquire)((__ATOMIC_LONG_PFX(_t) *)(l), \  
(__ATOMIC_LONG_TYPE *)(old),  
(__ATOMIC_LONG_TYPE)(new))  
  
#define __ATOMIC_LONG_PFX(x) atomic64 ## x
```

[ConcurrentProgramming\test_atomic.c](#)

Make file to create object file

[ConcurrentProgramming\test_atomic_makefile](#)

Copy make file as Makefile and run
make to create object files

Now create assembly file →

objdump -d -S test_atomic.ko >
test_atomic.assembly

MUTEX implementation in today's Multi-core System

```
10:  c7 00 02 00 00 00      movl $0x2,(%rax)
16:  48 8b 05 00 00 00 00  mov  0x0(%rip),%rax    # 1d <init_module+0x1d>
1d:  f0 83 00 02          lock addl $0x2,(%rax)
21:  5d                  pop  %rbp
```

lock instruction tell processor running this code to lock the data bus so that other processor cannot perform any read or write operation. In other words lock instruction is implemented at hardware level

Mutual Exclusion Implementation in POSIX Thread - MUTEX

MUTEX is like a key to access the critical section that has access to only one thread at a time

```
#include <pthread.h>
```

MUTEX variable (containing union of structures) is represented as pthread_mutex_t data type defined in /usr/include/bits/pthreadtypes.h

Before we can use mutex variable memory allocation has to be done for which use function:

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr) → return 0 on success
```

When mutex variable is no longer required memory should be freed for which we use function:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex) → return 0 on success
```

Mutual Exclusion Implementation in POSIX Thread - MUTEX

```
#include <pthread.h>
```

To lock mutex: if mutex is already locked by another thread, thread trying to load mutex will be blocked

```
int pthread_mutex_lock(pthread_mutex_t *mutex) → return 0 on success
```

To lock mutex: if mutex is already locked by another thread, function will rerun error code EBUSY

```
int pthread_mutex_trylock(pthread_mutex_t *mutex) → return 0 on success
```

To unlock mutex:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex) → return 0 on success
```

Avoidance of Race Condition using MUTEX

ConcurrentProgramming\thread_racecond without mutex.c

Race condition output without using MUTEX (remove all pthread_mutex_* function calls)

\$./thread_racecond.out → **Race condition exists without using MUTEX**

12345678901abcde23fgh456789012345ij6789012345678kl90mnopqrstuvwxyzabcdefghijkl
mnopqrstuvwxyz

ConcurrentProgramming\thread_racecond with mutex.c

With MUTEX implementation never get race condition

\$./thread_racecond.out

1234567890123456789012345678901234567890abcdefghijklmnopqrstuvwxyzabcdefghijkl
mnopqrstuvwxyz

OR

\$./thread_racecond.out

abcdefghijklmnopqrstuvwxyzabcdefghijklmнопqrstuvwxyz12345678901234567890123456
78901234567890

Disadvantage of MUTEX

- Deadlock can occur if:
 - Thread locking the same **MUTEX** twice
 - Thread1 holding lock on mutex1 wants to get lock on mutex2 while thread2 holding lock on mutex2 wants to get lock on mutex1
 - Solution: As much as possible use `pthread_mutex_trylock` function
- Only one thread is allowed to lock mutex but in some situations such as Reader-Write problem or **Dinning-Philosopher** problem multiple locks should be possible
 - Solution: Reader-Writer Locks and Semaphores

Reader-Writer Locks

- Multiple readers should be allowed to have a lock on mutex
- Only one writer is allowed to have a lock on mutex
- Reading and Writing activities are mutually exclusive i.e. if reader lock is already there writer is not allowed and vice versa
- In most implementation, if writer requesting lock is waiting due mutex which already has a reader lock then reader requests after the writer request will also wait so that writer is not waiting indefinitely (known as starvation problem)

```
#include <pthread.h>
```

To initialize use i.e. allocate memory:

```
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr_t *restrict attr)
```

To free the allocated memory: It will destroy the lock even if there are already locks on it so before destroy application should keep track of all the locks and unlock them before calling destroy

```
int pthread_rwlock_destroy(pthread_rwlock_t *restrict rwlock)
```

Reader-Writer Locks

```
#include <pthread.h>
```

For requesting read lock: **reader will be blocked if there is already a write lock on mutex**

```
int pthread_rwlock_rdlock(pthread_rwlock_t *restrict rwlock)
```

For requesting write lock: **writer will be blocked if there is already at least one read lock or another write lock on mutex**

```
int pthread_rwlock_wrlock(pthread_rwlock_t *restrict rwlock)
```

For requesting unlock: **works for both read or write. If multiple reader locks are held, you have to initiate the same number of unlock for release (i.e. only the last unlock will release the lock otherwise it remains in reader lock state)**

```
int pthread_rwlock_unlock(pthread_rwlock_t *restrict rwlock)
```

Non-blocking functions:

```
int pthread_rwlock_tryrdlock(pthread_rwlock_t *restrict rwlock)
```

```
int pthread_rwlock_trywrlock(pthread_rwlock_t *restrict rwlock)
```

[ConcurrentProgramming\reader_writer.c](#)

Reader-Writer Locks Usage

- Producer-Consumer Problem
- Several Operating System Data Structures:
 - Memory Page Tables
 - Process Control Blocks
 - Kernel Thread Tables
 - etc.

Dining-Philosopher Problem

- Five philosophers spend their time eating and thinking.
- They are sitting in front of a round table with spaghetti served.
- There are five plates at the table and five chopsticks set between the plates.
- Eating the spaghetti **requires the use of two chopsticks** which the philosophers pick up one at a time.
- Philosophers do not talk to each other.
- Semaphore **chopstick [5]** initialized to **1**



Semaphores

- Allow multiple locks
- Semaphore S → integer variable
- Modified by two operations → wait() and signal()
- wait() – originally called P() for Dutch word “proberen” which means try
 - wait(S) {
 while S <= 0; // no-op
 S--;
}
- signal() – originally called V() for Dutch word “verhogen” which means increase
 - signal(S) {
 S++;
}
- Note wait() and signal() are atomic operation
- Semaphore Types:
 - Binary Semaphore: S value can be 0 (locked) or 1 (unlocked)
 - Counting Semaphores: S value can any integer
 - S=0 → locked by at least one thread
 - S=-n → n threads are waiting to acquire a lock
 - S=n → n locks are available for threads

Semaphore Usage

- Binary Semaphore in place of mutex
- Counting Semaphore used when multiple resources are to be shared with multiple consumers
 - Processes or threads sharing processors
 - Processes or threads sharing multiple network devices
 - Courses sharing classrooms
 - Faculties sharing office computers
 - etc.

Thread Synchronization using Semaphores

Example of Binary Semaphore

//Thread1:

```
int t;  
wait(sem)  
sum = sum + x;  
t = sum;  
  
...  
signal(sem);
```

//Thread2:

```
int t;  
wait(sem)  
sum = sum + y;  
t = sum;  
  
...  
signal(sem);
```

Dining-Philosopher Problem: Solution

The structure of Philosopher i:

```
do {  
    wait ( chopstick[i] ); //lock  
    wait ( chopStick[ (i + 1) % 5] ); //lock  
    // eat  
    signal ( chopstick[i] ); //unlock  
    signal (chopstick[ (i + 1) % 5] );  
    //unlock  
    // think  
} while (true);
```

- Problem is deadlock : when all philosopher decides to eat at the same time each will pick up one of the 2 chopsticks they need and wait for the other
- Solution: Allow to lock if both chopsticks are available at the same time
 - Lock the 1st chopstick
 - For 2nd chopstick, check if it can be locked otherwise release the 1st one

POSIX: Semaphores

- creating a semaphore:

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- initializes a semaphore object pointed to by sem to integer value of “value”
- pshared is a sharing option; a value of 0 means the semaphore is local to the calling process i.e. shared by all threads of the same process; a +ve number indicates it can be shared across multiple processes using shared memory instructions

- terminating a semaphore:

```
int sem_destroy(sem_t *sem);
```

- frees the resources allocated to the semaphore sem
- usually called after pthread_join()
- an error will occur if a semaphore is destroyed for which a thread is waiting

POSIX: Semaphores

- semaphore control:

```
int sem_post(sem_t *sem); → same as signal()
```

sem_post atomically increases the value of a semaphore by 1, i.e.,
when 2 threads call sem_post simultaneously, the semaphore's value
will also be increased by 2 (there are 2 atoms calling)

```
int sem_wait(sem_t *sem);
```

sem_wait atomically decreases the value of a semaphore by 1; but
always waits until the semaphore has a non-zero value first

Semaphore Example

1 Resource, 1 semaphore with 1 thread

[ConcurrentProgramming\semaphore1.c](#)

\$./semaphore1.out

Starting thread, semaphore is unlocked.

Hello from thread!

Hello from thread!

Semaphore locked.

Semaphore unlocked.

Hello from thread!

\$

- [ConcurrentProgramming\semaphore2.c](#)
- 2 Semaphores and 2 Resources with 4 threads

Dining-Philosopher Problem: Solution

[ConcurrentProgramming\dining-philosopher.c](#)

Philosopher 1 is thinking

Philosopher 2 is thinking

Philosopher 3 is thinking

Philosopher 4 is thinking

Philosopher 5 is thinking

Philosopher 1 is Hungry

Philosopher 2 is Hungry

Philosopher 3 is Hungry

Philosopher 4 is Hungry

Philosopher 5 is Hungry

Philosopher 5 takes fork 4 and 5

Philosopher 5 is Eating

Philosopher 5 putting fork 4 and 5 down

Philosopher 5 is thinking

Philosopher 4 takes fork 3 and 4

Philosopher 4 is Eating

Philosopher 1 takes fork 5 and 1

Philosopher 1 is Eating

Philosopher 4 putting fork 3 and 4 down

Philosopher 4 is thinking

Philosopher 3 takes fork 2 and 3

Philosopher 3 is Eating

Philosopher 5 is Hungry

Philosopher 1 putting fork 5 and 1 down

Philosopher 1 is thinking

Philosopher 5 takes fork 4 and 5

Philosopher 5 is Eating

.....

Additional Slides

[Left from Here](#)

Mutual Exclusion

- Problem Statement:
- Suppose we have 2 threads with structure on the right side
- In which way the **protocol** can be implemented so that we can guarantee the mutual exclusion?

Thread T
loop
entry **protocol**
critical section
exit **protocol**
non-critical section
end
End T

Mutual Exclusion – Problem 1

Thread T1

```
loop  
  flag1 := up  
  while flag2 = up do  
    null  
  end  
  critical section  
  flag1 := down  
  non-critical section  
  end  
end T1
```

Thread T2

```
loop  
  flag2 := up  
  while flag1 = up do  
    null  
  end  
  critical section  
  flag2 := down  
  non-critical section  
  end  
end T2
```

What is the problem here?

- T1 sets flag1 := up and at the same time T2 sets flag2 := up
- T2 test flag1 and goes into busy-wait
- T1 test flag2 and goes into busy-wait
- **Problem : livelock**
- In this case both will go into busy-wait
- **Why? None of the threads checks upfront if it is possible to enter the critical section**

Mutual Exclusion – Solution

Thread T1

```
loop  
  while flag2 = up do  
    null  
  end  
  flag1 := up  
  critical section  
  flag1 := down  
  non-critical section  
end  
end T1
```

Thread T2

```
loop  
  while flag1 = up do  
    null  
  end  
  flag2 := up  
  critical section  
  flag2 := down  
  non-critical section  
end  
end T2
```

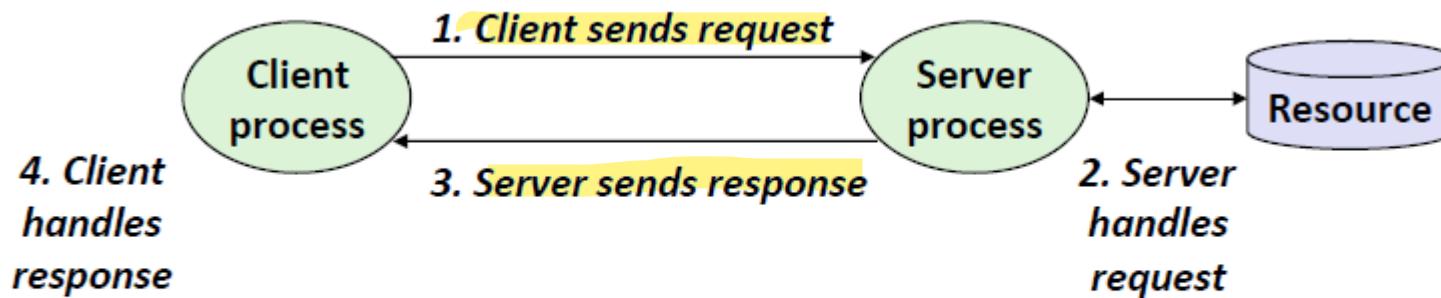
What is the problem now?

- T1 and T2 not in critical section (flag1 := down and flag2 := down)
- T2 test flag1 → found down
- T1 test flag2 → found down
- T2 sets flag2 := up and executes critical section
- T1 sets flag1 := up and executes critical section
- Problem : T1 and T2 both are in critical section (FAULT)
- Why? Testing of flag of other thread and setting its own flag are 2 individual action but they need to be atomic

Systems
Software/Programming
Network Programming

A Client-Server Transaction

- Most network application are based on client-server model
 - A server process and one or more client processes
 - Server manages some resource
 - Server provides services to client by managing client resources
 - Server activated by request from client

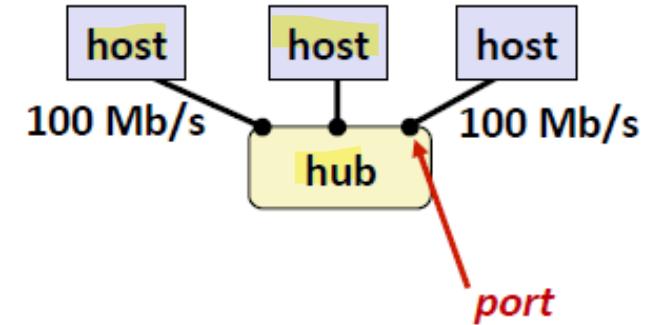


*Note: clients and servers are processes running on hosts
(can be the same or different hosts)*

Computer Network

- A **network** is a hierarchical system of boxes and wires organized by geographical proximity
 - SAN (System Area Network) spans a cluster or machine room Switched Ethernet, Quadrics QSW, ...
 - LAN (Local Area Network) spans a building or campus Ethernet is most prominent example
 - WAN (Wide Area Network) spans a country or world Typically high-speed point-to-point phone lines
- An **internetwork (internet)** is an interconnected set of networks The Global IP Internet (uppercase “I”) is the most famous example of an internet (lowercase “i”)
- Let's see how an internet is built from the ground up

