CITS2002

The C programming language

C is a general purpose programming language. It is procedural and not object oriented.

- Strongly typed variables
- Constants
- Standard data types
- Enumerated types
- User-defined types
- Aggregate structures
- Standard control flow
- Recursion
- Program modularization

C is an excellent systems programming language. Provides an excellent operating system interface through its well designed hardware and operating system independent standard library. Development of C began in 1972, and was later used to write unix but now is used for the basis of all modern operating systems.

C is beneficial over assembly languages as it is human readable - small performance increases do not match the huge increase in efficiency of writing the program.

C has been standardised in the late 1980s to ensure a standard across all C compilers and systems. C99 is the current primary standard, however C11 has since come out. Formalized standards provide:

- Representations of C programs
- The syntax and constraints of the C language
- The semantic rules for interpreting C programs
- The representations of input data to be processed by C programs
- The representation of output data produced by C programs
- The restrictions and limits imposed by a conforming implementation of C

They do not specify:

- The mechanism by which C programs are transformed for use by a data processing system
- The mechanism by which C programs are invoked for use by a data-processing system

- The mechanism by which input data are transformed for use by a C program
- The mechanism by which output data are transformed after being produced by a C program
- The size or complexity of a program
- Minimum requirements of a data processing system that conforms to C implementation

These standards omit things such as graphics, networking and cryptography, but instead provides the ability to create third party libraries.

The structure of a C program

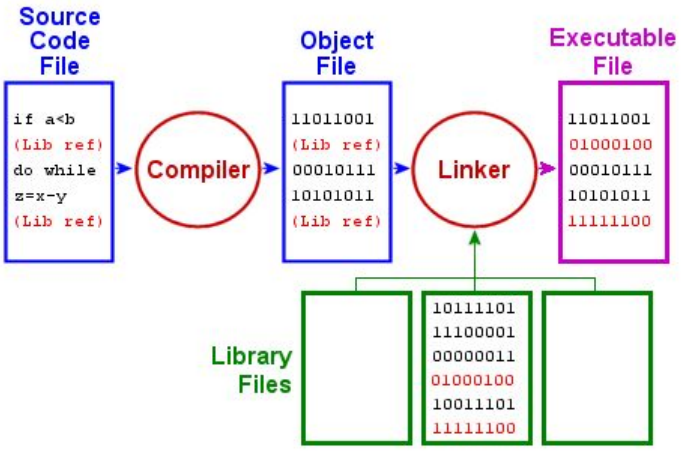
Any line that is preceded by a "#" is a preprocessor line.

```
#include <stdio.h>

#define ROT 13
```

Are two preprocessor functions which create macros and import libraries.

The C compiler converts human readable text into a program - this human readable text is called portable at the source-code level as it can be ported between machines before it is compiled. Before we run our programs they must be compiled on a system - the compiler translates source code files into object-code files. The compiler then links object files to produce an executable program (which are called binary or executables).



Variables have their basic data types (bool, int, char, float, double) and are stored in the memory. They also have scope - global scope is in which the variable is accessible from outside any specific function, and block scope is in which a variable is declared (and therefore only accessible from) in a function or statement block.

Flow-control in C programs

C programs will typically start from within the main function, and execute their statements, and then exit to the operating system. Default flow control will execute each statement in order from top to bottom - The use of conditional statements and loops, however, can change this.

A conditional statement will first evaluate a boolean condition and take action based on whether it is true or false. In c to use the boolean data type the following is required:

```
#include <stdbool.h>
```

Switch statements offer a very similar use as an if else statement, however make more sense for using a large number of distinct values.

Bounded loops allow for repetitive actions to occur in a distinct number of iterations (ie. the number of iterations is typically known in advanced). The while loop and the do while loop are examples of unbounded loops, and are for when the number of iterations required is not necessarily known, and are typically used on more global variables than a variable created specifically for the loop (as occurs in the for loop). Loops can be nested within each other to create a 2D or 3D type effect on matrices.

To change the flow from within a loop we will typically use:

```
break;

continue;
```

Break will break out of the loop to the block above (which may be another loop), whereas continue will move to the next iteration of the same loop, skipping the rest of the block below it.

An introduction to operating systems

An operating system is: A piece of systems software that provides a convenient, efficient environment for the execution of user programs. They are the largest and most complex programs on a computer, they provide:

- A user's portal
- A programming environment
- Time and space efficiency
- Economically efficient

An operating system must also be able to support hardware and software that is not around at the time of its creation.

The operating systems primary focus is to be the manager of software and hardware resources. Traditional operating systems offer the following services:

- Cpu Scheduling: Distribution of computing time among several processes to create the appearance of simultaneous processes
- Memory management: Dividing and sharing physical memory between processes
- Swapping: moving processes and their data between primary and secondary memory to present the illusion of greater primary memory
- I/O device support: provide specialized code to optimally support device requirements
- File systems: organize mass storage into files and directories
- Utility programs: Accounting, settings, manipulating the file systems
- A command interface: textual or graphic to enable interactive interrogation and manipulation of the operating system's features
- System calls: allow constrained access to the interior of the running operating system
- Protection: keep processes from interfering with each other
- Communication: allows users and their programs to communicate across single machines or across a network

Early operating systems had no real concern for the security of the system, programs, or data - but as resources progressed this security became paramount.

Functions in C

C is a procedural programming language, meaning that its primary synchronous control flow mechanism is the procedure call. C calls its procedures "functions". Functions are required for a few reasons:

- Functions allow grouping of statements together which have a common purpose. Each function should perform a single task
- Stops repeated examples of the same code in a single program which causes confusion and creates bloat
- The operating system's kernel uses functions as well defined "entry-points" called system calls
- Functions provide a convenient mechanism to package and distribute code. We frequently use libraries for this purpose

The role of the **main** function is an important one. We never want to place all of our statements inside of the main function. Main should only:

- Receive and check the program's command line arguments
- Report errors detected with command-line arguments and call them an **EXIT_FAILURE**
- Call functions from main
- If all went well it should **EXIT_SUCCESS**

Any function will have a return datatype - this can be one of the basic data types, or **void** if nothing is to be returned. The function will have to have **return** before the end of the function and it must always be accessible from every flow direction.

