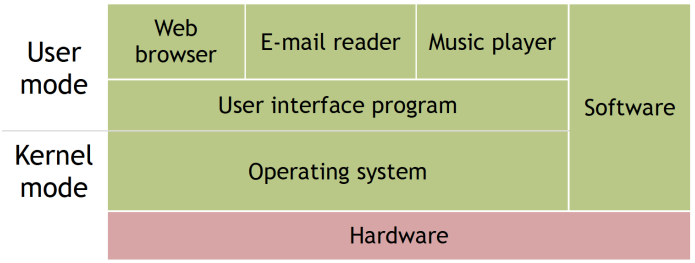
Operating Systems

- OS - is a layer of software that:
  - provide user programs with a better, simpler, cleaner, model of the computer and to handle managing all the resources just mentioned
  - providing application programmers a clean abstract set of resources instead of the messy hardware ones and managing these hardware resources.
- Kernrel mode:
  - The operating system, the most fundamental piece of software, runs in kernel mode (also called supervisor mode).
  - complete access to all the hardware and can execute any instruction the machine is capable of executing.
- User Mode:
  - The rest of the software runs in user mode, in which only a subset of the machine instructions is available.



- OS again:
  - An important distinction between the operating system and normal (usermode) software is that if a user does not like a particular email reader, he is free to get a different one or write his own if he so chooses; he is not free to write his own clock interrupt handler, which is part of the operating system and is protected by hardware against attempts by users to modify it.
  - I/O are forbidden for user mode, to obtain IO, user program must make a syscall named trap, which traps into kernel and invokes an OS.
- Operating system as an Extended machine:

Problem:

- hardware is very complicated and present difficult and inconsistent interfaces to the software developers

Solution:

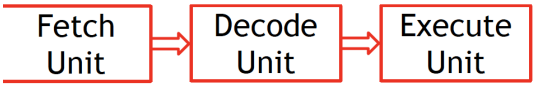
- use abstractions, for instance, a disk driver, that deals with the hardware and provides an interface to read and write disk blocks

- OS as an Resource Manager:
  - manage all pieces of complex system - I/O, memories, etc.

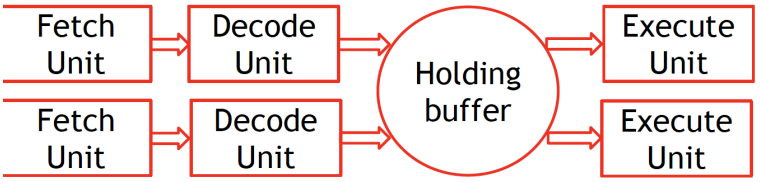
- Multiplexing (sharing) resources can be done in two ways:
  - **Time multiplexing**
    - different programs or users take turns using it (example: CPU, printers)
  - **Space multiplexing**
    - instead of the customers taking turns, each one gets part of the resource (example: memory, disks)

- OS tied to hardware and helps to manage it
  - The basic cycle of every CPU:
    - **Fetch Instruction** - read next expected instruction into buffer
    - **Decode Instruction** - determine opcode & operand specifiers
    - **Calculate Operands** - calculate the effective address of each source operand
    - **Fetch Operands** - fetch each operand from memory. Operands in registers need not be fetched
    - **Execute Instruction** - perform the indicated operation and store the result
    - **Write Operand** - store the result in memory
- CPU contains some registers inside to hold key variables and temporary results:
  - **General registers** - hold variables and temporary results
  - **Program counter** - contains the address of next instruction to be fetched
  - **Stack pointer** - points to the current stack in memory
  - **PSW (Program Status Word)** - bits of results of comparison instructions, the CPU priority, the mode (kernel or user) and various other control bits
-

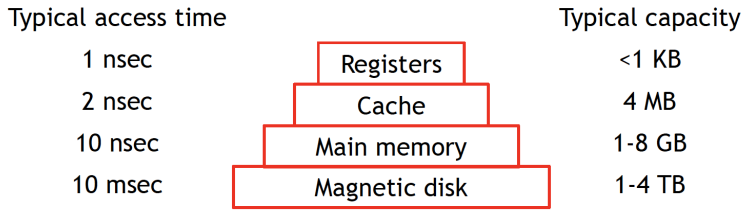
- A three-stage pipeline (Figure 1.7)



- A superscalar CPU (Figure 1.7)