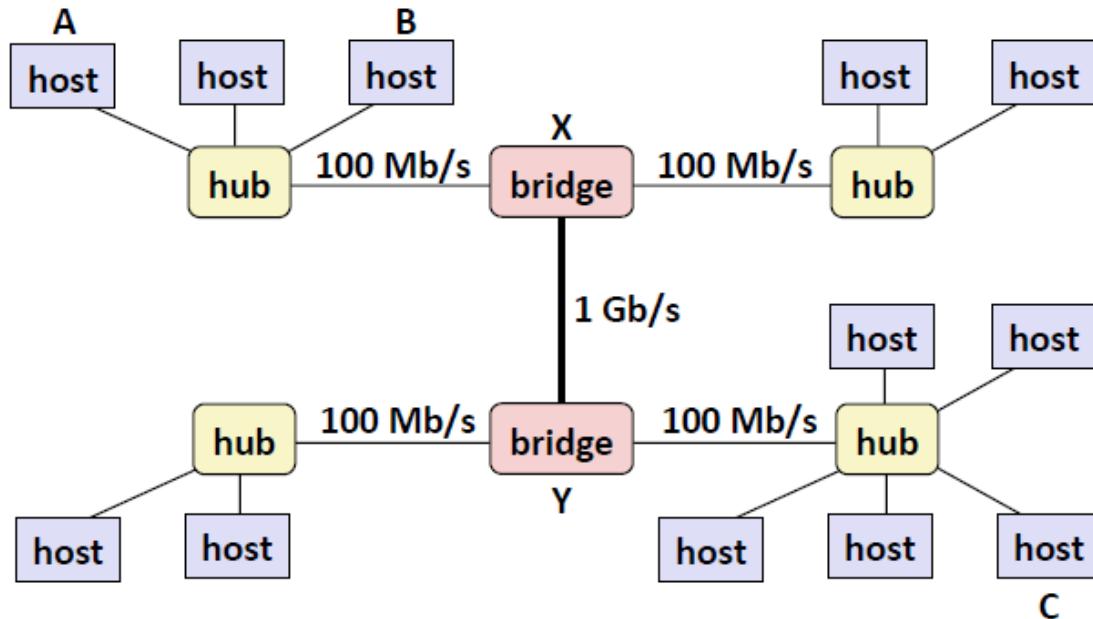
Lowest Level : Ethernet Segment



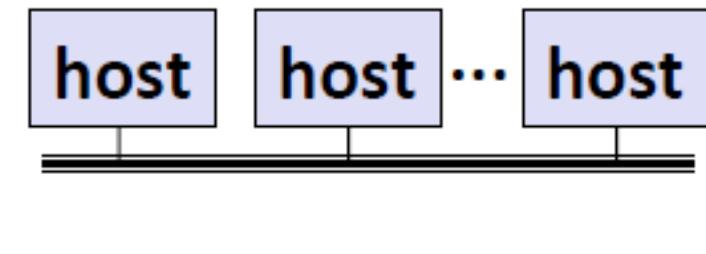
- Ethernet segment consists of a collection of hosts (twisted pairs) to a hub
- Spans room or floor in a building
- Operation
 - Each Ethernet adapter has a unique 48-bit address (MAC address)
 - E.g., 00:16:ea:e3:54:e6
 - Hosts send bits to any other host in chunks called frames
 - Hub slavishly copies each bit from each port to every other port
 - Every host sees every bit
 - Note: Hubs are on their way out. Bridges (switches, routers) became cheap enough to replace them

Next Level : Bridged Ethernet Segment

- Spans building or campus
- Bridges cleverly learn which hosts are reachable from which ports and then selectively copies frames from port to port

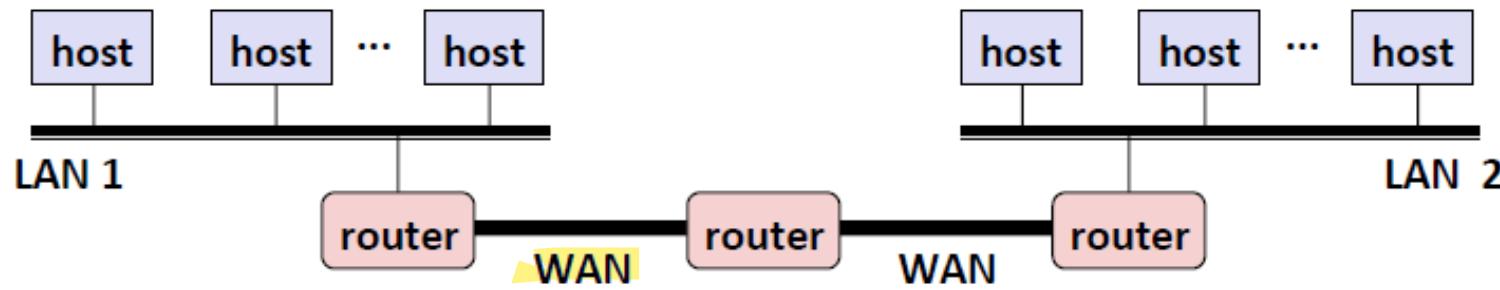


For simplicity hubs and bridges including wires are shown as a line called Local Area Network (LAN)



Next Level : internets (internetworks)

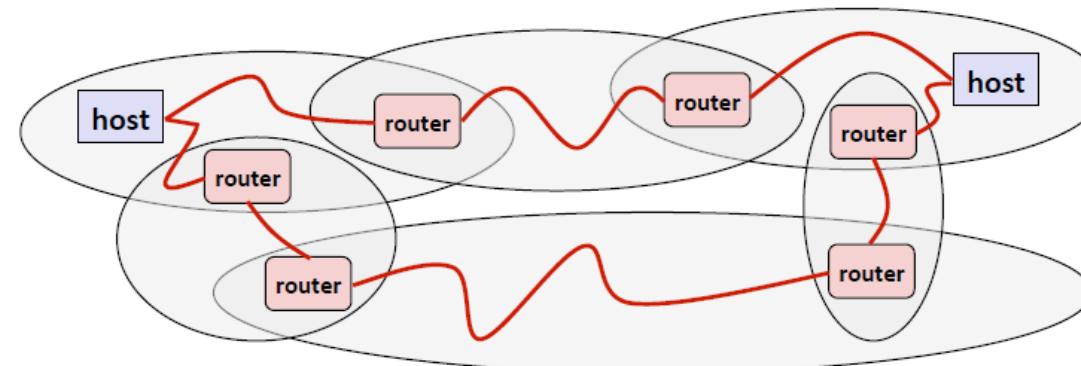
- Multiple incompatible LANs can be physically connected by special computers called **routers**
- The connected networks are called **internet**



*LAN 1 and LAN 2 might be completely different, totally incompatible
(e.g., Ethernet, Fibre Channel, 802.11*, T1-links, DSL, ...)*

Logical Structure of an internet

- Ad hoc interconnection of networks
 - No particular topology
 - Vastly different routers and link capacities
- Send packets from source to destination by hopping through networks
 - Router forms bridge from one network to another
 - Different packets may take different routes



Notion of internet Protocol

- How is it possible to send bits across incompatible LANs and WANs?
- Solution: **protocol** software running on each host and router
 - Protocol is a set of rules that governs how hosts and routers should cooperate when they transfer data from network to network.
 - Smooths out the differences between the different networks

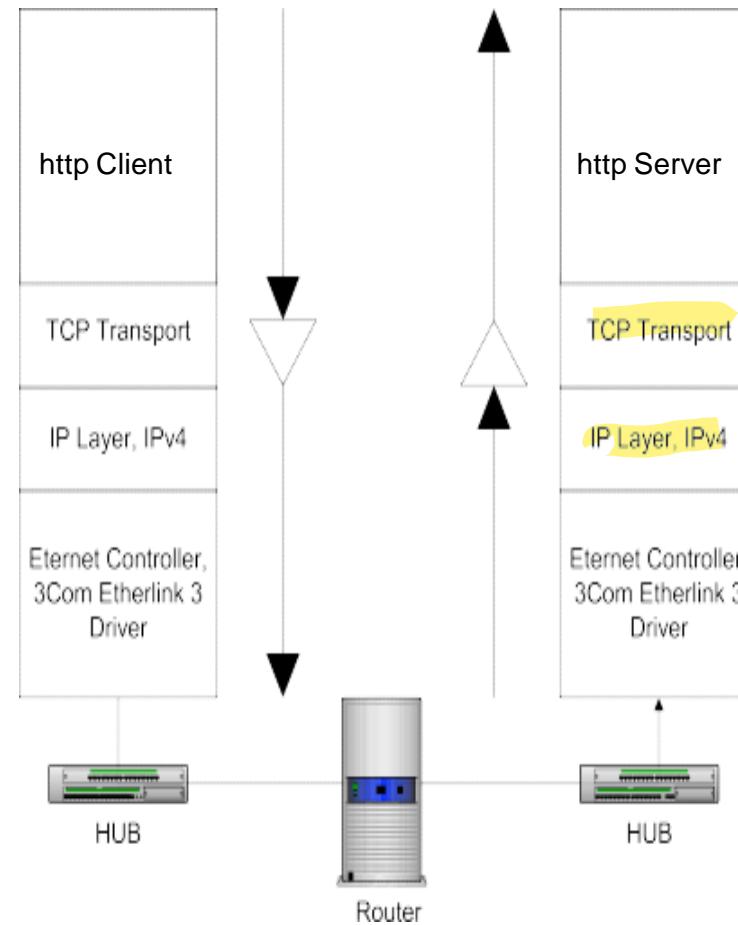
What Does internet Protocol Do?

- Provides a naming scheme
 - An internet protocol defines a uniform format for **host addresses**
 - Each host (and router) is assigned at least one of these internet addresses that uniquely identifies it
- Provides a delivery mechanism
 - An internet protocol defines a standard transfer unit (**packet**)
 - Packet consists of **header** and **payload**
 - Header: contains info such as packet size, source and destination addresses
 - Payload: contains data bits sent from source host

Global IP (Internet Protocol)

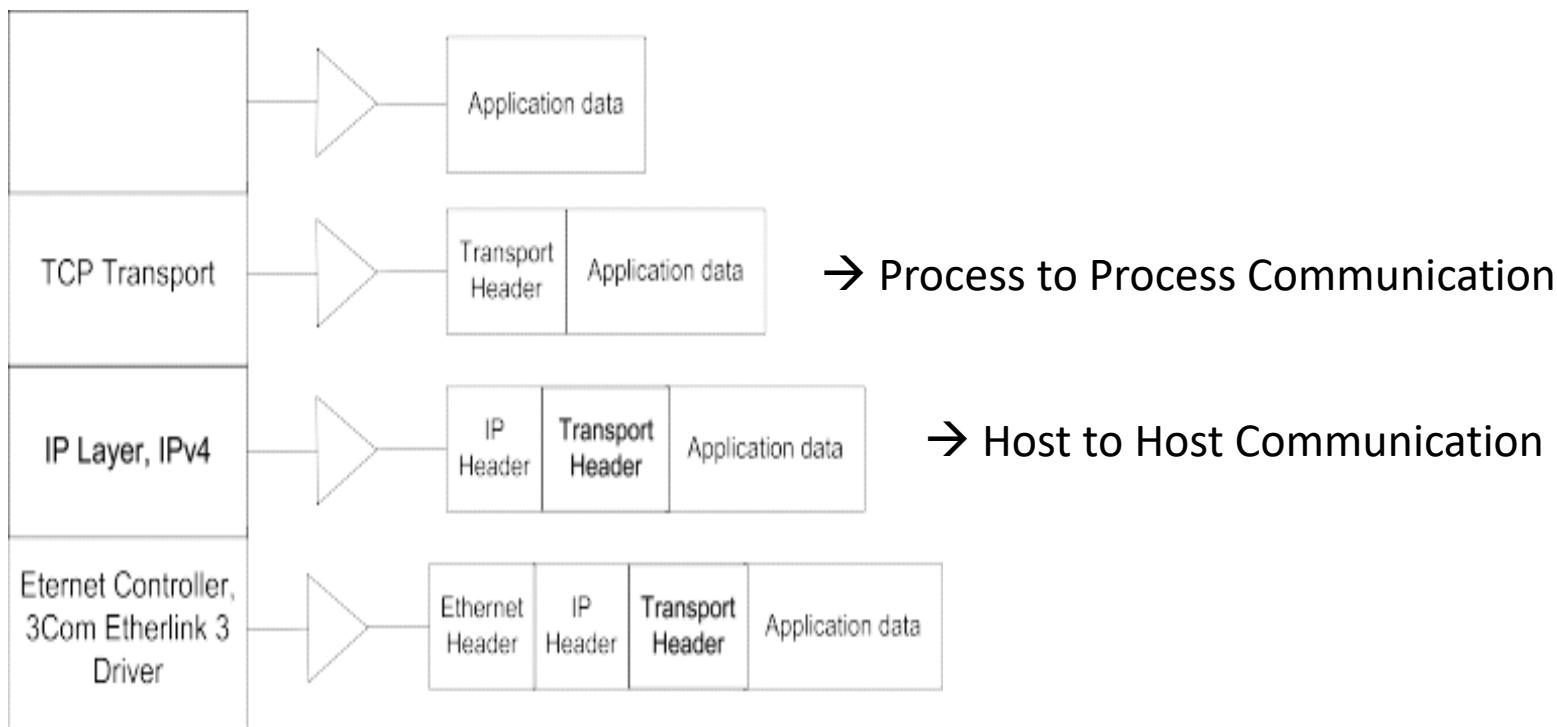
- Based on the TCP/IP protocol family
 - IP (Internet Protocol) :
 - Provides **basic naming scheme** and unreliable **delivery capability** of packets (datagrams) from host--to--host
 - UDP (User Datagram Protocol)Used to send data to multiple receiver like broadcasting without any much concern about quality and time of data
 - Uses IP to provide **unreliable** datagram delivery from **process--to--process**
 - TCP (Transmission Control Protocol)Used to send data to particular receiver on time
 - Uses IP to provide **reliable** byte streams from **process--to--process** over connections
- Accessed via a mix of Unix file I/O and functions from the **sockets interface**

How Client Communicate to Server ?



How Client Communicate to Server ?

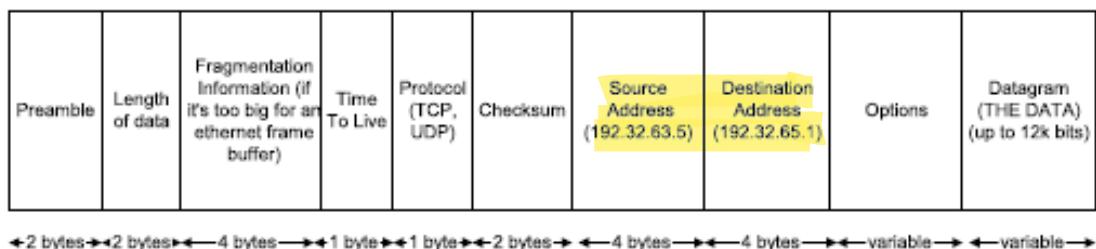
- Client Application puts data through sockets
- Each successive layer wraps the received data with its own header:



TCP/IP Header Formats

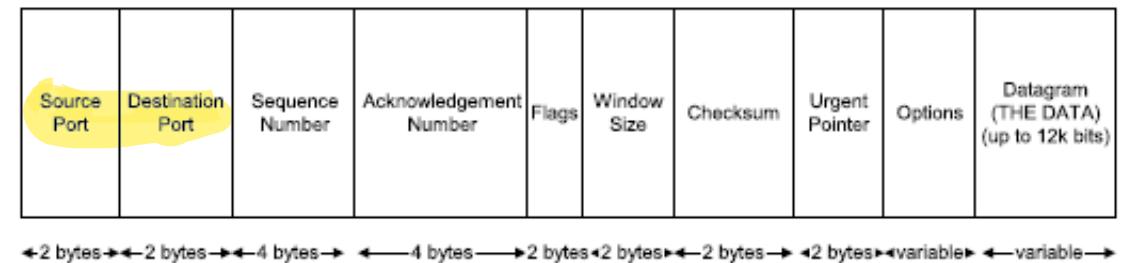
IP Header Format

- Packets may be broken up called **fragments** if data is too large to fit in a single packet
- Packets if not delivered will live in the network till Time-To-Live