Functions have the ability to have parameters passed to them - as with the return datatype, this can be **void**, or one of the basic data types. This can also have multiple inputs. Note that the order in which a function evaluates (the order of evaluation) is not defined. Also note that **return** is not a function call! It is not written as:

```
return(x);
```

But simply as:

```
return x;
```

An overview of computer hardware components

Traditionally inside a computer are 4 main structural components:

- CPU: undertakes arithmetic and logical computations, and directs most I/O service from memory and peripherals
- RAM: used to store both programs and data. The CPU reads and writes items to the memory both at the direction of programs and as an artifact of programs running
- Peripheral devices, secondary storage, I/O etc.
- A communications bus which connects all of the components together

The flow of a basic single systems bus operating system is as follows:

1. The CPU fetches a copy of the contents of a uniquely-addressed memory location by identifying the required location in its MAR
2. Depending on why the CPU requires the value, it executes the contents as an instruction, or operates on the contents as data
3. Similarly, the CPU locates data from, or for, distinct location in the I/O devices using its I/O

IO data rates are a huge bottle neck of the system - as a cpu may be running at 4.4GHz, the data rate of IO can slow this down:

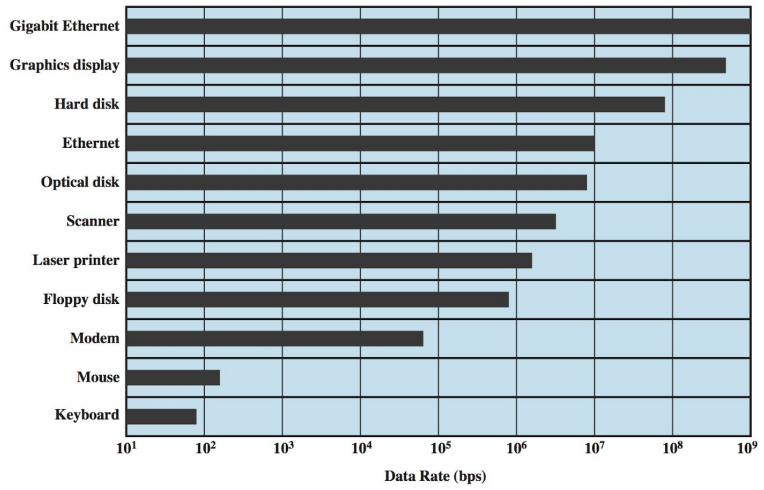
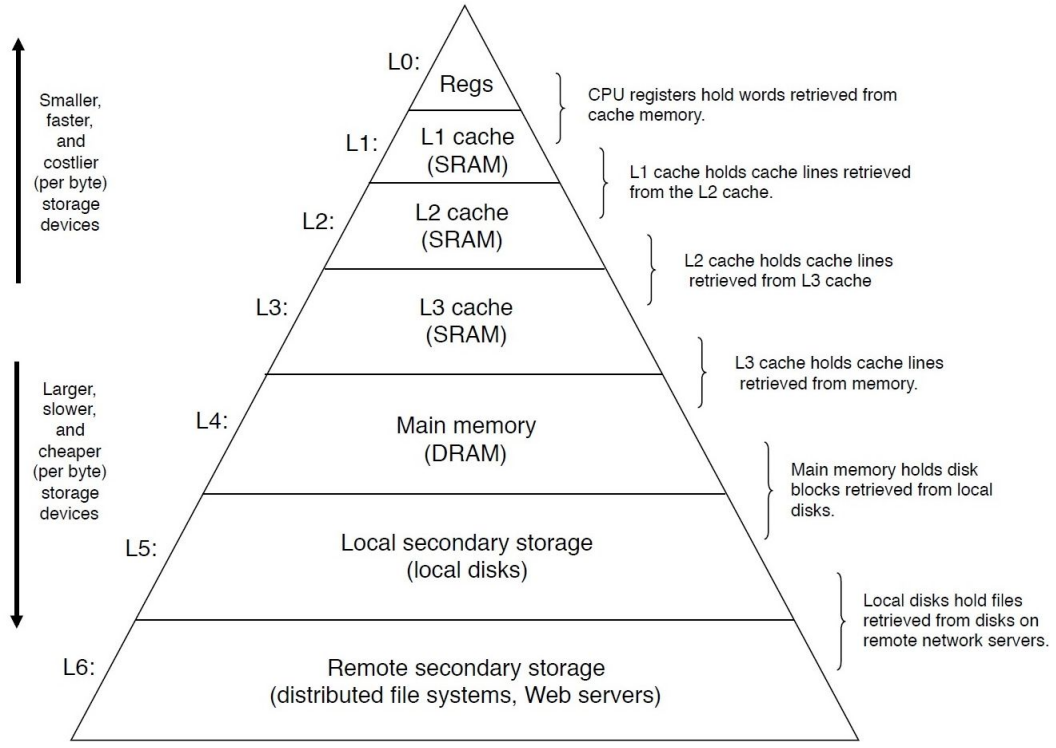


Figure 11.1 Typical I/O Device Data Rates

The processor contains a small number of very fast memory locations called **processor registers**. Data in the register can be read and written very rapidly, much faster than the main memory. Registers generally come in two of the following types:

- **User accessible** registers are accessible to programs and are broken down into one of the two subtypes:
    - **Data registers** hold values before the execution of certain instructions and hold the results after execution
    - **Address registers** hold the addresses of memory locations used in the execution of the program
- **Control and status registers** hold data that the processor itself maintains in order to execute the programs

There is a memory hierarchy within the system which is key to providing the greatest efficiency and fastest operation:



To maintain the maximum efficiency the system relies on a combination of different memories - this ensures a balance between cost efficiency (registers cost far more per kilobyte) and time efficiency.

Bits: *can contain a single value of 0 or 1, or true or false.*

Bytes: *are the smallest addressable value, and are typically 8-bits in modern computers.*

Word: *is the default data size for a processor (ie. 16 bit, 32 bit, 64 bit)*

Critical errors occur when a bit sequence is interpreted in the wrong context. If a processor attempts to execute a meaningless sequence of instructions, a processor fault will generally result: Linux announces this as a "bus error". Similar faults occur when instructions expect data on aligned data boundaries, but are presented with unaligned addresses.

Aggregate data structures in C - arrays and strings

An array is a simple data structure that allows data to be stored and accessed, where the data items themselves are closely related:

- 1-dimensional arrays are often termed vectors,
- 2-dimensional arrays are often termed matrices (as in our example, above),
- 3-dimensional arrays are often termed volumes, and so on.

Note that strings are 1 dimensional arrays of characters. 1-dimensional character arrays, in order for them to be strings, must be terminated by a **null byte '\0'** at the end of the string. This will mean that the character array will need to be one place larger than the string,

allowing for the additional null byte on the end - even though this will report as the size of the string not including the null byte.

Operating system processes

A process is a fundamental activity of an operating system - The creation, management and termination of a process is paramount. A process can be described as the following:

- A program under execution
- The animated existence of a program
- An identifiable entity executed on a processor by the operating system\
- An executable instance of a program
- The associated data operated upon by the program
- The program's execution context

Each process is in a certain state at any one time, and this can be taken from the perspective of the processor or the process.