- Multithreading or hyper-threading
  - It allows the CPU to hold the state of two different threads and then switch between them in nanoseconds
  - If one of the processes needs to read a word from memory (long operation), a multithreaded CPU can just switch to another thread thus saving time
  - Multithreading does not offer true parallelism
- CPU chips and GPU has many cores or processors, which require multiprocessor OS. They do thousands of computations in parallel, which is true parallelism
- The memory system is constructed as a hierarchy of layers. The top layers have higher speed, smaller capacity, and greater cost per bit than the lower ones



- Main memory is divided up into cache lines, typically 64 bytes. The most heavily used cache lines are kept in a high-speed cache located inside the CPU

- 64 bytes of cache used because cached the same time as separately, and locality principle

When the program needs to read a memory word, the cache hardware checks to see if the line needed is in the cache. If it is (a cache hit), no memory request is sent over the bus to the main memory. It normally takes about 2 clock cycles

Some machines have two or even three levels of cache, each one slower and bigger than the one before it

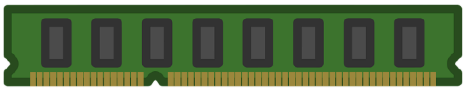
-

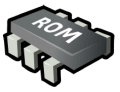
Caching system issues:

- When to put a new item into the cache
- Which cache line to put the new item in
- Which item to remove from the cache when a slot is needed
- Where to put a newly evicted item in the larger memory

-

RAM (Random-Access Memory) serves all CPU requests that cannot be satisfied out of the cache





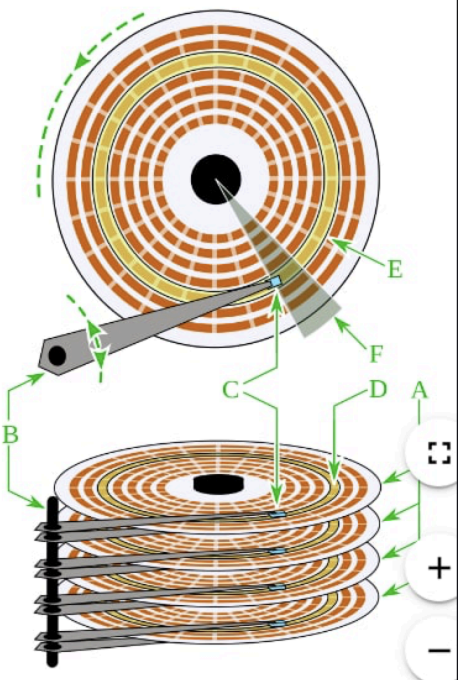
ROM (Read-Only Memory) does not lose its contents when the power is switched off, is programmed at the factory and cannot be changed afterward. On some computers, the bootstrap loader used to start the computer is contained in ROM

-

- EEPROM (Electrically Erasable Programmable ROM) and flash memory are also nonvolatile, but in contrast to ROM can be erased and rewritten
- Flash memory is also commonly used as the storage medium in portable electronic devices. It is intermediate in speed between RAM and disk and it wears out
- Many computers use complementary metal-oxide-semiconductor (CMOS) memory to hold the current time and date and the configuration parameters, such as which disk to boot from
-

Disks (1/2)

- Multi-platter HDD
  - A. Platters
  - B. Actuator arm
  - C. Read/write head
  - D. Segment - an arc on the track
  - E. Track
  - F. Sector



Time to read/write from disk consists of

- **Seek time** is a physical positioning of read/ write head
- **Rotation delay (latency)** is the amount of time it takes for the disc to rotate to the required position for the read/write head
- **Transfer time (data rate)** is the amount of time it takes for data to be read or written

- Generally consist of two parts:
  - a controller that accepts commands from the operating system and carries them out
  - a device itself that has fairly simple interfaces, both because they cannot do much and to make them standard
- IO Device

Device driver is a software that talks to a controller, giving it commands and accepting responses. It has to be put into the OS so it can run in kernel mode

Every controller has a small number of registers that are used to communicate with it. The collection of all the device registers forms the **I/O port space**

On some computers, the device registers are mapped into the operating system's address space (the addresses it can use), so they can be read and written