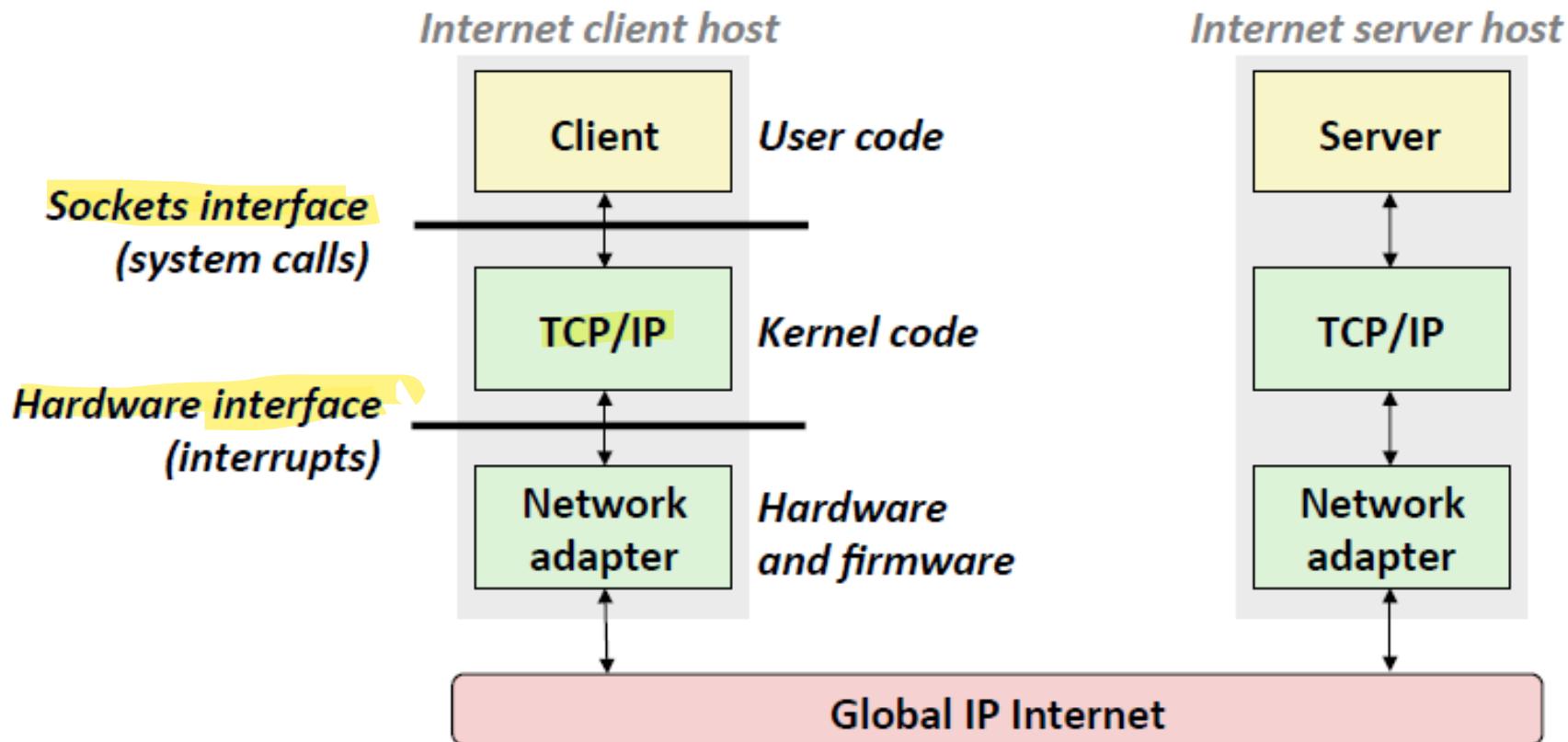
TCP Header Format

- Source and Destination ports
- Sequence number indicates which byte in overall data this **segment** starts with
- Acknowledgement number indicates all bytes up to which recipient has received successfully



Hardware and Software Organization of an Internet Application

socket address is combination of IP address and port number



Internet From a Programmers View

1. Hosts are mapped to a set of 32--bit **IPv4 addresses** e.g. 128.2.203.179
2. The set of IP addresses is mapped to a set of identifiers called Internet **domain names**.
 - 104.238.110.159 is mapped to www.daiict.ac.in
\$ ping www.daiict.ac.in
PING www.daiict.ac.in (104.238.110.159) 56(84) bytes of data.
64 bytes from ip-104-238-110-159.ip.secureserver.net (104.238.110.159): icmp_seq=1 ttl=57
time=349 ms
^C
--- www.daiict.ac.in ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 349.388/349.388/349.388/0.000 ms
You can also use <https://www.whatismyip.com/dns-lookup/> to get **IP address mapped to domain name.**
3. A process on one Internet host can communicate with a process on another Internet host over a **Internet connection**

(1) IP Addresses

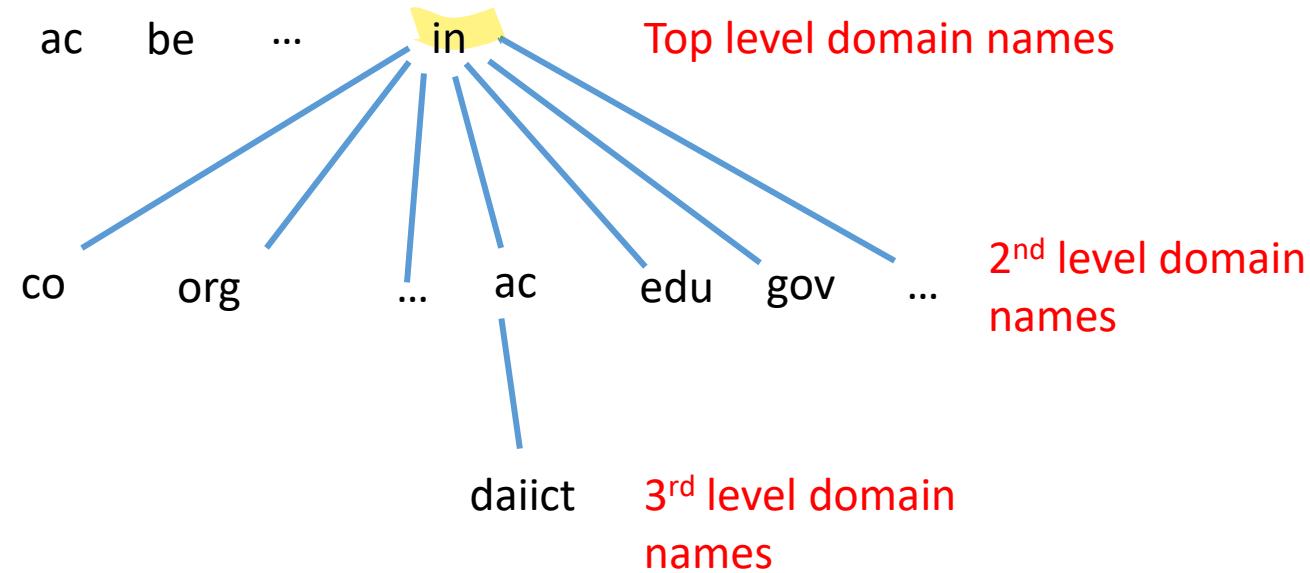
- 32-bit IP addresses are stored in an **IP address struct**
 - IP addresses are always stored in memory in **network byte order** (big-endian byte order)
 - True in general for any integer transferred in a packet header from one machine to another.
 - E.g., the port number used to identify an Internet connection.
- By convention, each byte in a 32-bit IP address is represented by its decimal value and separated by a period
 - IP address: 0x8002C2F2 = 128.2.194.242 → Dotted Decimal Format
 - Big-endian

1000	1001	1002	1003
LSB	0xF2	0xC2	0x02
0x80	MSB	MSB	LSB
 - Little-endian

1000	1001	1002	1003
0x80	0x02	0xC2	0xF2
MSB	MSB	LSB	LSB
- Use getaddrinfo and getnameinfo functions to convert between IP addresses and dotted decimal format.

(2) Internet Domain Names

- The Internet maintains a mapping between IP addresses and domain names in a huge worldwide distributed database called DNS
- Conceptually, programmers can view the DNS database as a collection of millions of host entries.
 - Each host entry defines the mapping between a set of domain names and IP addresses.
 - In a mathematical sense, a host entry is an equivalence class of domain names and IP addresses.



Properties of DNS Mappings

- Can explore properties of DNS mappings using nslookup
 - Output edited for brevity
- Each host has a locally defined domain name localhost which always maps to the loopback address 127.0.0.1

```
$ nslookup localhost
```

Address: 127.0.0.1

- Use hostname to determine real domain name of local host:

```
$ hostname
```

faculty-OptiPlex-3040

- Simple case: one--to--one mapping between domain name and IP address:

```
$ nslookup abel.daiict.ac.in
```

Address: 10.100.71.142

Properties of DNS Mappings

- **Multiple domain** names mapped to the **same IP** address:

```
$ nslookup cs.mit.edu
```

Address: 18.25.0.23

```
$ nslookup eecs.mit.edu
```

Address: 18.25.0.23

- **Same domain** names mapped to **multiple IP** addresses

```
$ nslookup www.google.com
```

Address: 74.125.200.103

Address: 74.125.200.105

Address: 74.125.200.104

Address: 74.125.200.99

Address: 74.125.200.106

(3) Internet Connections

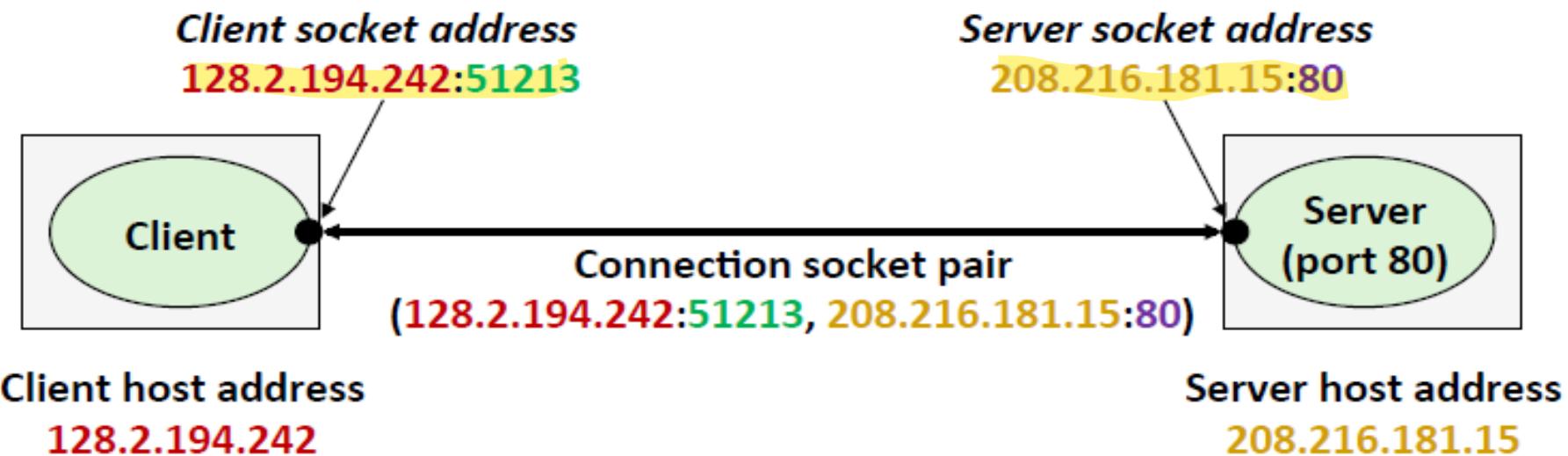
- Clients and servers communicate by sending streams of bytes over **connections**. Each connection is:
 - Point-to-point: connects a pair of processes.
 - Full-duplex: data can flow in both directions at the same time,
 - Reliable: stream of bytes sent by the source is eventually received by the destination in the same order it was sent.
- A **socket** is an endpoint of a connection
 - Socket address is an **IPaddress:port pair**
- A **port** is a 16--bit integer that identifies a process:
 - **Ephemeral port**: Assigned automatically by client kernel when client makes a connection request.
 - **Well-known port**: Associated with some service provided by a server (e.g., port 80 is associated with Web servers)

Well-known Ports and Service Names

- Popular services have permanently assigned **well-known ports** and corresponding **well-known service names**:
 - echo server: 7/echo
 - ssh servers: 22/ssh
 - email server: 25/smtp
 - Web servers: 80/http
 - File Transfer Protocol server : 21/ftp
- Mappings between **well-known ports and service names** is contained in the file **/etc/services** on each Linux machine.

Anatomy of Connection

- connection is uniquely identified by the socket addresses of its endpoints (**socket pair**) :
 - (clientIPaddr:clientport, serverIPaddr:serverport)

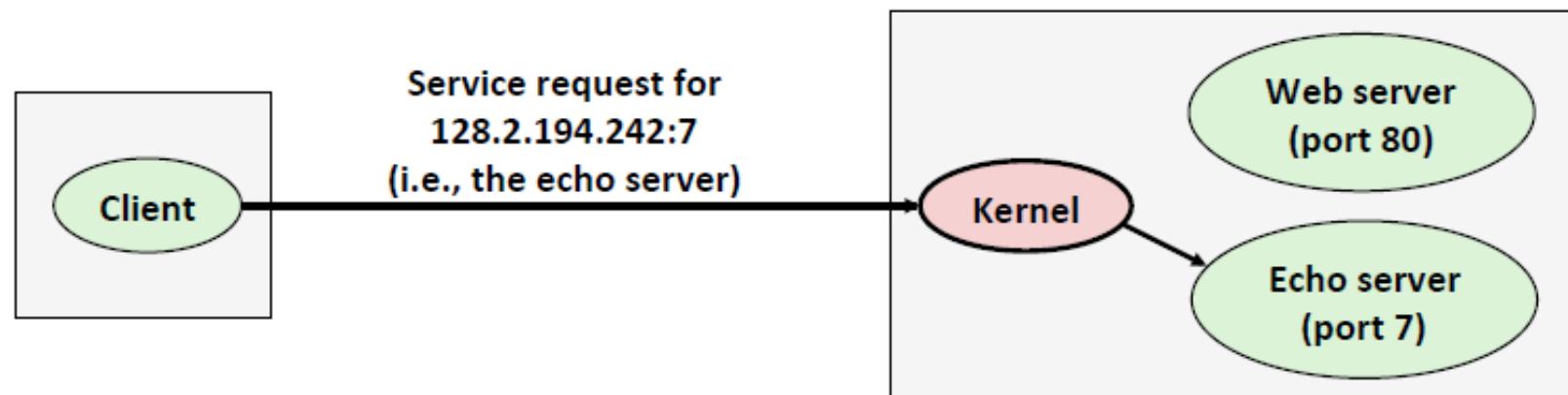
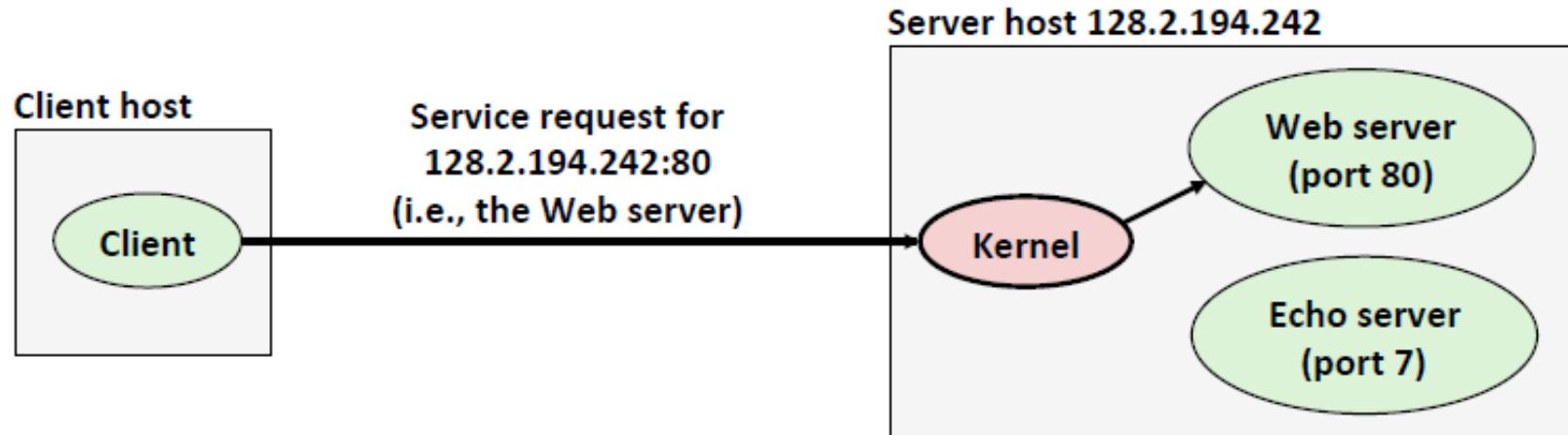