For the processor:

- the processor's only role is to execute machine instructions from main memory
- the processor continually executes the sequence of instructions indicated by the program counter (PC).
- The processor is unaware that different sequences of instructions have logical existence.

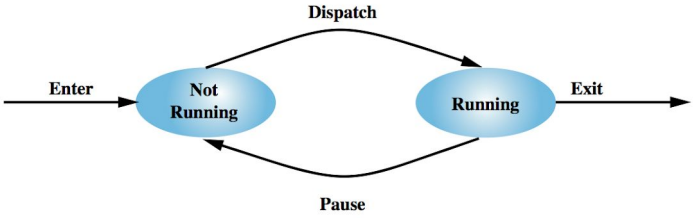
From the processes point of view:

- It is either been executed by the processor (**running)**, or
- It is waiting to be executed (**Ready**)

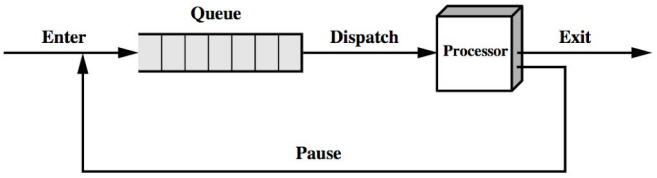
The operating system must manage the execution of existing and newly created processes, moving them between the two states until they have completed. The procedure of a process is as follows:

1. Newly created processes are marked as ready
2. There can only be a single process in the running state - it will either complete its execution and terminate or be paused, butting it in the the ready state
3. Once there is no running process, one of the cued ready processes will be commenced

The operating system has this role as a dispatcher - dispatching work for the processor. The simple two state model can be displayed in the following:



(a) State transition diagram



(b) Queuing diagram

In the creation of a new process the operating system must allow resources for both the operating system and the process. There are a few typical reasons why a process will terminate:

- normal termination,
- execution time-limit exceeded,
- a resource requested is unavailable,
- an arithmetic error (division by zero),
- a memory access violation,
- an invalid request of memory or a held resource,
- an operating system or parent process request, or
- its parent process has terminated.

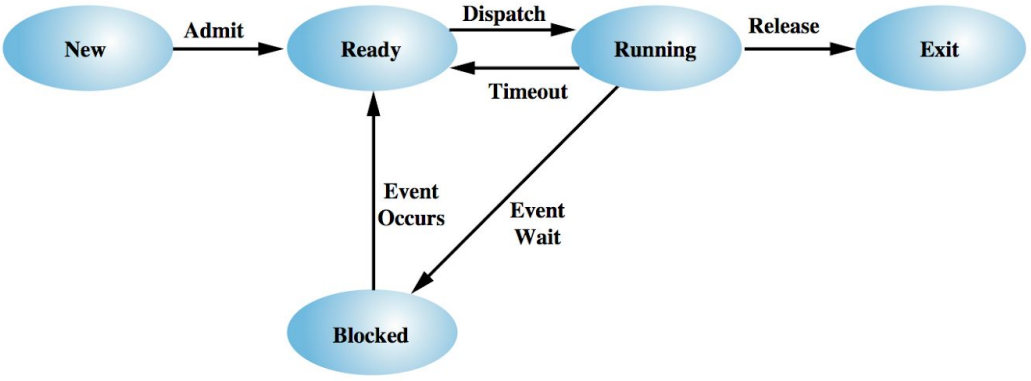
The return result (exit status) of a process will be made available to some other processes.

The operating system must be fair amongst the processes to maximize efficiency. This can be done by allowing a predetermined period before moving the running process to the ready queue - This can be done with timer interrupts which is created by a hardware timer which periodically generates an interrupt. When this interrupt is met by the processor, it will begin execution of the interrupt handler.

The blocked state is one in which the process cannot have access to resources that it requires to do its process. This blocked state moves the process from a running state to a ready state, and is the following order:

1. requesting I/O to or from the device,
2. moving the process from Running to Blocked,
3. preparing to accept an interrupt when I/O completes.

Basically when the I/O completion interrupt occurs, the requesting process is moved from blocked to ready. This can now be visualised in the following 5-state process:



Where the new state is for newly created processes not yet admitted to the ready queue, and the exit state is for terminated processes whose return result or resources may be required by another process. The possible state transitions are as follows:

Null → New	a new process is requested.
New → Ready	resources are allocated for the new process.
Ready → Running	a process is given a time quantum.
Running → Ready	a process's execution time quantum expires.
Running → Blocked	a process requests slow I/O.
Blocked → Ready	an I/O interrupt signals that I/O is ready.
Running → Exit	normal or abnormal process termination.
Ready or Blocked → Exit	external process termination requested.

Swapping is a way in which the processor can create more room for processes. The processor will kick out processes from the main memory onto the disk to create more room - this can make things slower.

File input and output (I/O)

General programs will often require reading from human readable files. We open text files using C's **fopen()** function. The returned value of this function is a **FILE pointer** and is used in all subsequent operations on that file.

```
FILE   *dict;
```

```
//  ATTEMPT TO OPEN THE FILE FOR READ-ACCESS
    dict = fopen(DICTIONARY, "r");
//  CHECK IF ANYTHING WENT WRONG
    if(dict == NULL) {
        printf( "cannot open dictionary '%s'\n", DICTIONARY);
        exit(EXIT_FAILURE);
    }
//  READ AND PROCESS THE CONTENTS OF THE FILE
    ....
//  WHEN WE'RE FINISHED, CLOSE THE FILE
    fclose(dict);
```

If fopen returns the NULL value, it indicates that the file may not exist, or the operating system has not given the process permission to access it as requested. We need to request access in a particular way:

- "r" open for reading
- "r+" open for reading and writing
- "w" create or truncate file, then open for writing
- "w+" create or truncate file, then open for reading and writing
- "a" create if necessary, then open for appending (at the end of the file)
- "a+" create if necessary, then open for reading and appending

This **access mode** will dictate what the process can and can't do with the particular file in the future. The **fgets()** function will be passed file pointer, and will return a single line at a time when traversing the file, and returns NULL when the end of the file has been reached, or an error has been detected. It is also important that the file is closed after operations on it have been completed.

Note that the fgets function will actually also get the **newline character '\n'** at the end of each line and can cause issues with processing - it can be useful to treat this as a null byte.

The **fputs()** function can be used to write a line of text to an opened file.

Just as text data can be read, so can binary data. **fread()** can be used to read binary data, as fgets will search for a newline or null byte, which is not present in binary data.

```
fread(void *ptr, size_t eachsize, size_t nelem, FILE
*stream);
```

Operating system services

All operating systems provide service points through which a general application program may request services of the operating system kernel. These points are variously termed system calls, system traps, or syscalls. The number of system calls should be minimised, in the belief that doing so simplifies the design.