- like ordinary memory words

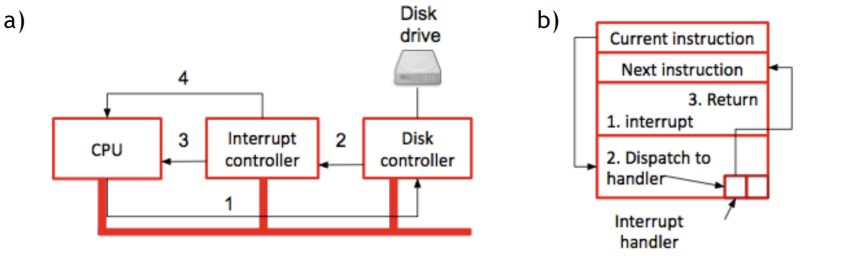
On other computers, the device registers are put in a special I/O port space, with each register having a port address. On these machines, special IN and OUT instructions are available in kernel mode to allow drivers to read and write the registers

Input and output can be done in three different ways:

- **Busy waiting:**
  - the driver starts the I/O and sits in a tight loop continuously polling the device to see if it is done. It has the disadvantage of tying up the CPU polling the device until it is finished

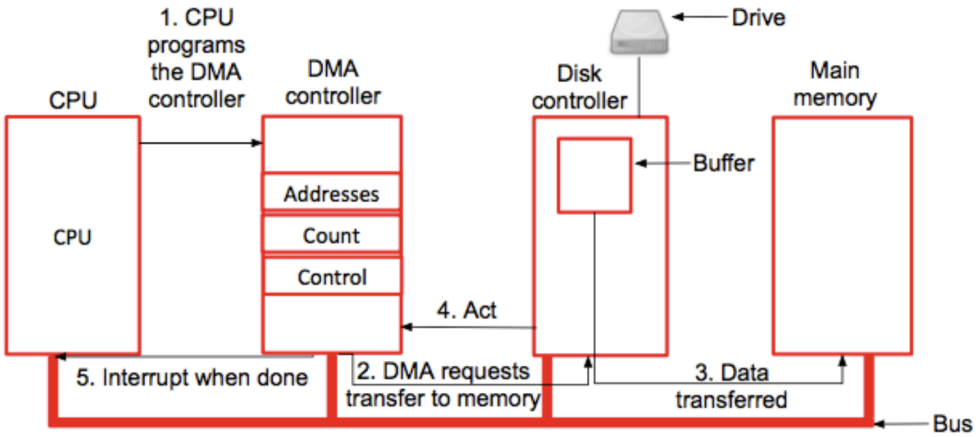
- **Interrupt:**
  - the driver starts the device and asks it to give an interrupt when it is finished. The operating system then blocks the caller if need be and looks for other work to do
  - When the controller detects the end of the transfer, it generates an interrupt to signal completion. The device number may be used as an index into part of memory to find the address of the interrupt handler for this device. This part of memory is called **the interrupt vector**

- a) The steps in starting an I/O device and getting an interrupt
  b) Interrupt processing involves taking the interrupt, running the interrupt handler, and returning to the user program



Direct Memory Access (DMA):

- DMA is a chip that can control the flow of bits between memory and some controller without constant CPU intervention
- The CPU programs the DMA chip, telling it what and where to transfer and lets it go. When the DMA chip is done, it causes an interrupt, which is handled as described above



The system has many buses, each with a different transfer rate and function. The OS must be aware of all of them for configuration and management.

A shared bus architecture means that multiple devices use the same wires to transfer data which needs an arbiter to determine who can use the bus.

A parallel bus architecture means that you send each word of data over multiple wires. For instance, in regular PCI buses, a single 32-bit number is sent over 32 parallel wires.

A serial bus architecture sends all bits in a message through a single connection, known as a lane. Parallelism is still used, because you can have multiple lanes in parallel (send 32 messages via 32 lanes).

- **Universal Serial Bus (USB)** is a centralized bus in which a root device polls all the I/O devices to see if they have any traffic.
- Before **plug and play**, each I/O card had a fixed interrupt request level and fixed addresses for its I/O registers. And two different pieces of hardware might use the same interrupt, so they will conflict

  Plug and play makes the system automatically collect information about the I/O devices, centrally assign interrupt levels and I/O addresses, and then tell each card what its numbers are
- **Basic Input Output System (BIOS)** is a program on the parent board that contains low-level I/O software.

  After the BIOS is started it performs **Power-On Self-Test (POST)** to test integrity and see how much RAM is installed and other basic devices are installed and responding correctly.