51213 is an ephemeral port
allocated by the kernel

assigned by OS to client

80 is a well-known port
associated with Web servers

Using Ports to Identify Services

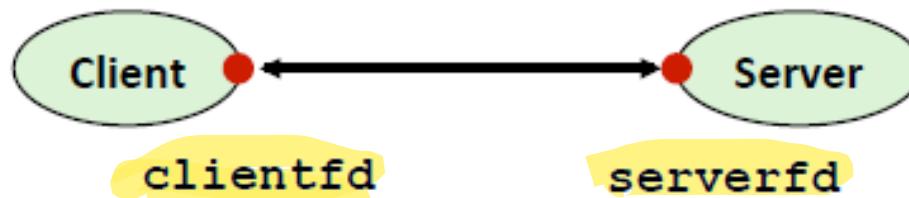


Socket Interface

- Set of system-level functions used in conjunction with Unix I/O to build network applications.
- Created in the early 80's as part of the original Berkeley distribution of Unix that contained an early version of the Internet protocols.
- Available on all modern systems
 - Unix variants, Windows, OS X, IOS, Android

Sockets

- What is a socket?
 - To the kernel, a socket is an endpoint of communication
 - To an application, a socket is a file descriptor that lets the application read/write from/to the network
 - Remember: All Unix I/O devices, including networks, are modeled as files
- Clients and servers communicate with each other by reading from and writing to socket descriptors

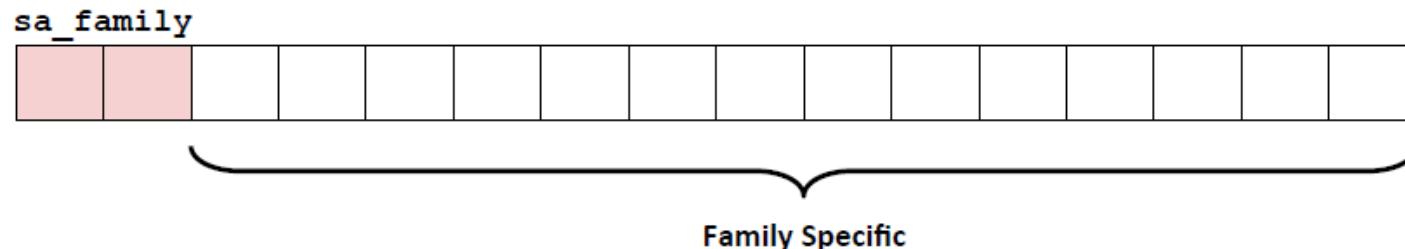


- The main distinction between regular file I/O and socket I/O is how the application “opens” the socket descriptors

Socket Address Structures

- Generic socket address:
 - For address arguments to connect, bind, and accept
 - Necessary only because C did not have generic (`void *`) pointers when the sockets interface was designed
 - For casting convenience, we adopt the Stevens convention:

```
struct sockaddr {  
    uint16_t sa_family; /* Protocol family */  
    char sa_data[14]; /* Address data. */  
};
```



Socket Address Structures

- Internet-specific socket address IPv4: Must cast (struct sockaddr_in *) to (struct sockaddr *) for functions that take socket address arguments.

used while binding at server side

```
struct in_addr {          It store IP address
    uint32_t s_addr; /* network byte order (big-
endian) */

};

struct sockaddr_in {        family is always AF_INET for IPv4
    uint16_t sin_family; /* Protocol family (always AF_INET) */
    uint16_t sin_port; /* Port num in network byte order */
    struct in_addr sin_addr; /* IP addr in network byte order */
    unsigned char sin_zero[8]; /* Pad to sizeof(struct sockaddr)
*/}

};
```

Internet-specific socket address IPv6

```
struct in6_addr {
    unsigned char s6_addr[16]; /* IPv6 address */
};

struct sockaddr_in6 {
    sa_family_t sin6_family; /* AF_INET6 */
    in_port_t sin6_port; /* port number */
    uint32_t sin6_flowinfo; /* IPv6 flow information */
    struct in6_addr sin6_addr; /* IPv6 address */
    uint32_t sin6_scope_id; /* Scope ID (new in 2.4) */
};
```

Get IP address for a given hostname using hostent structure

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
struct hostent {
    char *h_name;          /* official name of host */
    char **h_aliases;      /* alias list */
    int   h_addrtype;      /* host address type */
    int   h_length;         /* length of address */
    char **h_addr_list;    /* list of addresses */
}
#define h_addr h_addr_list[0] /* for backward compatibility */
```

gethostbyname() and inet_ntoa()

It returns hostent structure

\$ nslookup www.google.com → returns two addresses first IPv4 and second IPv6

Name: www.google.com

Address: 172.217.160.132

Name: www.google.com

Address: 2404:6800:4007:80a::2004

These functions works only for IPv4

struct hostent *gethostbyname(const char *name); → returns ptr to hostent structure given hostname

char *inet_ntoa(struct in_addr in); → convert in_addr network byte order into string

[NetworkProgramming\gethostbyname_example.c](#)

\$./gethostbyname_example.out www.google.com

Hostname: www.google.com

IP Address 1: 172.217.160.132

gethostbyname2() and inet_ntop()

These functions works for IPv4 or IPv6 but not both

struct hostent *gethostbyname2(const char *name, int af); → returns ptr to hostent structure given hostname and protocol family (i.e. af = either AF_INET or AF_INET6)

const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);
→ convert src network byte order into string pointed by dst

[NetworkProgramming\gethostbyname2_example.c](#)

```
$ ./gethostbyname2_example.out www.google.com
```

Hostname: www.google.com

IP Address 1: 2404:6800:4009:80a::2004

Host and Service Conversion: getaddrinfo()

Works for both IPv4 and IPv6 simultaneously

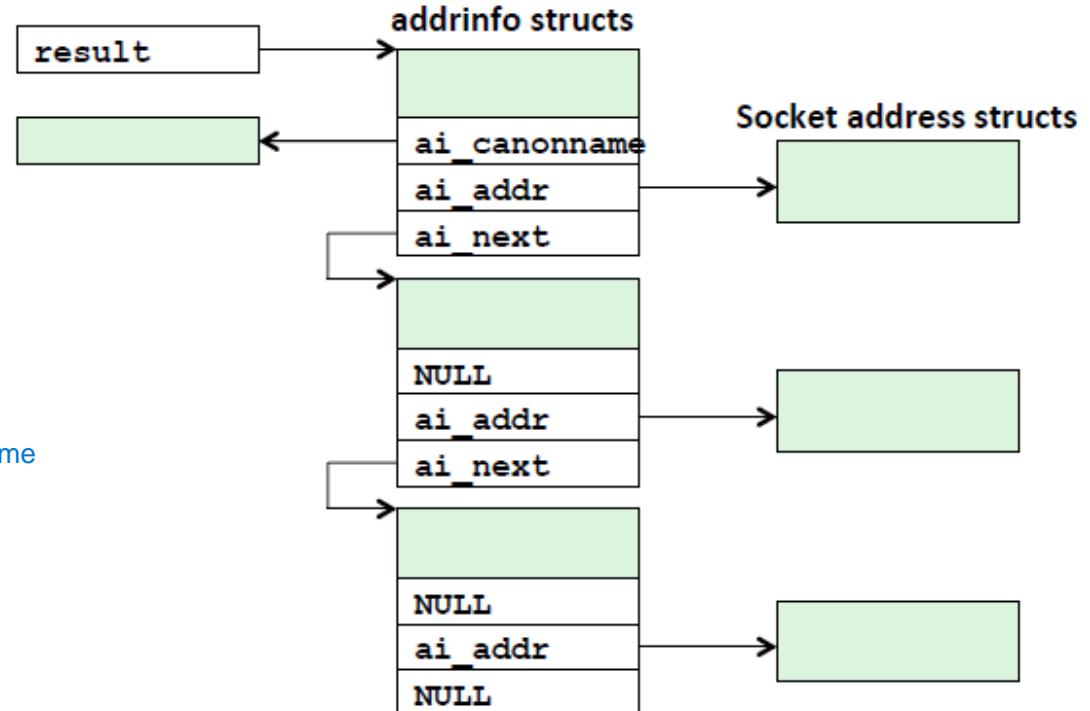
- Given host and service, `getaddrinfo` returns **result** that points to a linked list of **addrinfo** structs, each of which points to a corresponding socket address struct, and which contains arguments for the sockets interface functions.

It works same as `gethostname` but it just helps us to filter whichever type of IP we need

```
int getaddrinfo(const char *host, /* Hostname or address */  
                const char *service, /* Port or service name */  
                const struct addrinfo *hints,/* Input parameters  
                (Filtering) */  
                struct addrinfo **result); /* Output linked list */  
  
void freeaddrinfo(struct addrinfo *result); /* Free linked list */  
const char *gai_strerror(int errcode); /* Return error msg from error code */
```

Linked List returned by getaddrinfo()

- **getaddrinfo** is the modern way to convert string representations of hostnames, host addresses, ports, and service names to socket address structures.
 - Replaces obsolete `gethostbyname`
- Advantages:
 - Reentrant (can be safely used by threaded programs). as we can have race condition if we use `gethostbyname`
 - Allows us to write portable protocol-independent code (works with IPv4 and IPv6 addresses)
- Disadvantages
 - Somewhat complex
 - Fortunately, a small number of usage patterns suffice in most cases.



addrinfo structure

```
struct addrinfo {  
    int ai_flags; /* Hints argument flags (AI_PASSIVE – used in server for passive TCP connection, AI_ADDRCONFIG – used so that IPv4 or IPv6 any type of addresses can be used, AI_NUMERICSERV – used when providing numeric value of port number*)/  
    int ai_family; /* First arg to socket function (AF_INET or AF_INET6 or AF_UNSPEC) */  
    int ai_socktype; /* Second arg to socket function (SOCK_STREAM or SOCK_DGRAM or 0 means ANY)*/  
    int ai_protocol; /* Third arg to socket function (0 means ANY – generally only 1 protocol per family) */  
    char *ai_canonname; /* Canonical host name */  
    size_t ai_addrlen; /* Size of ai_addr struct */  
    struct sockaddr *ai_addr; /* Ptr to socket address structure */  
    struct addrinfo *ai_next; /* Ptr to next item in linked list */  
};
```

used for filtering in getaddrinfo function at server side

Used for TCP

Used for UDP

- Each addrinfo struct returned by getaddrinfo contains arguments that can be passed directly to socket function.
- Also points to a socket address struct that can be passed directly to connect and bind functions.

Host and Service Conversion: getnameinfo()

```
int getnameinfo(  
    const struct sockaddr *sa, socklen_t salen, /*  
     In: socket addr */  
    char *host, size_t hostlen, /* Out: host */  
    char *serv, size_t servlen, /* Out: service */  
    int flags); /* optional flags */  
  