An application program may invoke a system call provided that the languages run-time mechanism supports the operating systems mechanism for doing this.

The system call will return **status values**. This is a consistent interface between application processes and the operating system's kernel. The globally accessible **errno** value will convey additional state and values returned via larger structures passed to the kernel. The errno system in c is found int **errno.h** and includes the following integer variable values:

```
#define EPERM     1     /* Operation not permitted */
#define ENOENT    2     /* No such file or directory */
#define ESRCH     3     /* No such process */
#define EINTR     4     /* Interrupted system call */
#define EIO       5     /* I/O error */
#define ENXIO     6     /* No such device or address */
#define E2BIG     7     /* Arg list too long */
#define ENOEXEC   8     /* Exec format error */
#define EBADF     9     /* Bad file number */
#define ECHILD   10     /* No child processes */
```

System calls will always return a value, with zero being a success, and nonzero being a failure. Using **perror** is perfect for getting the errno from an event, and will provide consistent error reporting.

Although it appears a program begins at main, standard libraries must often prepare the process's execution environment. An **environmental variable** and are maintained with calls to standard library functions such as **putenv** and **getenv**. The processes environment is inherited by its child processes, and the user's environment variable are never used by the kernel itself.

Execlp is an example of a C library function that may be called to commence execution of a new program. The following process is followed:

1. Execlp will receive the name of the new program and the arguments provided to the program

2. It will locate the value of the environment variable PATH, assuming it to be a colon-separated list of directory names to search, and append the program's name to the directory component
3. It makes successive calls to the system call **execve()** until one of them succeeds in the beginning execution of the required program.

Creating new processes

The c preprocessor allows for the programmer to make **assertions** about the correctness of the code. Each assertion is checked as the program runs.

- If the assertion is correct the execution continues
- If it asserts fail then execution is halted, and an error is returned describing the failed assertions.

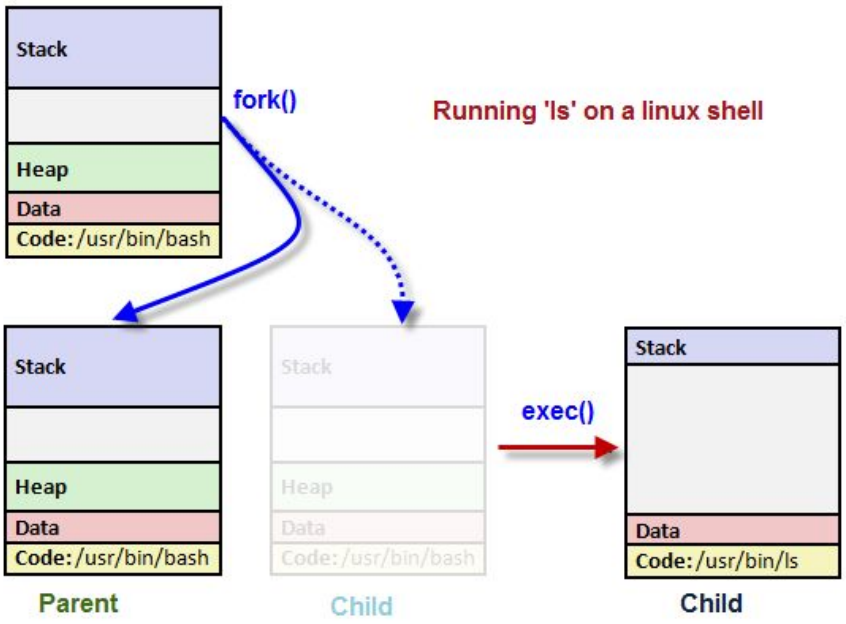
```
#include  <stdio.h>
#include  <assert.h>
    ....
    for(int i=0 ; i <= 5 ; i = i+1)
    {

           assert(i != 5);
           printf("i = %i\n", i);

           assert(i*i < 25);
           printf("i squared = %i\n", i*i);

    }
```

The **fork()** system call is used to create a new process. Each process is uniquely identified by an integer value termed the **process id** (pid). A process can get its own process-id by using the system call **getpid()**, and get its parent's process-id with **getppid()**.



The returned value of fork is different for the parent and child process:

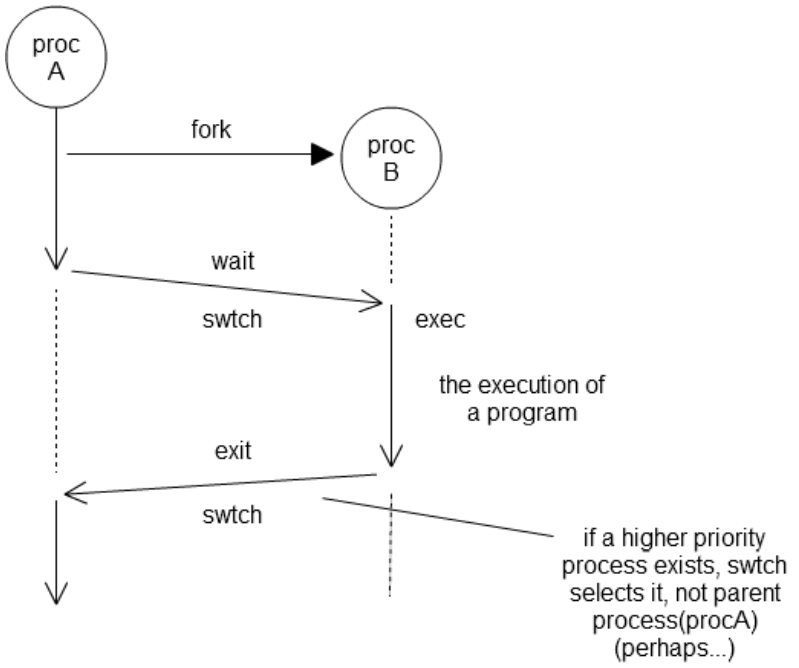
- Fork returns the process id of the child process to the parent process
- Fork returns 0 to the child process - which is not a valid process-id

Both the parent and the child have their own copy of their programs memory, the parent naturally uses the memory it had before, and the child receives its own separate copy of this memory. The two processes can update their own memory without affecting the other process, and they cannot communicate.

Typically the parent will want to know when the child terminates - the following is a typical sequence of events:

1. The parent forks a new child process
2. The parent waits for the child's termination by calling the blocking function **wait(&status)**
3. The child calls exit (either through itself or through an execve function)
4. The child's value is given to exit and is written by the operation to the parents status.

This can be summarised graphically:



When an exec family system call is run it will replace the currently running process (normally the child) with a new process, but under the same process id. This is done by overwriting the current processes memory with instructions and data for the new program.