- It starts out by scanning the buses to detect all the devices attached to them. Then it determines the boot device by trying a list of devices stored in the CMOS memory

  CMOS (Complementary metal-oxide-semiconductor) is a technology for constructing integrated circuits
- The boot sector (first sector from the boot device) is read into memory and executed. This sector contains a program that normally examines the partition table at the end of the boot sector to determine which partition is active
- **A secondary boot loader** is read in from that partition. This loader reads in the OS from the active partition and starts it. The OS then queries the BIOS to get the configuration information. For each device, it checks to see if it has the device driver.
- Once it has all the device drivers, the OS loads them into the kernel. Then it initializes its tables, creates whatever background processes are needed, and starts up a login program or GUI

- **Mainframes** are room-sized computers in data centers. The operating systems for mainframes are heavily oriented toward processing many jobs at once, most of which need prodigious amounts of I/O
- Typically kinds of service:
  - A **batch** system processes routine jobs without any interactive user present
  - **Transaction-processing** systems handle large numbers of small requests
  - **Timesharing** systems allow multiple remote users to run jobs on the computer at once, such as querying a big database
- They run on servers, which are either very large personal computers, workstations, or even mainframes. They serve multiple users at once over a network and allow the users to share hardware and software resources.

  Examples: Solaris, FreeBSD, Linux, Windows Server.
- ## Multiprocessor Operating Systems

  - Connect multiple CPUs in a single system:
    - parallel computers
    - multicomputers
    - multiprocessors
  - Nowadays all PC and notebooks OS deal with small-scale multiprocessors
    - Examples: Linux, Windows

Embedded Operating Systems

- Run on the computers that control devices that are not generally thought of as computers and which do not accept user-installed software: microwave ovens, MP3 players, TV sets, cars, etc.
  - Examples: Embedded Linux, QNX, VxWorks

Handheld Computer Operating Systems

- OS for PDAs earlier and Mobile Devices nowadays
  - Example: Google Android, Apple iOS

Personal Computer Operating Systems

- You know what is it.
  - Examples: FreeBSD, Linux, Mac OS X, Windows

Sensor-Node Operating Systems

- Each sensor node is a real computer, with a CPU, RAM, ROM, and one or more environmental sensors
- These nodes communicate with each other and with a base station using wireless communication
  - Example: TinyOS

Real-time Operating Systems

- **A hard real-time system** must provide absolute guarantees that a certain action will occur by a certain time.
- **A soft real-time system** is one where missing an occasional deadline, while not desirable, is acceptable and does not cause any permanent damage.
- Example: QNX

QNX SOFTWARE SYSTEMS

f

Smart Card Operating Systems

- The smallest operating systems run on smart cards, which are credit-card-sized devices containing a CPU chip
- **Java oriented smart cards** - means that the ROM on the smart card holds an interpreter for the Java Virtual Machine (JVM). Java applets (small programs) are downloaded to the card interpreted by the JVM interpreter

- Operating systems are normally large C (or sometimes C++) programs consisting of many pieces written by many programmers
- Composite data types can be constructed using:
  - arrays
  - structures
  - unions
- The control statements in C are the next statements (similar to Java):
  - if
  - switch
  - for
  - while
- Pointer is the one feature of C that Java and Python do not have

  A **pointer** is a variable that points to (i.e., contains the address of) a variable or data structure

  Consider the following example:

```
char c1, c2, *p;
c1 = 'c';
p = &c1;
c2 = *p;
```

c1 and *c2* are character variables

p is a variable that points to (i.e., contains the address of) a character

The first assignment (c1 = 'c';) stores the ASCII code for the character "c" in the variable *c1*

The second one (p = &c1;) assigns **the address of *c1*** to the **pointer variable *p***

The third one (c2 = *p;) assigns the contents of the variable pointed to by *p* to the variable *c2*, so after these statements are executed, c2 also contains the ASCII code for "c"

- Some things that C **does not have**:
  - built-in strings
  - threads
  - packages
  - classes
  - objects
  - type safety
  - garbage collection
- All storage in C is either static or explicitly allocated and released by the programmer, usually with the library functions *malloc()* and *free()*
- Programmer have full control over memory
- Along with explicit pointers it makes C attractive for writing OSs
- OSs are basically real-time systems. When an interrupt occurs, the operating system may have only a few microseconds to perform some action or lose critical information
- Having the garbage collector kick in at an arbitrary moment is intolerable

Preprocessor directives are special commands which are evaluated before compilation

- Preprocessor directives start with a hash symbol (“#”)
- They must be written at the beginning of the line, they are not an instruction of the language
- A semicolon (;) is NOT required at the end
- If the entire command does not fit one line, you have to inform the preprocessor that the command continues on the next line by adding a backslash (\) at the end of the line