flags = NI_NUMERICHOST | NI_NUMERICSERV;  
/* Display address string instead of domain  
 name and port number instead of service  
 name */
```

- **getnameinfo** displays a socket address to the corresponding host (name or IP) and service (service or port).
 - Replaces obsolete **gethostbyaddr** and **getservbyport** funcs.
 - Reentrant and protocol independent.

getaddrinfo() example

[NetworkProgramming\hostinfo.c](#)

\$./hostinfo.out [www.daiict.ac.in](#)

220.226.182.128 → Returned IPv4 IP address

\$./hostinfo.out localhost

127.0.0.1

\$./hostinfo.out [www.twitter.com](#)

104.244.42.65

104.244.42.1

\$./hostinfo.out [www.google.com](#)

172.217.163.100

2404:6800:4007:809::2004

\$./hostinfo.out [www.facebook.com](#)

157.240.13.35 → Returned IPv4 IP address

2a03:2880:f10c:83:face:b00c:0:25de → Returned IPv6 IP address

If we disable line

#define IPv4 1, it will provide IPv4 as well as IPv6 addresses

\$./hostinfo.out [www.google.com](#)

172.217.163.100

2404:6800:4007:811::2004

If we enable line

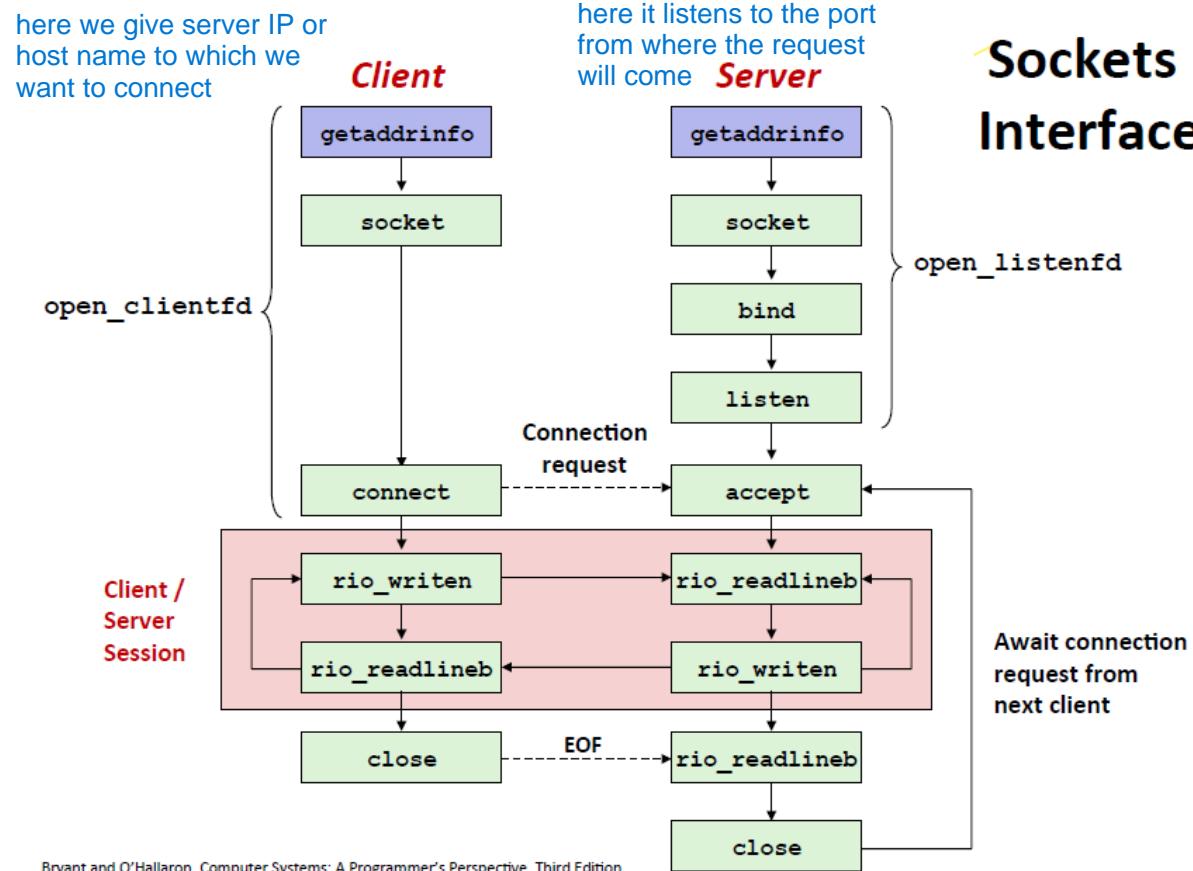
#define IPv4 1, it will provide IPv4 addresses only

\$./hostinfo.out [www.google.com](#)

172.217.163.100

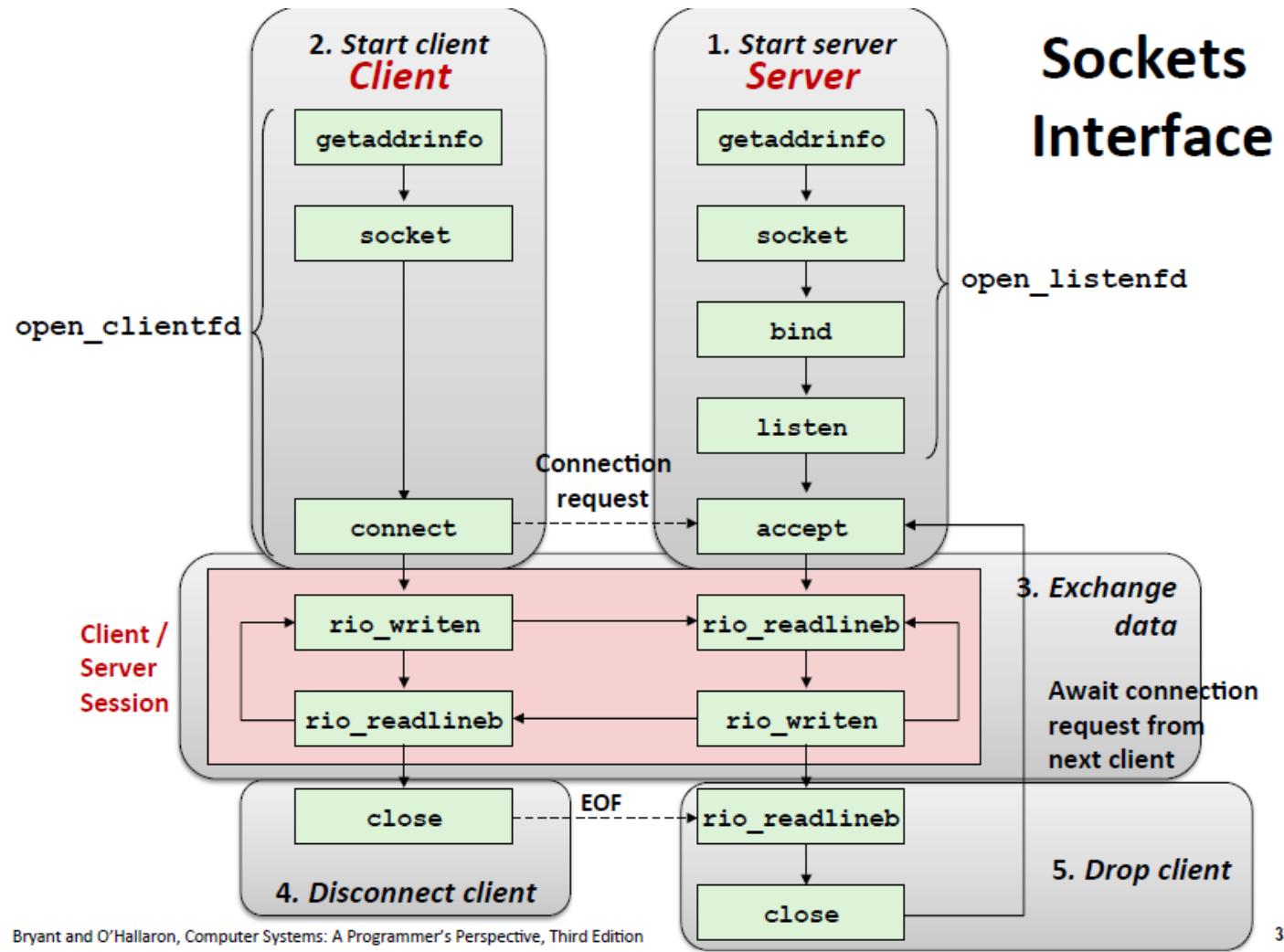
Client Server Communication using Socket Interface

Host and Service Conversion: getaddrinfo()



- **Clients:** Using server IP/hostname and service/port calls `getaddrinfo`. It walks through the returned list of server socket addresses, trying each socket address in turn, until the calls to `socket` and `connect` succeed.
- **Server:** Using service/port calls `getaddrinfo`. It walks through the returned list of socket addresses (possible to have different IP) until calls to `socket` and `bind` succeed.

Socket Interface

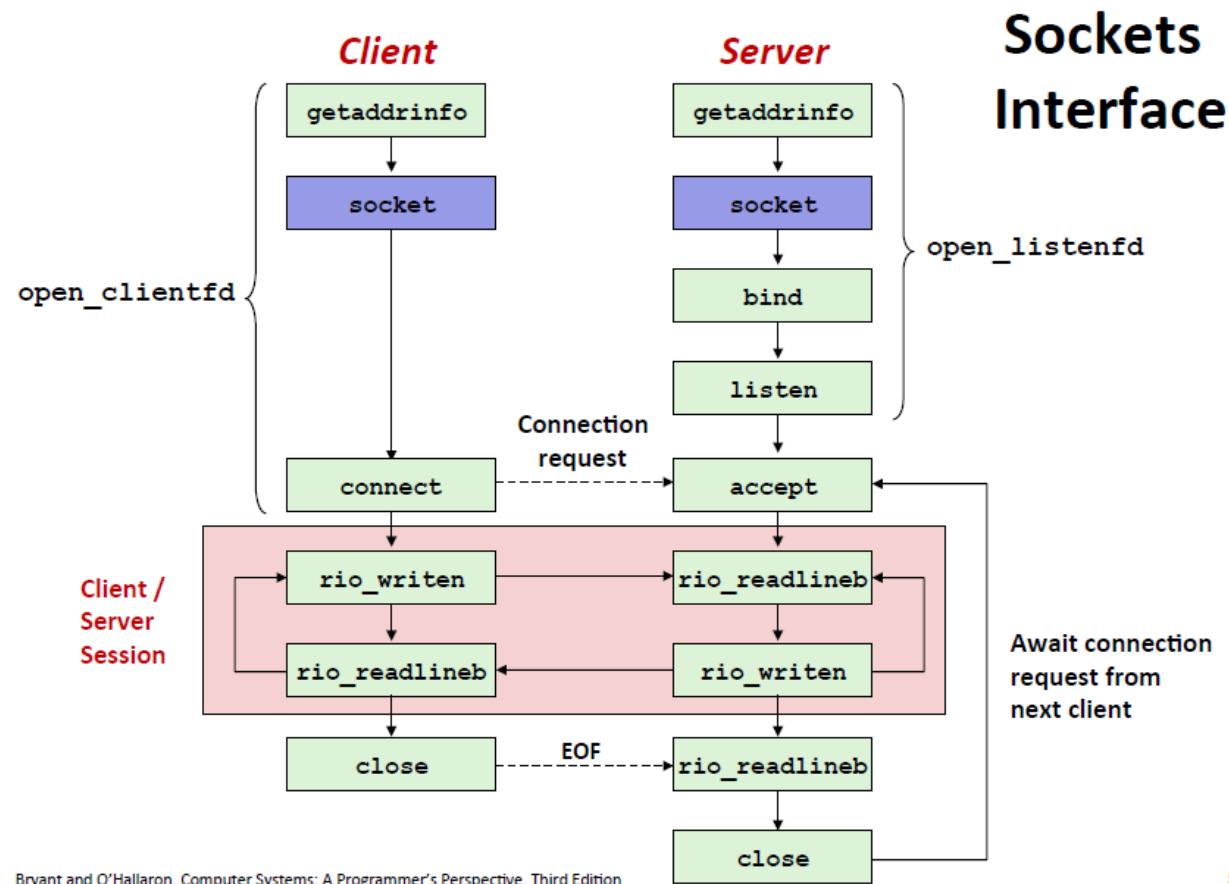


Sockets Interface

open_clientfd → user function to start client
open_listenfd → user function to start server

rio_written = write(fd, buf, numofbytes)
rio_readlineb = read(fd, buf, numofbytes)

Socket Interface: socket()



Socket Interface: socket()

- Clients and servers both use the socket function to create a socket descriptor:

int socket(int domain, int type, int protocol)

domain : indicates protocol family (*same as ai_family from struct addrinfo*)

type : socket type to use with protocol family (*same as ai_socktype from struct addrinfo*)

Protocol : protocol to be used from protocol family with a specific socket type. Generally there is only one protocol so we pass 0 (*same as ai_protocol from struct addrinfo*)

- Example:

int sockfd = socket(AF_INET, SOCK_STREAM, 0);

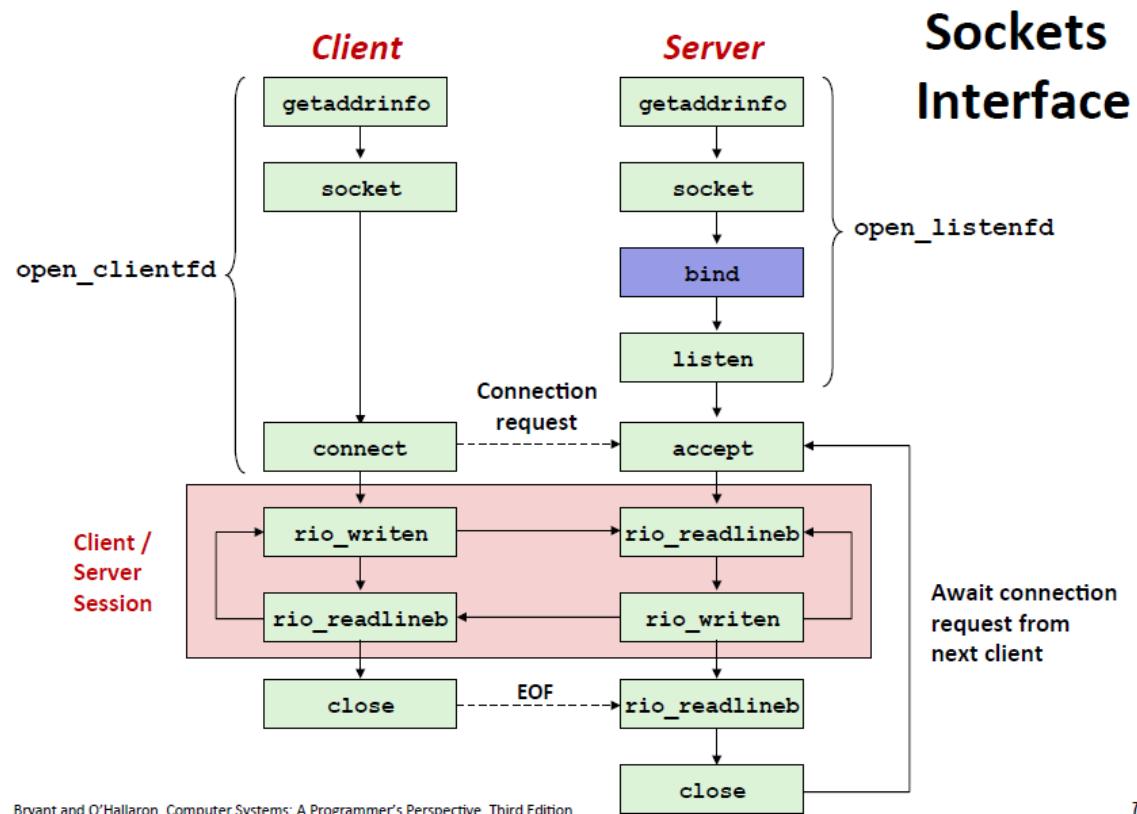
sockfd will be same as clientfd for Client and listenfd for Server

- Protocol specific! Best practice is to use getaddrinfo to generate the parameters automatically, so that code is protocol independent.

Socket Domains and Types

Domains		Types	
Name	Purpose	Name	Purpose
AF_UNIX, AF_LOCAL	Local communication	SOCK_STREAM	Provides sequenced, reliable, two-way, connection-based byte streams. An out-of-band data transmission mechanism may be supported
AF_INET	IPv4 Internet protocols	SOCK_DGRAM	Supports datagrams (connectionless, unreliable messages of a fixed maximum length).
AF_INET6	IPv6 Internet protocols	SOCK_SEQPACKET	Provides a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; a consumer is required to read an entire packet with each input system call.
AF_IPX	IPX - Novell protocols	SOCK_RAW	Provides raw network protocol access
AF_NETLINK	Kernel user interface device	SOCK_RDM	Provides a reliable datagram layer that does not guarantee ordering
AF_X25	ITU-T X.25 / ISO-8208 protocol	SOCK_PACKET	Obsolete and should not be used in new programs
AF_AX25	Amateur radio AX.25 protocol		
AF_APPLETALK	AppleTalk		
AF_PACKET	Low level packet interface		

Socket Interface: bind()



Socket Interface: bind()

- A server uses bind to ask the kernel to associate the server's socket address with a socket descriptor:

```
int bind(int listenfd, struct sockaddr *srv_addr, socklen_t  
srv_addrlen);
```

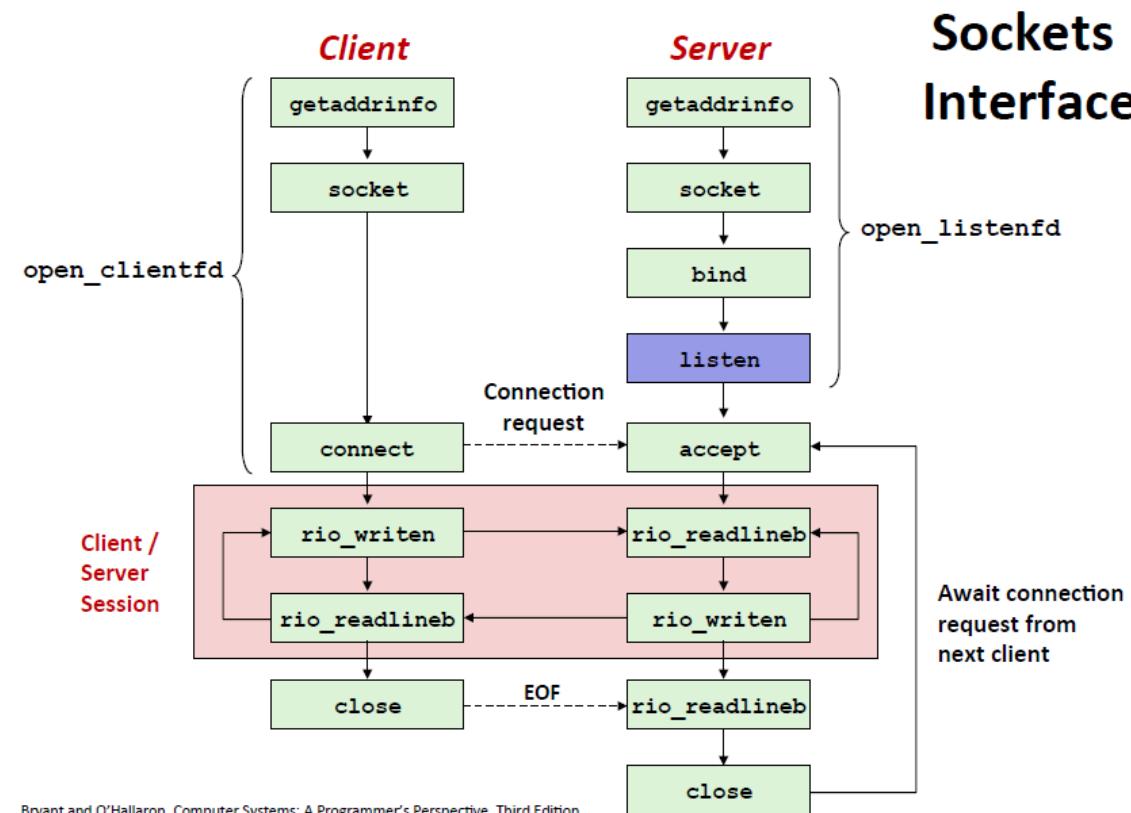
It binds the socketfd with socket address

- The process can read bytes that arrive on the connection whose endpoint is srv_addr by reading from descriptor listenfd.
- Similarly, writes to listenfd are transferred along connection whose endpoint is srv_addr.
- Best practice is to use getaddrinfo to supply the arguments srv_addr and addrlen

struct sockaddr: Casted Address for any of the following types

Name	Purpose	sockaddr variants
AF_UNIX, AF_LOCAL	Local communication	sockaddr_un
AF_INET	IPv4 Internet protocols	sockaddr_in
AF_INET6	IPv6 Internet protocols	sockaddr_in6
AF_IPX	IPX - Novell protocols	sockaddr_ipx
AF_NETLINK	Kernel user interface device	sockaddr_nl
AF_X25	ITU-T X.25 / ISO-8208 protocol	sockaddr_x25
AF_AX25	Amateur radio AX.25 protocol	sockaddr_ax25
AF_APPLETALK	AppleTalk	sockaddr_atalk
AF_PACKET	Low level packet interface	sockaddr_ll

Socket Interface: `listen()`



Socket Interface: listen()

- By default, kernel assumes that descriptor from socket function is an **active socket** that will be on the client end of the connection.
- A server calls the listen function to tell the kernel that a descriptor will be used by a server rather than a client:

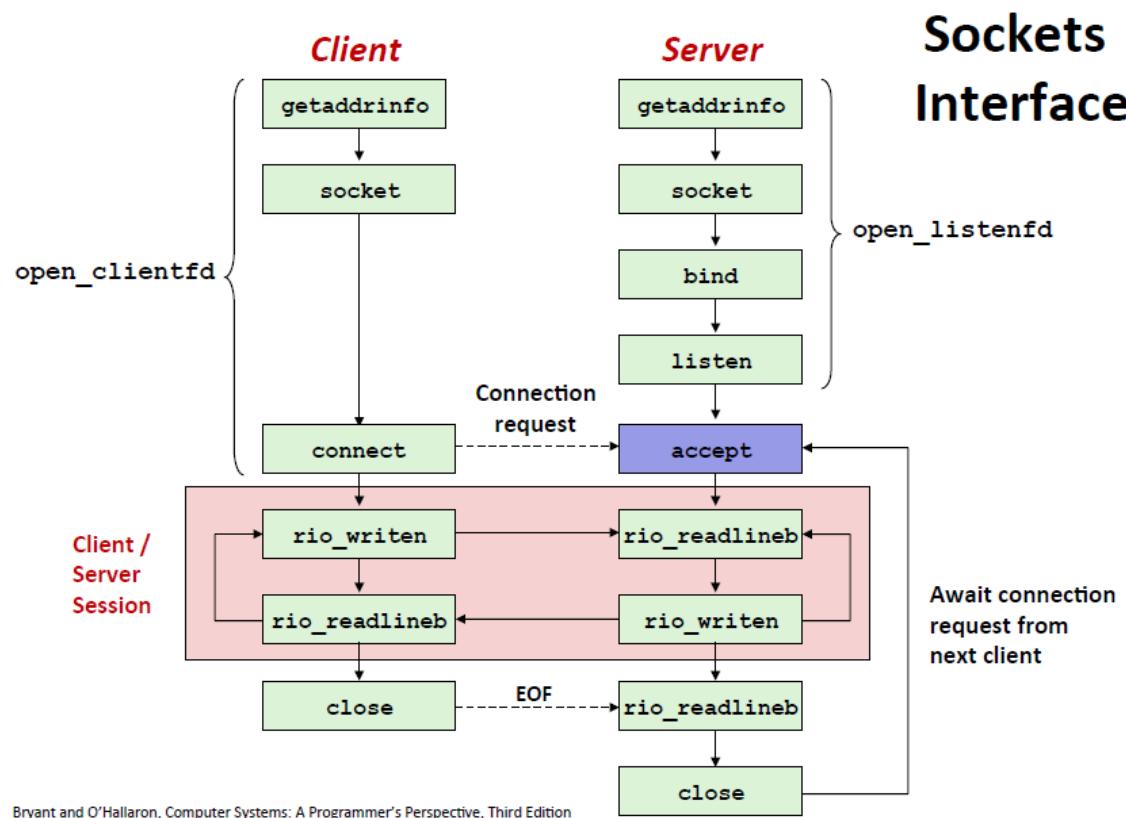
backlog : means at a time how many connection can be established

int listen(int listenfd, int backlog); used to make socket from active to passive as we just need to listen to the client connection and not to initiate new connection

- Converts listenfd from an active socket to a **listening socket** that can accept connection requests from clients.
- backlog is the pending number of connection requests that the kernel should queue up before starting to refuse requests.

Socket Interface: accept()

It is the blocking call on server side



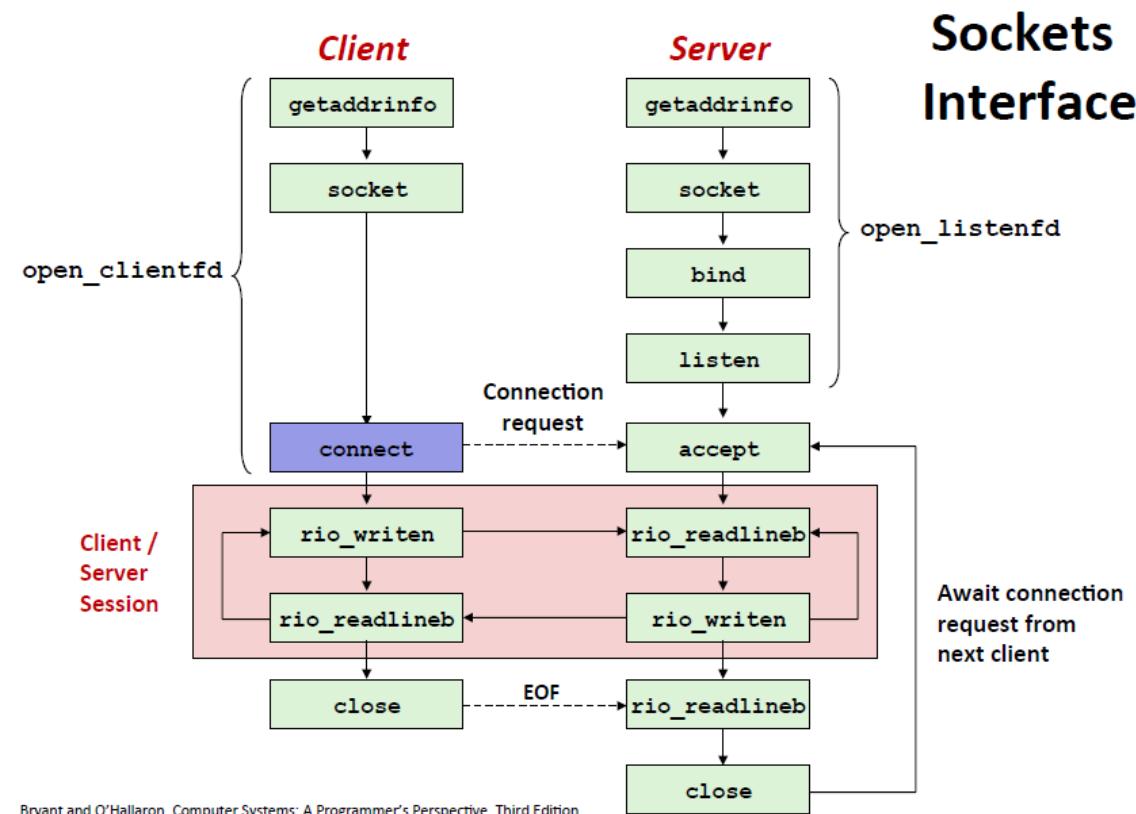
Socket Interface: accept()

- Servers wait for connection requests from clients by calling accept:

int accept(int listenfd, struct sockaddr *clnt_addr, int *clnt_addrlen);

- Waits for connection request to arrive on the connection bound to listenfd, then fills in client's socket address in clnt_addr and size of the socket address in clnt_addrlen.
- Returns a new **connection descriptor connfd** that is the bound to clnt_addr which is used to communicate with the client via Unix I/O routines. So for every new accepted connection, a new connfd is created for an accepted client connection to be used for communication with that client only. (so that listenfd can be freed to be used for accepting new connections)

Socket Interface: connect()



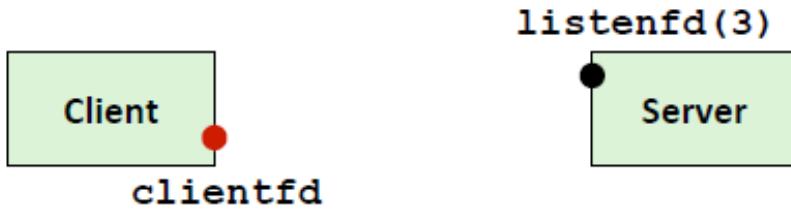
Socket Interface: connect()

- A client establishes a connection with a server by calling connect:

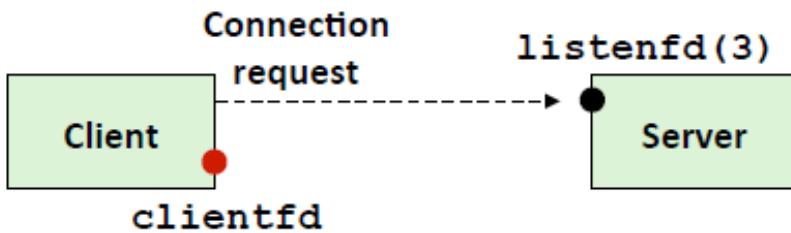
```
int connect(int clientfd, struct sockaddr *srv_addr, socklen_t srv_addrlen);
```

- Attempts to establish a connection with server at socket address `srv_addr`
 - If successful, then `clientfd` is now ready for reading and writing.
 - Resulting connection is characterized by socket pair (`x:y,addr.sin_addr:addr.sin_port`)
 - `x` is client address
 - `y` is ephemeral port that uniquely identifies client process on client host
- Best practice is to use `getaddrinfo` to supply the arguments `srv_addr` and `srv_addrlen`.

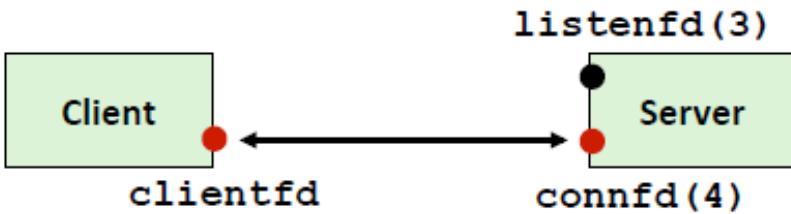
accept Illustrated



1. *Server blocks in accept, waiting for connection request on listening descriptor listenfd*



2. *Client makes connection request by calling and blocking in connect*

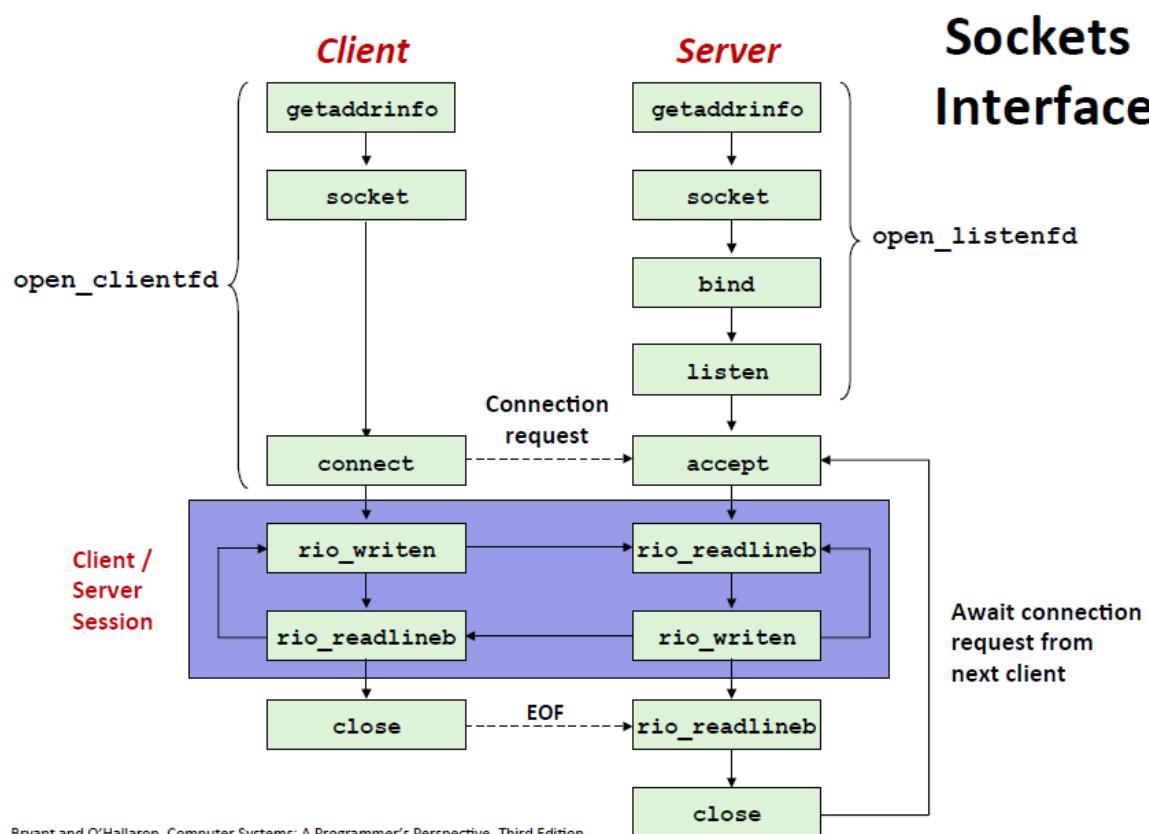


3. *Server returns connfd from accept. Client returns from connect. Connection is now established between clientfd and connfd*

Connected vs Listening Descriptors

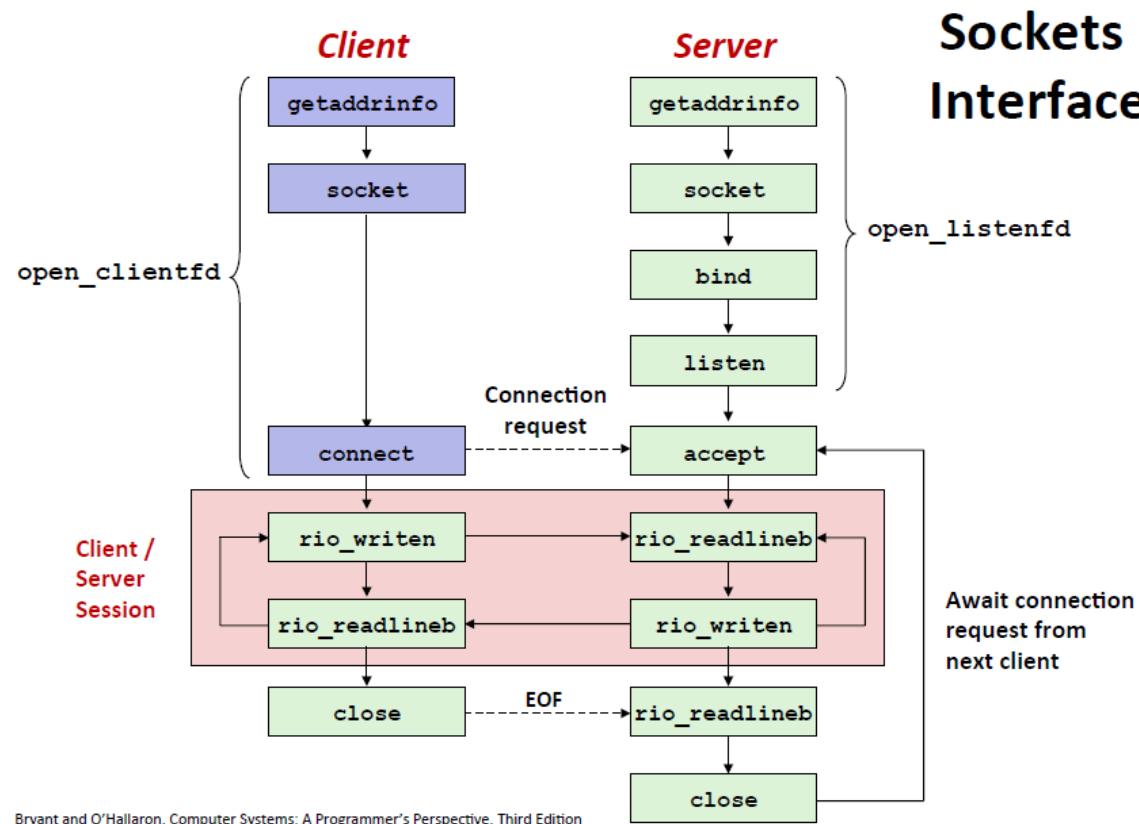
- Listening descriptor (listenfd)
 - End point for client connection requests
 - Created once and exists for lifetime of the server
- Connected descriptor (connfd)
 - End point of the connection between client and server
 - A new descriptor is created each time the server accepts a connection request from a client
 - Exists only as long as it takes to service client
- Why the distinction?
 - Allows for concurrent servers that can communicate over many client connections simultaneously
 - E.g., Each time we receive a new request, we fork a child to handle the request

Client-Server Session



- Server side **listenfd** is bound to server socket address for listening to clients connection requests.
- Client side **clientfd** is bound to server socket address to send and receive data to server.
- Server side **connfd** is bound to client socket address to send and receive data to client.

Socket Helper Function: open_clientfd



Socket Helper Function: open_clientfd