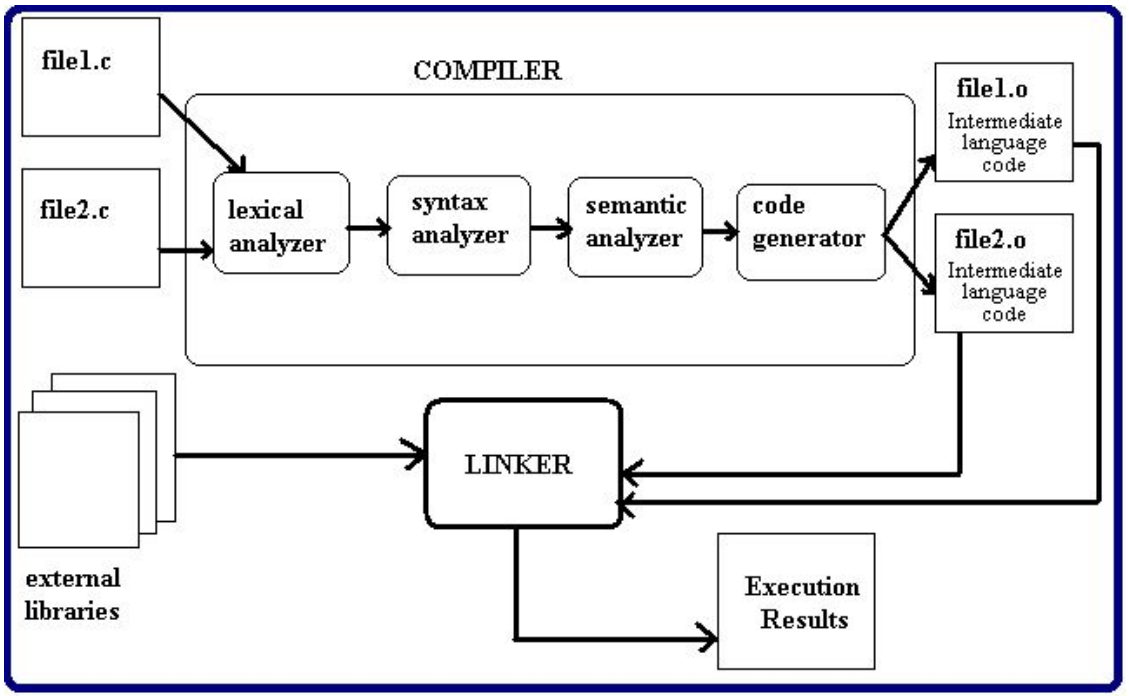
On success execve does not return as it would not be able to return anywhere. On error it will return -1 and the errno is set appropriately.

The exit status on the program is important because it can help the programs direct control-flow (such as in the shell). It is important to know if a child process has worked or not for whether a program such as shell should continue.

Developing C programs in multiple files

When we compile a program using cc we are doing multiple passes of an activity which converts the C source files to executable programs. The following must occur:

1. For each C source file were compiling
    a. The C source code is given to the preprocessor
    b. The preprocessor output is given to the C parser
    c. The parser's output is given to the code generator
    d. The code generator's output is given to the code optimizer
    e. The code optimizers output, called object code, is written to a disk file called an object file
2. All object files are presented to a program called the linker and combined together
3. The linker's output is written to disk as a executable file



The concept of **modularization** must be used when making larger C programs. The motivations for multiple file C programs are as follows:

- each file (often containing multiple related functions) may perform (roughly) a single role,
- the number of unnecessary global variables can be significantly reduced,
- we may easily edit the multiple files in separate windows,
- large projects may be undertaken by multiple people each working on a subset of the files,
- each file may be separately compiled into a distinct object file,
- small changes to one source file do not require all other source files to be recompiled.

After this is all done the object files are all linked into a single executable program.

A header file will be created to provide preprocessor constants and macros, globally visible functions and globally visible variables to all of the relevant files, and will look like the following:

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

// DECLARE GLOBAL PREPROCESSOR CONSTANTS
#define  MAXMARKS  200
```

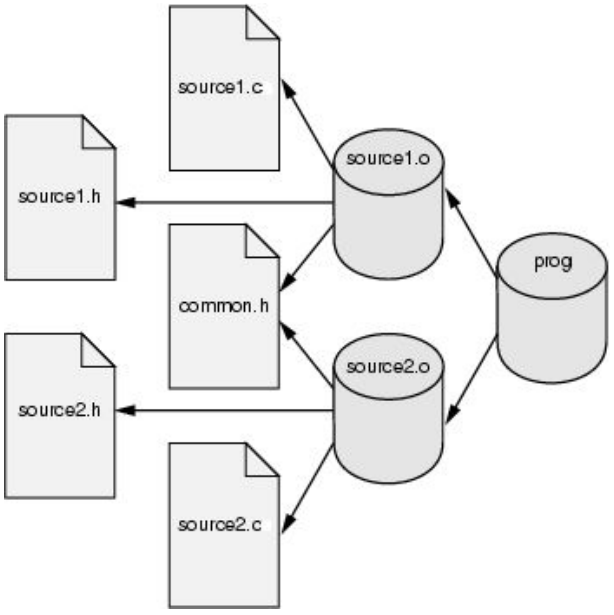
```
// DECLARE GLOBAL FUNCTIONS
extern int          readmarks(FILE *); // parameter is not
named
extern void         correlation(int);  // parameter is not
named

// DECLARE GLOBAL VARIABLES
extern double       projmarks[];       // array size is not
provided  extern double      exammarks[];       // array size
is not provided