- Reasons to use preprocessing
  - Source code can be made clearer (using #define)
  - Source code can be split into multiple modules easily (using #include)
  - Sections of code can be shared by several different programs (using #include)
  - It can reduce the work of the compiler by automatically excluding unnecessary sections of code (using #if) or replacing symbolic names with real values (using #define)

#include (1/3)

- It allows to separate the code into separate units in order to modularize the code. This allows to:
  - Develop the code separately
  - "Divide and conquer" approach: clearly separate the problem into smaller parts and solve the sub-problems (Reduction of complexity)
  - Modules can be tested and verified separately
  - Single modules can be exchanged and extended
- This requires clear interfaces between the different modules. This is the aim in using header files.
- Including the **.h** file means to include the interface of the file, which used to be the unit of modularization.

Format:

```
#include <myclass.h>
#include "myClass.h"
```

The filename has to be written in double quotes ("**xxx**") if it is in the same directory or in the user-specified include path

The filename has to be written in angle brackets (**<xxx>**) if it is in the system include path

The include statement is replaced with the content of the given header file

#define (1/15)

Why use #define in C?

To define constants in C

- To make the program easier to read
- If you use a variable throughout the program and then decide to change it, the #define directives eliminate the need to manually change each statement and therefore help to avoid errors

Why not use public variables instead of #define?

- The variable could be inadvertently changed during the execution of the program
- The compiler can generate faster and more compact code for constants than for variables

#define (2/15)

- Format:

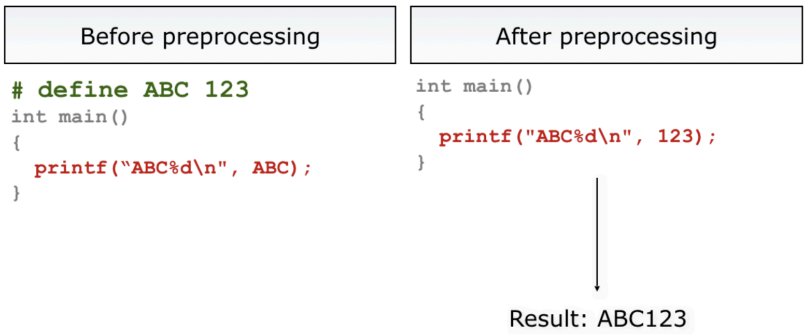
```
#define identifier token-string
```

- Example:

```
#define one 1
```

- The #define directive substitutes a given identifier with the given token-string
- The token-string consists of a series of tokens, such as keywords, constants, or complete statements

#define (4/15)



- #define - Text literals (again constants)

#define (6/15)

```
#define BEGIN {
#define END }
using namespace std;
int main()
BEGIN
  printf("ABC");
END
```

- #define – dangerous step: altering the syntax of C++

#define (7/15)

- #define – macros as kind of functions
- Format:
  #define identifier(identifier, ...) token-string
- Example:
  #define plus(x, y) x+y
- In this case a function-like macro is created where identifier plus parameters are replaced in source code
- In the given example, this means that whenever the preprocessor finds the macro *plus* with two parameters, it replaces it with *x+y*

#define (8/15)

- Why use #define macros ?
  - It is more efficient from a memory usage perspective because the macro is only stored once in the executable even if it appears many times in the code
  - #define macros can be used to make the code easier to read
  - Arguments may be of any data type
  - Avoids overhead of a function call

- Macros are inserted into the code by purely replacing the macro body with the defined keyword

- No interpretation occurs

- This leads to several problems as we will see...

- #define - Pitfalls (2)

```
#define times(x,y) (x*y)
int main()
{
  int a, b;
 // read a and b
  ...
  int s = times(a+5,b-2);
  cout << "Result: " << s <<
    endl;
  return 0;
}
```

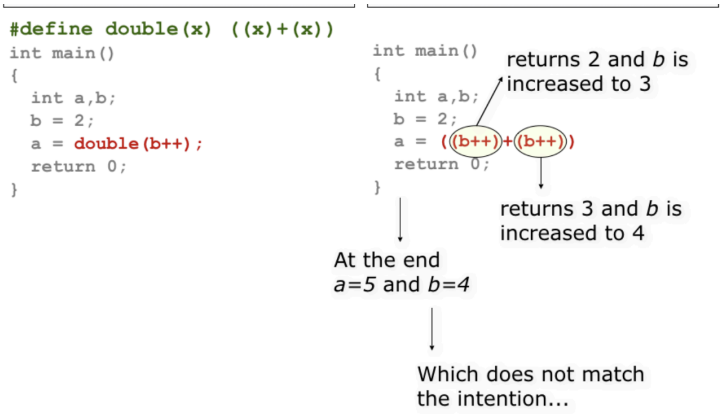
```
int main()
{
  int a, b;
  ...
  int s = (a+5*b-2);
  return 0;
}
```

Which does not match the intention...