```
int open_clientfd(char *hostname, char *port)
{
    int clientfd;    struct addrinfo hints, *listp, *p;
    /* Get a list of potential server addresses */
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_socktype = SOCK_STREAM; /* Open a connection */
    hints.ai_flags = AI_NUMERICSERV; /* ... using a numeric port arg. */
    hints.ai_flags |= AI_ADDRCONFIG; /* Recommended for connections
where we get IPv4 or IPv6 addresses */
    getaddrinfo(hostname, port, &hints, &listp);
```

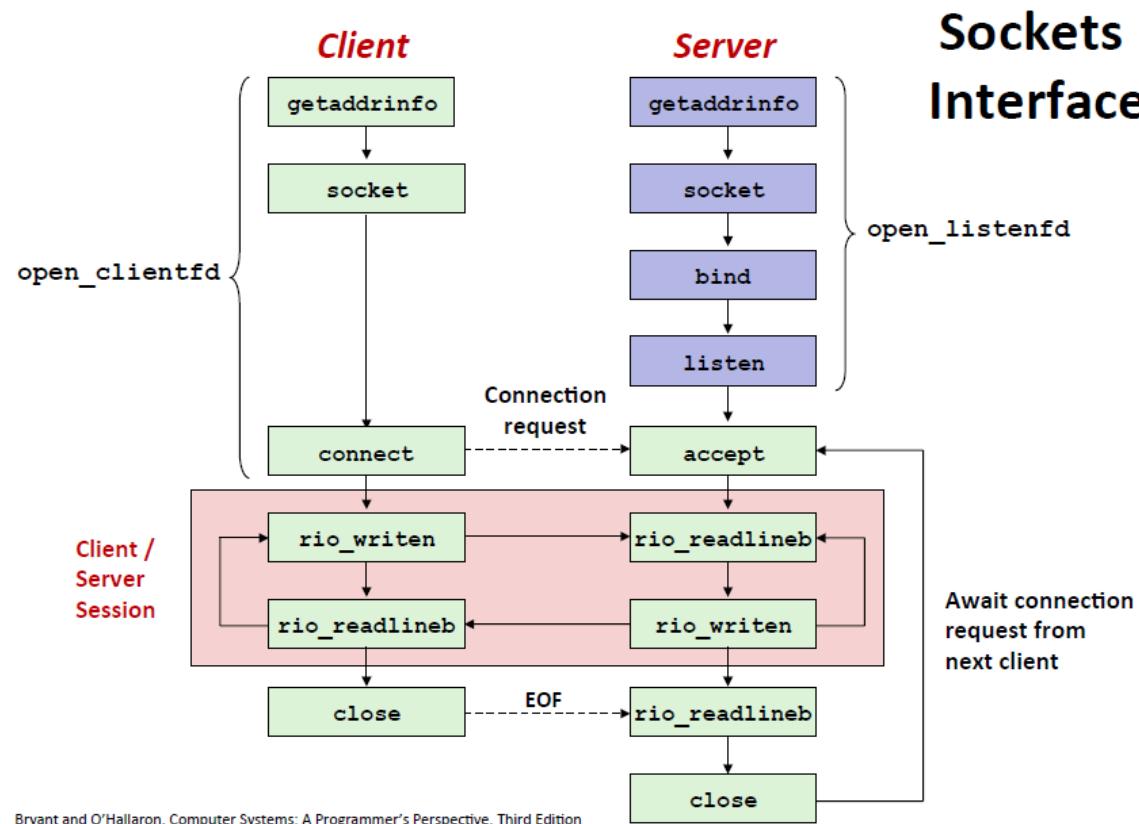
Socket Helper Function: open_clientfd

```
/* Walk the list for one that we can successfully connect to */
for (p = listp; p; p = p->ai_next) {
    /* Create a socket descriptor */
    if ((clientfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) < 0)
        continue; /* Socket failed, try the next */
    /* Connect to the server */
    if (connect(clientfd, p->ai_addr, p->ai_addrlen) != -1)
        break; /* Success */
    close(clientfd); /* Connect failed, try another */
} /* end for */
```

Socket Helper Function: open_clientfd

```
/* Clean up */  
freeaddrinfo(listp);  
if (!p) /* All connects failed */  
    return -1;  
else /* The last connect succeeded */  
    return clientfd;  
} /* end open_clientfd */
```

Socket Helper Function: open_listenfd



Socket Helper Function: open_listenfd

```
int open_listenfd(char *port) {
    struct addrinfo hints, *listp, *p;
    int listenfd, optval=1; /* Get a list of potential server addresses */
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_socktype = SOCK_STREAM; /* Accept connections */
    hints.ai_flags = AI_PASSIVE | AI_ADDRCONFIG; /* ... on any IP address
AI_PASSIVE - used for server for TCP passive connection, AI_ADDRCONFIG -
to use both IPv4 and IPv6 addresses */
    hints.ai_flags |= AI_NUMERICSERV; /* ... using port number */
    getaddrinfo(NULL, port, &hints, &listp);
```

Socket Helper Function: open_listenfd

```
/* Walk the list for one that we can bind to */
for (p = listp; p; p = p->ai_next) {
    /* Create a socket descriptor */
    if ((listenfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) < 0)
        continue; /* Socket failed, try the next */

    /* Eliminates "Address already in use" error from bind in case if process was killed during previous
       execution and port was not freed. SOL_SOCKET =Socket API, set SO_REUSEADDR =optval(1)*/
    setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, (const void *)&optval , sizeof(int));
    /* Bind the descriptor to the address */
    if (bind(listenfd, p->ai_addr, p->ai_addrlen) == 0)
        break; /* Success */
    close(listenfd); /* Bind failed, try the next */
} /* end for */
```

Socket Helper Function: open_listenfd

```
/* Clean up */
freeaddrinfo(listp);
if (!p) /* No address worked */
    return -1;
/* Make it a listening socket ready to accept connection requests */
if (listen(listenfd, LISTENQ) < 0) {
    close(listenfd);
    return -1;
} /* end if */
return listenfd;
} /* end open_listenfd */
```

Test Your Own Echo Client Using Default Echo Server on Port 7 (without using your own server)

- First, make sure echo service is running on port 7:

```
$ cat /etc/services | grep 7/tcp  
echo      7/tcp
```

If not you will have to install inetd service to have echo service on port 7

Our Echo Client Implementation:
[NetworkProgramming\echoclient.c](#)

```
$ ./echoclient.out localhost 7  
host:127.0.0.1, service:7  
message from our client to echo  
server on port 7  
message from our client to echo  
server on port 7 → echoed  
another test  
another test → echoed  
^C
```

Test Your Own Echo Server Using Telnet (without using your own client)

```
$ telnet 10.0.0.6 15020
Trying 10.0.0.6...
Connected to 10.0.0.6.
Escape character is '^].
this is a test from telnet client
this is a test from telnet client
howdy from telnet client
howdy from telnet client
^]
telnet> Connection closed.
$
```

Our Echo Server Implementation:
[NetworkProgramming\echoserver.c](#)

```
$ ./server.out 15020
Waiting for a new Client to connect
Connected to (10.0.0.6, 56166)
Start Communication with Client
server received 35 bytes
server received message : this is a test from telnet client

server received 26 bytes
server received message : howdy from telnet client

End Communication with Client
Waiting for a new Client to connect
```

Test Your Own Echo Client and Echo Server

```
$ ./client.out faculty-Optiplex-3040 15010  
this is a test from client 1  
this is a test from client 1  
howdy from client 1  
howdy from client 1
```

```
$ ./client.out faculty-Optiplex-3040 15010  
this is a test from client 2  
this is a test from client 2  
hello from client 2  
hello from client 2  
$
```

\$./server.out 15010	Waiting for a new Client to connect	Continue →
	Connected to (localhost, 45752)	Waiting for a new Client to connect
	Start Communication with Client	Connected to (localhost, 45754)
	server received 29 bytes	Start Communication with Client
	server received message : this is a test from client 1	server received message : this is a test from client 2
	server received 20 bytes	server received 21 bytes
	server received message : howdy from client 1	server received message : hellow from client 2
	server received 1 bytes	server received 1 bytes
	server received message :	server received message :
	End Communication with Client	End Communication with Client
	→ Continue	Waiting for a new Client to connect

Test Servers Using Telnet

- The telnet program is invaluable for testing servers that transmit ASCII strings over Internet connections
 - Our simple echo server
 - Web servers
 - Mail servers
- Usage:
 - \$ telnet <host> <portnumber>
 - Creates a connection with a server running on <host> and listening on port <portnumber>

Echo Server Problem

- Echo Server is able to handle only 1 client connection at a time because the main thread goes in loop unless client end the connection
- How to fix it?

Solution of Echo Server Problem

```
while (1) {  
    printf("Waiting for a new Client to connect\n");  
    clientlen = sizeof(struct sockaddr_storage); /* Important! */  
    connfd = accept(listenfd, (struct sockaddr *)&clientaddr, &clientlen);  
    if (fork() == 0)      /* child will handle a new client everytime server accepts a connection from client */  
    {  
        getnameinfo((struct sockaddr *) &clientaddr, clientlen, client_hostname, MAXLINE, client_port, MAXLINE, 0);  
        printf("Connected to (%s, %s)\n", client_hostname, client_port);  
        printf("Start Communication with Client\n");  
        echo(connfd);  
        printf("End Communication with Client\n");  
        close(connfd);  
    }  
}
```

Solution of Echo Server Problem

- Would it be better to use multiple threads or multiple processes to handle clients?
- What are the challenges with each scheme?

Systems
Software/Programming
Device Driver

<https://lwn.net/Kernel/LDD3/>

Linux Architecture

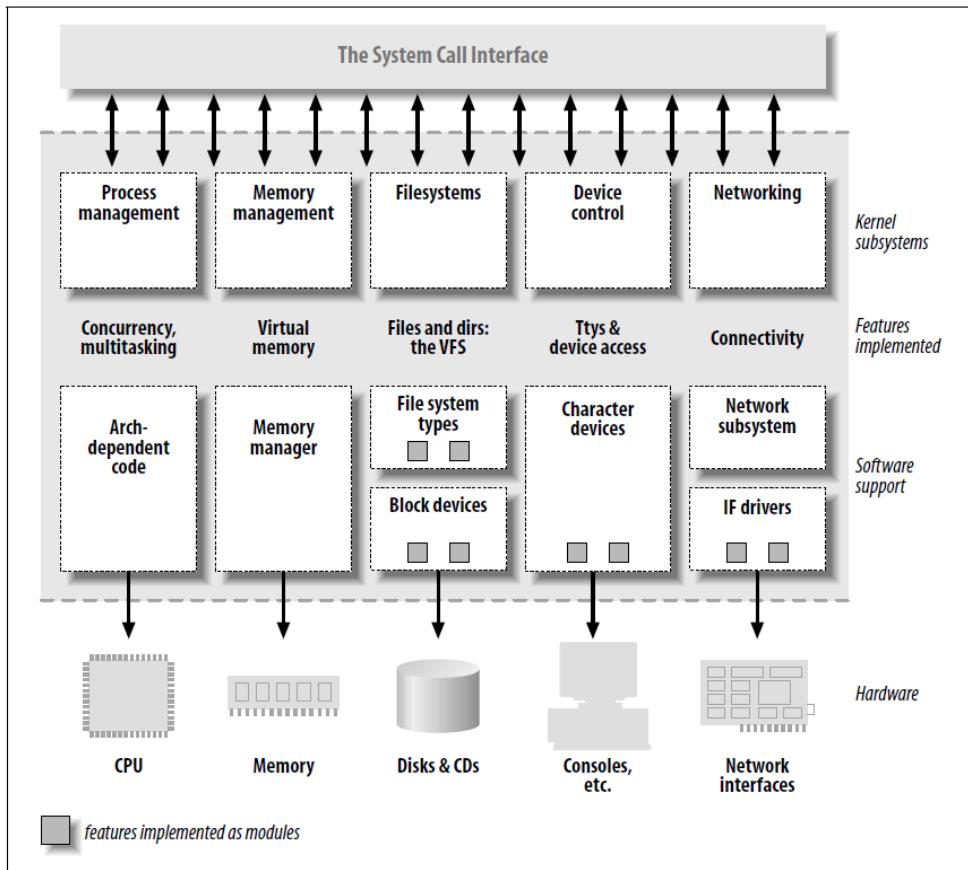


Figure 1-1. A split view of the kernel

- Linux Kernel has several modules:
 - Process Management (studied)
 - Memory Management – create and manage of virtual address space
 - Filesystems – Manages files and directories stored in physical disk by dividing it into logical components.
 - Device Control/Driver – Software designed to communicate and operate specific hardware so that kernel use the hardware without having to know the details.
 - Networking – Protocol implementation

Device Driver Types

- Character Device
 - Read/write character at a time and implements open, read, write, close system calls
 - Character devices are synchronous and only one process can use it at a time
 - Examples: console, keyboard, mouse etc
- Block Device
 - Read/write block of data at a time
 - Writing is done asynchronously so multiple process can write at the same time
 - Examples: CDROM, HDD, USB etc
- Network Device
 - Sends or receives data in form of packets.
 - Example: Hardware device: Ethernet card, Software device: Loopback adapter

Device Driver Module Information

- License:
 - Specified using MODULE_LICENSE("license") macro
 - <license> : "GPL", "GPL v2", "Dual BSD/GPL", "Dual MIT/GPL", "Proprietary" etc
- Author:
 - Specified using MODULE_AUTHOR("Author") macro
- Module Description:
 - Specified using MODULE_DESCRIPTION("My Driver") macro
- Module Version:
 - Format [<epoch>:]<version>[-<extra-version>]
 - Specified using MODULE_VERSION("2:1.0") macro

Constructor and Exit functions

- Init function: `module_init(* func())` macro
 - Called when driver module is loaded e.g. using insmod
- Exit function: `module_exit(* func())` macro
 - Called when driver module is unloaded e.g. using rmmod

```
static int __init mydriver_init(void) /*  
Constructor */  
{  
    return 0;  
}  
module_init(mydriver_init);  
  
void __exit mydriver_exit(void)  
{  
}  
module_exit(mydriver_exit);
```

printf()

- printf() kernel level function similar to printf() which is user level
- Except, printf() allow assigning loglevel/priority of messages
 - KERN_EMERG: emergency
 - KERN_ALERT: requiring immediate attention
 - KERN_CRIT: critical condition related to hardware or software
 - KERN_ERR: error reporting
 - KERN_WARNING: warning reporting
 - KERN_NOTICE: take a note
 - KERN_INFO: informational messages
 - KERN_DEBUG: debugging information

Simple Driver

- Source [DeviceDriver\mydriver1.c](#) and makefile
[DeviceDriver\mydriver1 makefile](#)
- Compile: \$sudo make
- List module: \$lsmod | grep mydriver1
- Load module: \$sudo insmod mydriver1.ko
- Unload module: \$ sudo rmmod mydriver1
- To see messages from printk(): \$dmesg

Passing Parameters to Driver Code

[DeviceDriver\mydriver2.c](#)

[DeviceDriver\mydriver2 makefile](#)

- module_param(name, type, perm): Initialize primitive type variable
 - creates the sub-directory under /sys/module
 - Type: bool, invbool, charp (char ptr), int, long, short, uint, ulong, ushort
 - Perm: S_I(W/R)(USR/GRP/OTH), can be combined with bit OR (|)
- module_param_array(name, type, num, perm): For arrays
- module_param_cb(): register callback function which is called when change in value is detected in parameter.
- Load module: \$ \$ sudo insmod mydriver2.ko value=13 name="DAIICT" arr_value=10,20,30,40
- Files for each parameter with its values is created in /sys/module/mydriver2/parameters/ → “cat name” or “cat cb_value”
- To change the value:
 - First login as root user by \$su
 - Change value : \$sudo bash –c ‘echo new_value > /sys/module/mydriver2/parameters/param_name’
 - e.g. sudo bash -c 'echo 20 > /sys/module/mydriver2/parameters/cb_value'
 - Check messages in system log using dmesg

All devices have Major and Minor Numbers

- Major number represents which driver will be used to manage a particular device. It allows multiple devices to have the same major number if they are managed by the same driver.
- Minor number represents a particular device that is managed by the driver identified using Major number.
- In the example below there are 4 tty devices represented as minor numbers 0, 10, 11, 12 but they are all managed by same driver number 4. Similarly 2 disk sda devices with minor numbers 0 and 1 are managed by driver major number 8.