extern bool         verbose;           // declarations do not
provide initializations
```

Note that global variables are automatically cleared, and are therefore already filled with null-bytes or zero-bytes.

Maintaining multi file projects can be difficult, but by effectively using **make** we can make if far simpler. Make maintains an up-to-date versions of programs that result from a sequence of actions on a set of files. A make file will dictate actions associated with rules if indicated files are out of date. Make is specifically interested in the dependencies between files.



Changes to some files will not affect others that have been unchanged, and this can allow for compilation to only occur when a large file has been modified for greater efficiency.

An introduction to memory management

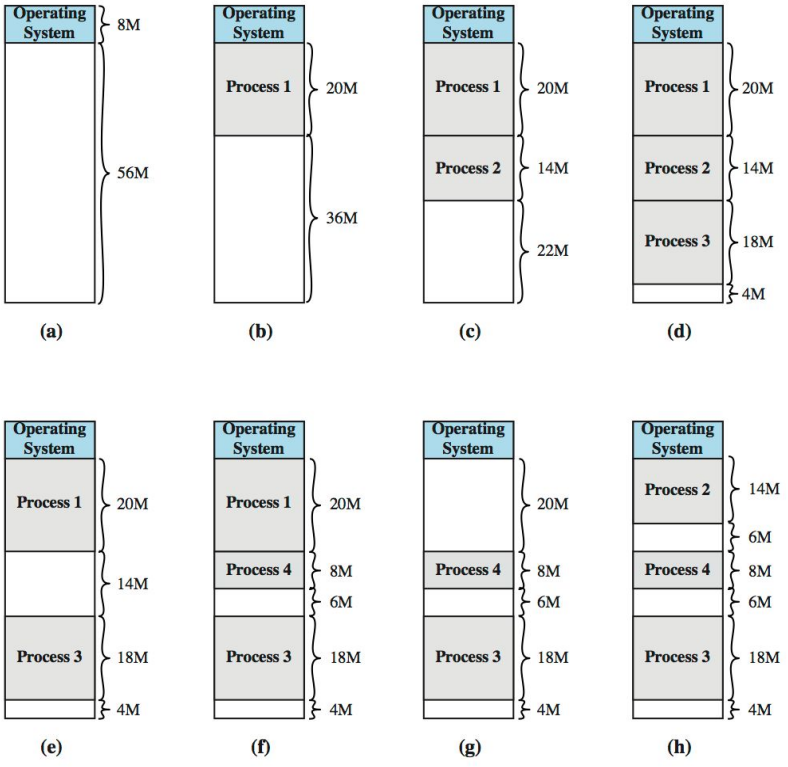
There are 5 main requirements of memory management:

- Logical organisation
- Physical organisation
- Sharing
- Relocation
- Protection

In modern operating systems offering memory management, the operating system itself occupies a portion of the main memory, and the remainder is available for the other processes. The simplest technique to consider main memory being fixed-sized partitions, with two clear choices:

- Equal sized partitions
    - A processes requirements may exceed the partition size
    - A small process still occupies a full partition, this is a wastage which is termed as **internal memory fragmentation**
- Unequal sized partitions - either:
    - A process is placed in the largest enough partition to minimise internal storage fragmentation
    - A process is placed in the smallest enough available partition

Dynamic memory partitioning overcomes some of the shortcomings of static partitioning - being that partitions are of variable lengths. As a process commences it starts with an exactly required size of memory, and no more:



There are three primary algorithms for partition placement algorithms:

- First fit: finds the first unused block of memory that can contain the process, searching from address 0
- Best-fit: Finds the smallest unused block that can contain the process
- Next-fit: remembers where the last processes memory was allocated and finds the first unused block that can contain the process, searching from the last memory block

Simple memory management schemes share one significant assumption: that, when a process is swapped-out, it will always be swapped back into memory, having access to the same memory locations as before.

Logical address: *is a reference to a memory location independent of any current assignment of data to main memory.*

Relative address: *is a logical address expressed relatively to a fixed location, such as the beginning of the processes memory*

Physical address (absolute): *is an actual location on physical main memory.*

Simple memory management schemes share one significant assumption: that, when a process is swapped-out, it will always be swapped back into memory, having access to the same memory locations as before.A **hardware base register** is used to remember the start of a process's partition. A **hardware bounds register** is used to remember the partition's extent. Each process requires a base and bounds register.

Addresses and pointers in C

Pointers are a method of addressing and modifying memory - if done correctly this can increase the program's efficiency, if done wrong can make programs do strange and incorrect things. Three statements can be used to summaries pointers:

- Pointers are variables that hold the address of a memory locations
- Pointers are variables that point to memory locations
- Pointers point to memory locations being used to hold variables values/contents

The & operator is pronounced as the "address of operator". It tells us the pointer address of a regular variable. A pointer is defined using the * operator. A pointer can be dereferenced (after it has been initialized) by using the same * operator - this will return the pointer's value.

Pointer arithmetic is a mechanism for changing where there pointer is pointing to, and moves to/from successive memory locations. This is generally done using pre and post increments.

```
int totals[N];
int *p = totals; // p points to the first/leftmost element of
totals
*p = 0;  // set what p points to to zero
++p;     // advance/move pointer p "right" to point to the
next integer
```

A pointer will move (when incremented) by values that are multiples of the size of the base type which the pointer is pointing to.

Pointers can be returned from functions, which is essentially returning an address to an already made variable. This can be used to return arrays.

The principle of referential locality

By comparing paging and segmentation with the much simpler technique of partition we see two clear benefits:

- As a process is swapped-out and back in again, they occupy different regions of physical memory - this is possible because hardware efficiently translates each logical address to a physical address at run time
- A process is broken into either pages or segments, and these do not need to be continuous in physical memory

If the above two characteristics are present, then it is not necessary for all pages (or all segments) of a process to be in memory at any one time during its execution. The following advantages come from paging and segmentation over partitioning:

- More processes may be maintained in physical memory (ready or running)
- If the swapping space is larger than the physical memory, any single process may now demand more memory than the amount of physical memory stored. This is referred to as “virtual memory”

The principle of program locality again tells us that at any time, only a small subset of a process's instructions and data will be required. In a steady state the memory can be fully occupied by ready or running processes but:

- If the processes working sets are permitted to be too large fewer processes can ever be ready
- If the processes working sets are forced to be too small then additional requests must be made of swapping space to retrieve required pages or segments

In simple paging each process has its own table entry - When a process's (complete) set of pages were loaded into memory, the current (hardware) page tables were saved and restored

by the operating system. With virtual memory the same approach is taken but the contents of the page tables become more complex, and require the additional informations:

- If the page is present in physical memory
- If the page has been modified since it was brought into physical memory

The following considerations must be made when implementing virtual memory on an operating system:

- When should a process's pages be fetched?
- Where in physical memory should pages be allocated?
- Which existing blocks should be replaced?
- How many processes to admit to the Ready and Running states?

Dynamic memory allocation

When a function runs some process on variables (for example a swap function) they are not actually being swapped - but rather are having copies being swapped. This is to say that a function will have its own copy of variables which are not accessible from the function calling the swap function (under normal operations). Rather we need to pass a **reference** to the function - this allows the swap function to access the variables and make changes to them:

```
swap(&a, &b);
```

What this essentially does is gives the memory location to the function so it can directly modify these items. This can also be useful for duplicating a string - to do this we will need to use memory allocation. To allocate new memory we won't necessarily know how big the memory will need to be until the function is called. Instead of using fixed size arrays, we need to **dynamically request memory** at runtime to hold our desired result. **malloc()** is one such function which allows memory allocation and is within the **stdlib.h** header file.

- Malloc is a function that returns a pointer, it is a **void *** or **void star** pointer as it does not know exactly what the return type is - as far as malloc is concerned it is simply a pointer to something
- Malloc needs to be informed of the amount of memory that it should allocate - the number of bytes required. **sizeof()** will return a **size_t** value which will inform malloc of the amount of memory needed.

If the malloc returns a null value this means the malloc failed.