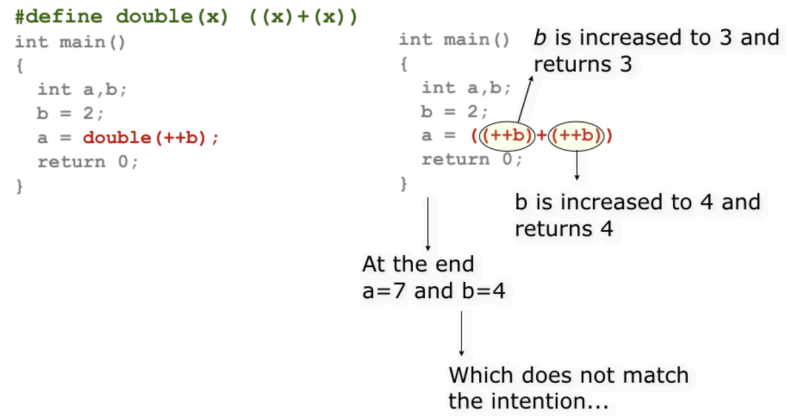
Better use:

#define times(x,y) ((x)*(y))

but...



- #define – Pitfalls (3)



- #define – Pitfalls (4)

Header Files (1/4)

- While *.c* files contain code of OS, *.h* header files contain declarations and definitions used by one or more code files
- They can also include **macros**, for example, **`#define BUFFER_SIZE 4096`**
- It allows programmer to name the constant. BUFFER_SIZE is replaced by 4096 everywhere in the code during compilation

Header Files (2/4)

Macros can have parameters. For example,

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

It allows programmer to write

```
i = max(j, k+1)
```

and get

```
i = (j > k+1 ? j : k+1)
```

Header Files (3/4)

- Headers can also contain conditional compilation:

```
#ifdef X86
intel_int_ack();
#endif
```

- it compiles into a call to the function intel_int_ack **only if the macro X86 is defined**
- Conditional compilation is used to isolate architecture-dependent code. It ensures that certain code is inserted only when the system is compiled on the X86, other code is inserted only when the system is compiled on a SPARC, and so on

A .c file can include zero or more header files using the #include directive.

There are many header files that are common to nearly every .c and are stored in a central directory

Large Programming Projects (1/6)

- To build the system, each *.c* file is compiled into an object file by the C compiler
- Object files, which have the suffix .o, contain binary instructions for the target machine
- They will later be directly executed by the CPU
- There is nothing like Java byte code or Python byte code in the C world

- The first pass of the C compiler is called the **C preprocessor**
  - It reads each .c file
  - Every time it hits a #include directive, it goes and gets the header file named in it and processes it:
    - expands macros
    - handles conditional compilation
    - passes the results to the next pass of the compiler as if they were physically included

- Having to recompile the entire code base every time one file is changed would be unbearable
  However, changing a key header file that is included in thousands of other files does require recompiling those files
  It is possible to keep track of which files need to be recompiled and which don't
  On UNIX systems, there is a program called *make* that reads the *Makefile* - special file that tells which files are dependent on which other files

make sees which object files are needed to build the OS binary

For each file it checks if any of the files it depends on have been modified subsequent to the last time the object file was created

If some of the files were modified, that object file has to be recompiled

When *make* has determined which *.c* files have to be recompiled, it invokes the C compiler to recompile them

In large projects, creating the *Makefile* is error prone, so there are tools that do it automatically

- Once all the .o files are ready, they are passed to a program called **the linker** to combine all of them into a **single executable binary file:**
  - any library functions called are included
  - interfunction references are resolved
  - machine addresses are relocated as need be

- When the linker is finished, the result is an executable program, traditionally called *a.out* on UNIX systems (Fig. 1-30)