```
$ ls -ltr /dev
crw--w---- 1 root    tty     4,  0 May 15 12:46 tty0
crw--w---- 1 root    tty     4, 10 May 15 12:46 tty10
crw--w---- 1 root    tty     4, 12 May 15 12:46 tty12
crw--w---- 1 root    tty     4, 11 May 15 12:46 tty11
brw-rw---- 1 root disk  8, 0 May 15 12:46 /dev/sda
brw-rw---- 1 root disk  8, 1 May 15 12:46 /dev/sda1
```

C stands for char device

B stands for block device

Application

Device File

Major and Minor Number

Device Driver

Hardware

Character Device Driver → Create Major and Minor Numbers [DeviceDriver\mydriver3.c](#)

[DeviceDriver\mydriver3 makefile](#)

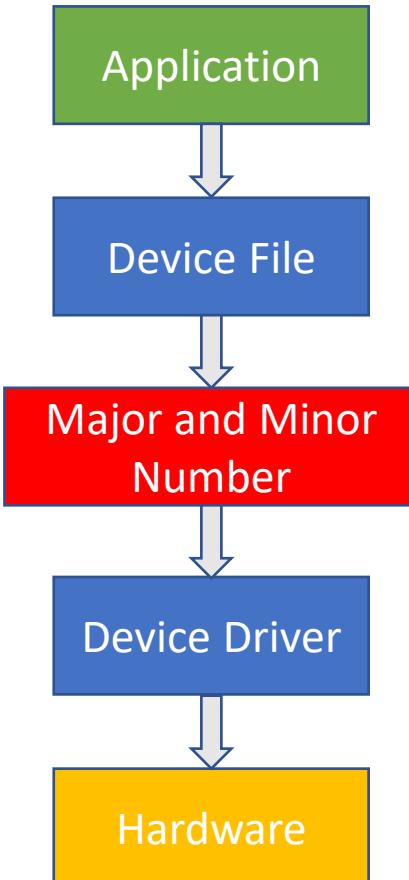
- dev_t struct from linux/types.h used to represent <major>:<minor>
- Create device's major and minor nos using macro MKDEV(int major, int minor) that will return dev_t structure.

```
dev_t dev = MKDEV(201, 0)
```

- Statically Register the Device → you provide value for major number, **can have conflict even if different drivers are used.**

```
int register_chrdev_region(dev_t first, unsigned int count, char *name);
```

- first → starting device number; dev_t dev = MKDEV(201, 0)
- count → number of devices requested i.e. number of minor numbers
- name → device name representing number range [first, first+count]



Character Device Driver → Create Major and Minor Numbers [DeviceDriver\mydriver3.c](#)

[DeviceDriver\mydriver3 makefile](#)

- Dynamically Register the device → Major number is allocated to you by kernel

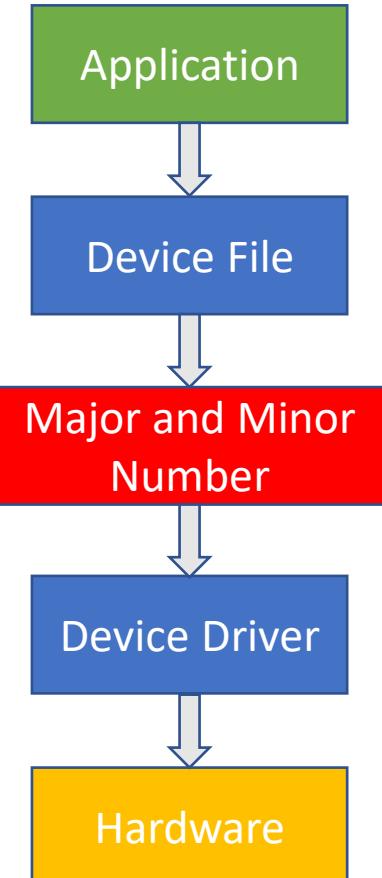
```
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor,  
unsigned int count, char *name);
```

- dev → major number is returned by kernel which is free to use
- Firstminor → you assign starting minor number
- Unregister the device

```
void unregister_chrdev_region(dev_t first, unsigned int count);
```

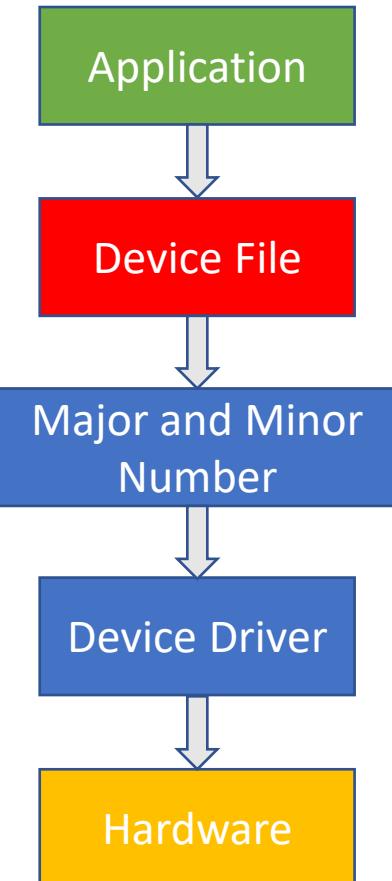
After driver is loaded using insmod, **check major number for your device in “cat /proc/devices”**

You will not find physical device in /dev directory yet since we have not created the actual device which will be done next.



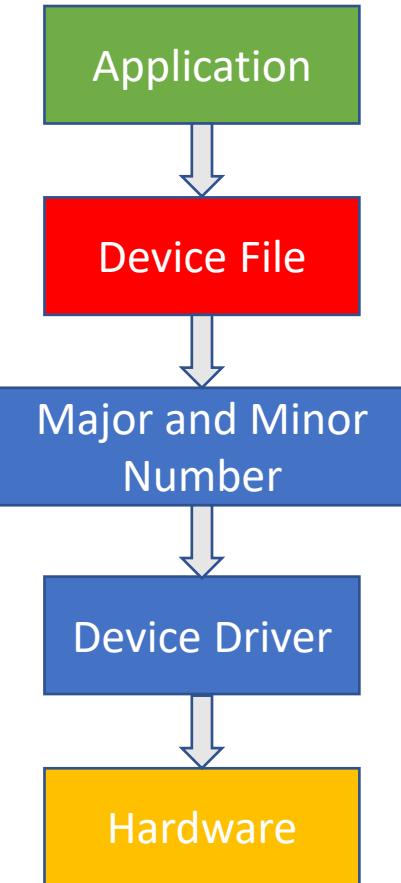
Character Device Driver → Create Device File

- All devices are treated as files and are stored in /dev directory in Unix-based OS.
- Create the device using shell command mknod:
 - `mknod -m <permissions> <name> <device type> <major> <minor>`
 - name – full path e.g. /dev/mydev
 - device type – c for char device, b for block device
 - Example: `$sudo mknod -m 666 /dev/mydev c 201 0`
 - Check device creation using → `ls -l /dev/ | grep <devname>`
- **Better approach is to create the device as part of the driver code itself.**



Character Device Driver → Create Device File

- Automatically (Programmatically) [DeviceDriver\mydriver4.c](#) [DeviceDriver\mydriver4 makefile](#)
 - header file linux/device.h and linux/kdev_t.h
 - Create structure class for our driver under /sys/class:
 - struct class * class_create (struct module *owner, const char *name);
 - void class_destroy (struct class * cls); // destroy once done
 - Create device:
 - struct device *device_create (struct class * cls, struct device *parent, dev_t dev, const char *fmt, ...);
 - void device_destroy (struct class * class, dev_t devt); // destroy when done
 - Check device creation using → ls -l /dev/ | grep <devname>



cdev Structure linux/cdev.h

- cdev structure is used by linux kernel to represent char device. **Inode of this device will have pointer to cdev structure**

```
struct cdev {  
    struct kobject kobj;  
    struct module *owner;  
    const struct file_operations *ops;  
    struct list_head list;  
    dev_t dev;  
    unsigned int count;  
};
```

- owner = THIS_MODULE defined in linux/module.h
- ops = list of function pointers which implements the operations for this device
- Kobj = information about driver kernel object
- Dev = your device

inode structure has pointer to cdev structure.
In the virtual box installation at /usr/src/linux-headers-5.8.0-48-generic/include/linux/fs.h

```
struct inode {
    . . .
    umode_t i_mode;
    unsigned short i_opflags;
    kuid_t     i_uid;
    kgid_t     i_gid;
    unsigned int i_flags;
    . . .
};

union {
    struct pipe_inode_info *i_pipe;
    struct block_device   *i_bdev;
    struct cdev           *i_cdev;
    char                  *i_link;
    unsigned               i_dir_seq;
};

};
```

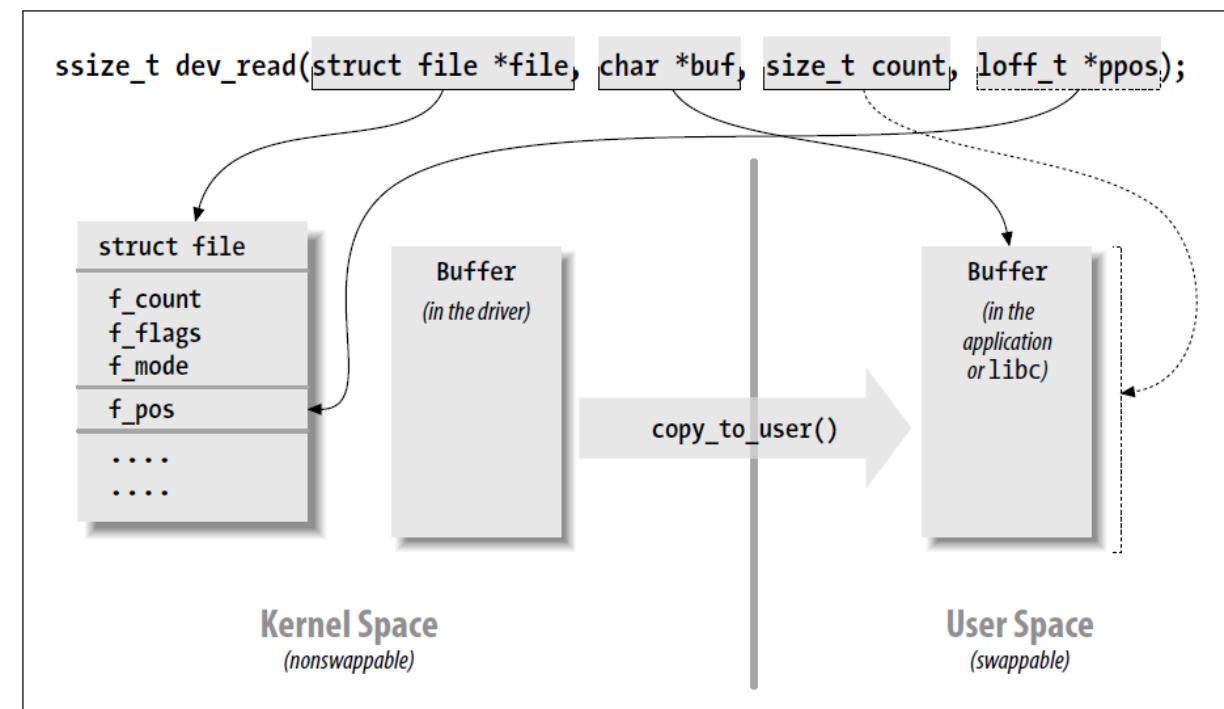
File Operation Structure defined in linux/fs.h

```
struct file_operations mydev_fops = {  
    .owner = THIS_MODULE,  
    .read = mydev_read,  
    .write = mydev_write,  
    .ioctl = mydev_ioctl,  
    .open = mydev_open,  
    .release = mydev_release,  
};
```

There are many other file_operations structure members but have listed important ones only.

File Operation Structure defined in linux/fs.h

- .owner : struct module *owner → owner of the structure (THIS_MODULE)
- mydev_read: ssize_t (*read) (struct file *file, char __user * buf, size_t count, loff_t *ppos); → function pointer which implements reading of data from device.
 - char __user * is pointer to user mode buffer to which data need to be read
 - size_t is number of bytes to be read
 - loff_t represents current file position for reading and writing.



File Operation Structure defined in linux/fs.h

- mydev_write: `ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *)`; → function pointer which implements writing of the data to device. `loff_t` represents current file position for reading and writing.
- mydev_ioctl : `int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long)`; → ioctl function implementation such as formatting etc
- mydev_open : `int (*open) (struct inode *, struct file *)`; → open the device and allocate file structure
- mydev_close: `int (*release) (struct inode *, struct file *)`; → close device and release file structure

Useful memory function inside driver code

- `void *kmalloc(size_t size, gfp_t flags);` defined in `linux/slab.h`
 - `GFP_USER` – Allocate memory on behalf of user. May sleep.
 - `GFP_KERNEL` – Allocate normal kernel ram. May sleep.
 - `GFP_ATOMIC` – Allocation will not sleep. May use emergency pools. For example, use this inside interrupt handlers.
 - `GFP_HIGHUSER` – Allocate pages from high memory.
 - `GFP_NOIO` – Do not do any I/O at all while trying to get memory.
 - `GFP_NOFS` – Do not make any fs calls while trying to get memory.
 - `GFP_NOWAIT` – Allocation will not sleep.
 - `__GFP_THISNODE` – Allocate node-local memory only.
- `void kfree(const void *objp)` → free allocated memory
- `unsigned long copy_from_user(void *to, const void __user *from, unsigned long n);` → copy data from user space buffer to kernel space buffer
- `unsigned long copy_to_user(const void __user *to, const void *from, unsigned long n);` → copy data from kernel space buffer to user space buffer

Cdev and File Operations

DeviceDriver\mydriver5.c

DeviceDriver\mydriver5 makefile

- Create file_operations structure with function pointers for function that you implement

```
static struct file_operations fops =
```

```
{  
    .owner      = THIS_MODULE,  
    .read       = mydev_read,  
    .write      = mydev_write,  
    .open       = mydev_open,  
    .release    = mydev_release,  
};
```

- Initialize cdev structure with specific file_operations

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

- Tell kernel which device uses this cdev structure

```
int cdev_add(struct cdev *cdev, dev_t dev, unsigned int count);
```

Test the driver Code using Application

- Your device will by default have permission to the user who created the device which is generally root because we use sudo

```
$ ls -l /dev/mydevice
```

```
crw----- 1 root root 243, 0 Mar 28 15:41 mydevice
```

- Enable the read-write permission for other users too (otherwise only root user can access the device)

```
$ ls -l /dev/mydevice
```

```
crw-rw-rw- 1 root root 243, 0 Mar 28 15:41 mydevice
```

Now you can test device with your application

[DeviceDriver\mydriver5_test.c](#)