```
char *new = malloc( strlen(str) + 1 )
```

It is often required to clear the newly assigned memory - to do this we used **calloc** which accepts: calloc(size_t nitems, size_t itemsize). It is used in the same way as malloc but will then not need to have the memory all assigned to a null value.

Once we are done with the memory it is appropriate to free it. This is done using **free()** which accepts the pointer to the malloc or calloc location. The operating system will do that for us if the process is finished but it is always best practice.

For when we are growing or shrinking our allocated memory, we use **realloc(void *oldpointer, size_t newsize)**.

Sorting an array is a common problem that we will face, and the **qsort()** function provided by the C standard library allows us to do this. qsort :

- Is a function returning no values
- Receives 4 parameters
    - First is the pointer to the array
    - Second is how many elements in the array
    - Third indicates the size of each element
    - Forth is broken down a bit more:
        - Comparison function is a function that we provide that returns an integer
        - It receives 2 parameters

A program's standard I/O streams

The two main standard output streams are **stdout** and **stderr**. We can print to this stream simply by using **printf()** or **fprintf()** , the latter of which can have its output specified.

The FILE * datatype (a file pointer) is an abstraction over the file descriptors to manage access to descriptors. When reading from a file the code for using file pointers and file descriptors is very similar - However with I/O streams **input buffering** must be taken into account.

Note that stdout is buffered (for efficiency) but stderr is unbuffered to ensure output comes out immediately.

In the standard command line we know that argc refers to the count of the arguments, and argv receives at least one argument (the program name). Argv is really an array of pointers to characters. A system called **getopt** has been developed to help command switches be parsed and used. **Getopt()** is not a C function and conforms to the POSIX standard, but **strdup()** is one which is widely used within C applications.

Defining and accessing structures

There is often a requirement in programs to have sets of closely related - but different data (for example in a spreadsheet etc.). By using user defined structures, we can collect these related variables under a single structure. This is done in C using the **struct** keyword.

```
//  DEFINE AND INITIALIZE ONE VARIABLE THAT IS A STRUCTURE
Struct
{
    char    *name;   // a pointer to a sequence of characters
    int     red;     // in the range 0..255
    int     green;
    int     blue;

} rgb_colour = {
    "DodgerBlue",
     30,
     144,
     255
};
```

We will also often want to create an array of these structures which is done in the following way:

```
Struct
{
    char    teamname[MAX_TEAMNAME_LEN+1];        // +1 for
null-byte

//  STATISTICS FOR THIS TEAM, INDEXED BY EACH TEAM'S 'TEAM
NUMBER'
    int     played;
    int     won;
    int     lost;
    int     drawn;
    int     bfor;
    int     bagainst;
    int     points;
} team[MAX_TEAMS];    //  DEFINE A 1-DIMENSIONAL ARRAY NAMED
team
```

To access from the structure we then need to know which element we are after (for example the team number). This can then allow us to read the team name and values:

```
Team[0].teamname
```

The above will give us the team name.

Structures can be very useful for storing OS information such as time intervals. The “.” notation of a structure gives us access to a single field, however by using “->” we are able to gain access to a structure via a single pointer to the structure.

Using structures we are able to define our own more complex data types - this is done using the **typedef** keyword.

```
typedef struct
{
    char    teamname[MAX_TEAMNAME_LEN+1];        // +1 for
null-byte
    ....
    int     played;
    ....
} TEAM;

TEAM    team[MAX_TEAMS];
```

The DIR datatype can be used to open file directories:

```
#include  <stdio.h>
#include  <sys/types.h>
#include  <dirent.h>
void list_directory(char *dirname)
{
    DIR             *dirp;
    struct dirent   *dp;

    dirp       = opendir(dirname);
    if(dirp == NULL)
    {
        perror( progname );
        exit(EXIT_FAILURE);
    }

    while((dp = readdir(dirp)) != NULL)
    {
        printf( "%s\n", dp->d_name );
    }
    closedir(dirp);
}
```

Self-referential data structures

Dynamically allocated data structures can be hugely useful when we don't know the size of a data structure at compile time, or at runtime even.

The stack is a simple FILO data structure, and is defined in the following way in C99:

```
typedef struct _s {
    int         value;
    struct _s   *next;
}  STACKITEM;
```

```
STACKITEM    *stack = NULL;
```

Adding items to the stack is done at run time. The need to do either adding, or removing is not known until runtime, and the data will determine how large the stack eventually grows to.

```
STACKITEM  *new = malloc( sizeof(STACKITEM) );
new->value   = newvalue;
new->next    = stack;
stack        = new;
```

In this way we are creating a new stack item, adding its value and then adding the next value as the previous value on the stack - then creating this new stack item as the top of the stack.

The list data structure addresses some issues with the stack data structure - it is defined in the following way:

```
typedef struct _l {
    char        *string;
    struct _l   *next;
} LISTITEM;

LISTITEM   *list  =  NULL;
```

File-system basics

It is clear that an operating system will require an efficient, convenient and robust filing system. The following are important concepts in a file system:

- Fields: represent the smallest logical item of data "understood" by a file-system: examples including a single integer or string. Fields may be of a fixed length (recorded by the file-system as "part of" each file), or be of variable length, where either the length is stored with the field or a sentinel byte signifies the extent.
- Records: are familiar collections of identically represented records, and are accessed by unique names. Deleting a file (by name) similarly affects its component records. Access control restricts the forms of operations that may be performed on files by various classes of users and programs.
- Databases: consist of one or more files whose contents are strongly (logically) related. The on-disk representation of the data is considered optimal with respect to its datatypes and its expected methods of access.

DBMS are used commercially for large scale databases. The DBMS manages the whole physical disk-drive, or a dedicated partition, and effectively assumes much of the operating system's role in managing access, security and backup facilities.

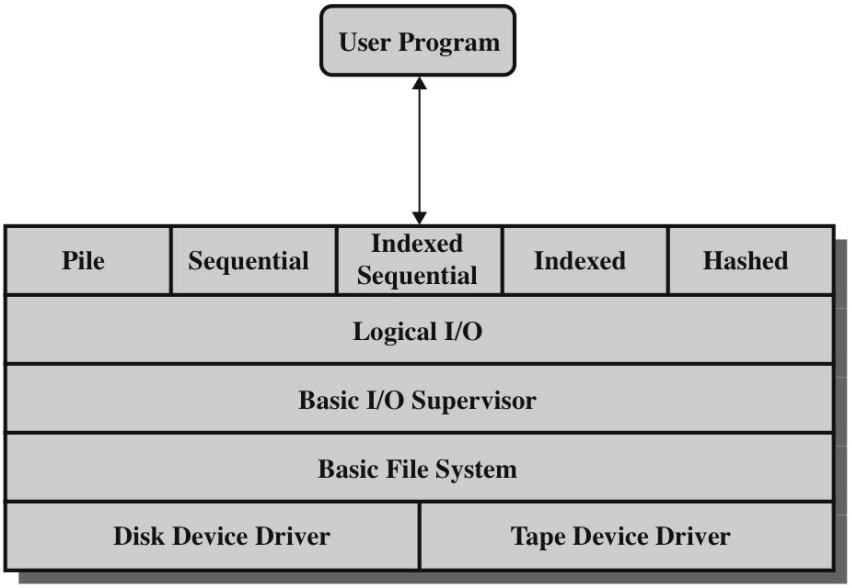
Operating systems use system calls to allow for services relating to disk allocation and access. System calls permit the OS to use file-system access as an opportunity to schedule processes. From the operating system's perspective, the file management system has the following goals:

- to support the storage, searching, and modification of user data,
- to guarantee the correctness and integrity of the data,
- to optimise both the overall throughput (from the operating system's global view) and response time (from the user's view).
- to provide "transparent" access to many different device types such as hard disks, CD-ROMs, and portable devices (accessed by their file-system),
- to provide a standardised set of I/O interface routines, perhaps ameliorating the concepts of file, device and network access.

There are also a few goals of the user in relation to databases:

- the recording of a primary owner of each file (for access controls and accounting),
- each file should be accessed by (at least) one symbolic name,
- each user should be able to create, delete, read, and modify files,
- users should have constrained access to the files of others,
- a file's owner should be able to modify the types of access that may be performed by others,
- a facility to copy/backup files to identical or different forms of media,
- a secure logging and notification system if inappropriate access is attempted (or achieved).

A few components make up the file management system and are displayed graphically in the following:



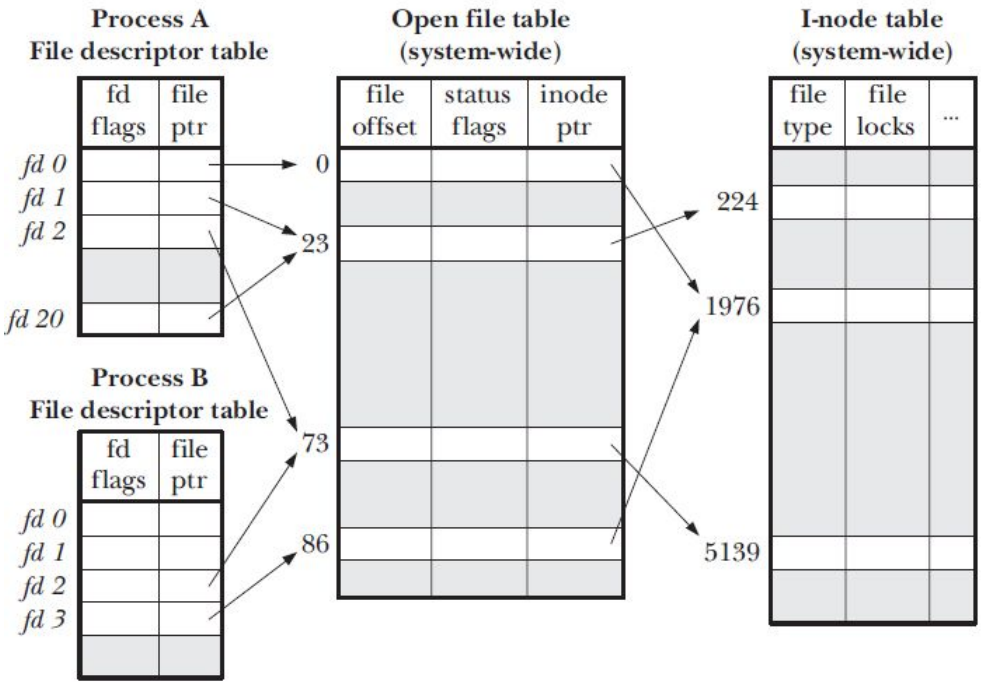
All modern operating systems have adopted the hierarchical directory model to represent collections of files. A **directory** is simply a special type of file storing information about the files it contains. The directory is owned by the operating system itself and must constrain access to the important and hidden information in the directory.

Early operating systems used information about the directory in the directory itself. However today hierarchical directory models are used, and separate information structures and used for each file. These information structures in unix are referred to as **inodes**. The inode will typically contain:

- the file's type (plain file, directory, etc),
- the file's size, in bytes and/or blocks,
- any limits set on the file's size (implied?),
- the primary owner of the file,
- information about other potential users of this file,
- access constraints on the owner and other users,
- dates and times of creation, last access and last modification,
- pointers to the file's actual data blocks.

A file which has been mentioned multiple times in different directories are referred to as file-system links.

- Individual processes access open files through their file descriptor table
- This table indexes the kernel's **global file table**
- That table indexes the devices inode table
-



Contiguous allocation: *Requires that the files maximum size be known at its creation and it cannot grow past that point. Like simple memory allocation schemes it can suffer from fragmentation. This type of allocation stores the files starting block and its length.*

Chained allocation: *Links the new block from the end of the previous block. New blocks may be allocated from any free block on the disk. This means that the blocks no longer need to be touching. As a file's length is extended all relevant touching blocks will be alerted that their start has changed.*

Indexed allocation: *This is the file allocation system used in both UNIX and windows based systems today. The file-allocation table contains a multi-level index for each file - just as we have seen in the use of inodes, which contain direct pointers to data blocks, and pointers to indirection blocks (which point to more data blocks).*

Portability of our programs

The goal of portability is to develop programs that have the maximum likelihood of being portable across a very diverse range of operating systems and hardware platforms.

Conditional compilation can be used to only compile specific things if we are on a specific platform. For instance: inside the preprocessor we can use:

```
#if defined(__APPLE__)
```

To run some specific preprocessor functions if we are on an apple computers. This allows us to change known buggy/different pieces of code on platforms.

Bitwise operators can be used to modify integers. They are essentially manipulating the individual bits of some variable.

Name	Example	Result	Description
bitwise-and	3 & 5	1	1 if both bits are 1.
bitwise-or	3 | 5	7	1 if either bit is 1.
exclusive-or (xor)	3 ^ 5	6	1 if both bits are different.
not	~3	-4	Inverts the bits.
left shift	3 << 2 n << p	12	Shifts the bits of n left p positions. Zero bits are shifted into the low-order positions.
right shift	5 >> 2 n >> p	1	Shifts the bits of n right p positions. If n is a 2's complement signed number, the sign bit is shifted into the high-order positions.

They can be used to store multiple values in a single integer (for example RGB values in a 32 bit integer).