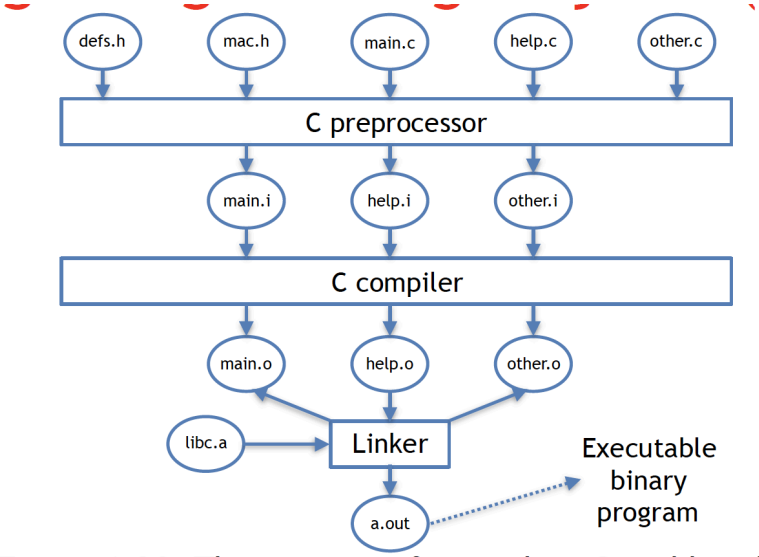


Figure 1-30. The process of compiling C and header files to make an executable.

The Model of Run Time (1/2)

- Once the OS binary has been linked, the computer can be rebooted and the new operating system started
- It may then dynamically load pieces that were not statically included in the binary such as device drivers and file systems
- At run time the operating system may consist of multiple segments:
  - the text (the program code): this segment is normally immutable, not changing during execution
  - the data: it starts out at a certain size and initialized with certain values, but it can change and grow as need be
  - the stack: is initially empty but grows and shrinks as functions are called and returned from
- Often the text segment is placed near the bottom of memory, the data segment just above it, with the ability to grow upward, and the stack segment at a high virtual address, with the ability to grow downward, but different systems work differently
- In all cases, the OS code is directly executed by the hardware, with no interpreter and no just-in-time compilation, as it is normal with Java

Exp.	Explicit	Prefix	Exp.	Explicit	Prefix
10^{-3}	0.001	milli	10^{3}	1,000	Kilo
10^{-6}	0.000001	micro	10^{6}	1,000,000	Mega
10^{-9}	0.000000001	nano	10^{9}	1,000,000,000	Giga
10^{-12}	0.000000000001	pico	10^{12}	1,000,000,000,000	Tera
10^{-15}	0.000000000000001	femto	10^{15}	1,000,000,000,000,000	Peta
10^{-18}	0.000000000000000001	atto	10^{18}	1,000,000,000,000,000,000	Exa
10^{-21}	0.000000000000000000001	zepto	10^{21}	1,000,000,000,000,000,000,000	Zetta
10^{-24}	0.000000000000000000000001	yocto	10^{24}	1,000,000,000,000,000,000,000,000	Yotta

- An array is a collection of data elements that are of the same type (e.g., a

Syntax:

<type> <arrayName>[<array_size>]

Ex. int arr[10];

The array elements are all values of the type **<type>**

The size of the array is indicated by **<array_size>**, the number of elements in the array

<array_size> must be an int constant or a constant expression. Note that an array can have multiple

- dimensions

```
void swap(int *first, int *second);

int main(void) {

    int arr[3];  // input integers
    // Read in three elements.
    printf("Enter three integers: \n");
    scanf("%d", &arr[0]);
    scanf("%d", &arr[1]);
    scanf("%d", &arr[2]);
    if (arr[0] > arr[1]) swap (&arr[0], &arr[1]);
    if (arr[1] > arr[2]) swap (&arr[1], &arr[2]);
    if (arr[0] > arr[1]) swap (&arr[0], &arr[1]);
    printf("The sorted integers are: %d, %d, %d",
              arr[0], arr[1], arr[2]);

    return EXIT_SUCCESS;
}
```

- The following loop initializes the array myList with random values between 0 and 99:

```
#define ARRAY_SIZE 100
…
for (int i = 0; i < ARRAY_SIZE; i++)
{
 myList[i] = rand() % 100;
}
```

Can you copy array using a syntax like this?

```
list = myList;
```

This is not allowed in C. You have to copy individual elements from one array to the other as follows:

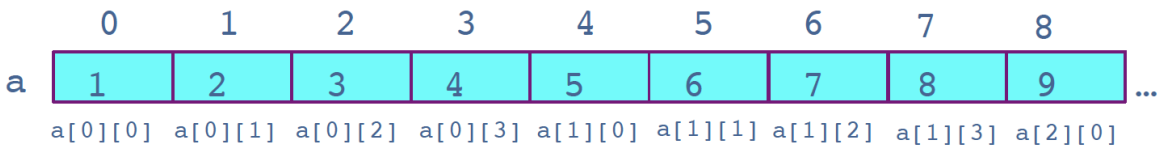
```
for (int i = 0; i < ARRAY_SIZE; i++)
{
  list[i] = myList[i];
}
```

Shifting Elements

```
// Retain the first element
double temp = myList[0];

// Shift elements left
for (int i = 1; i < ARRAY_SIZE; i++)
{
  myList[i - 1] = myList[i];
}

/* Move the first element
to fill in the last position */
myList[ARRAY_SIZE - 1] = temp;
```

```
int a[3][4] =
{
      {1,2,3,4},
      {5,6,7,8},
      {9,10,11,12}
};
```

In memory:



Structures

- A **structure** is a collection of related data items, possibly of different types
- A structure type in C is called **struct**
- A struct is heterogeneous in that it can be composed of data of different types
- In contrast, **array** is homogeneous since it can contain only data of the same type
-
  - Structures hold data that belong **together**
  - Examples:
    - Student record: student id, name, major, gender, start year, …
    - Bank account: account number, name, currency, balance, …
    - Address book: name, address, telephone number, …
  - In database applications, structures are called records
-
  Individual components of a struct type are called **members** (or **fields**)

  Members can be of different types (simple, array or struct)

  A struct is named as a **whole** while individual members are named using **field identifiers**

  Complex data structures can be formed by defining arrays of structs
-

Definition of a structure:

```
struct <struct-type>{
  <type> <identifier_list>;
  <type> <identifier_list>;
  ...
} ;
```

Each identifier defines a *member* of the structure

Example:

```
struct Date {
  int day;
  int month;
  int year;
};
```

The Date structure has 3 members: day, month & year.

- Example:

```
struct StudentInfo{
  int Id;
  int age;
  char Gender;
  double CGA;
};
```

The *StudentInfo* structure has 4 members of different types

- Example:

```
struct StudentGrade{
  char Name[15];
  char Course[9];
  int Lab[5];
  int Homework[3];
  int Exam[2];
};
```

The *StudentGrade* structure has 5 members of different array types

Declaration of a variable of struct type:

```
struct <struct-type> <identifier_list>;
```

Example:

```
struct Date d1, d2;
```

d1 and *d2* are variables of *Date* type

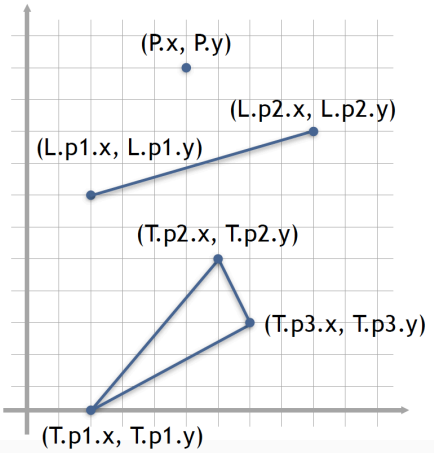
```
int main(void) {
   struct Student s1;

   s1.id = 811;
   strcpy(s1.name, "Stanislav Litvinov");
   strcpy(s1.dept, "MSIT-SE");
   s1.gender = 'M';
   printf("The student is ");
   switch (s1.gender) {
   case 'F': printf("Ms. "); break;
   case 'M': printf("Mr. "); break;
   }
   printf("%s", s1.name);

   return EXIT_SUCCESS;
}
```

- The values contained in one *struct* type variable can be assigned to another variable of the same *struct* type
- Example:

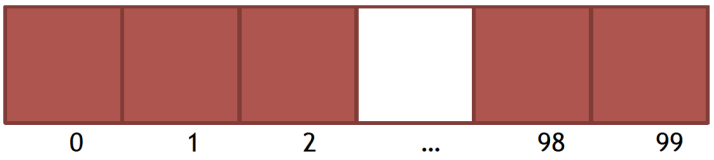
```
...
struct Student s1, s2;
   s1.id = 811;
   strcpy(s1.name, "Stanislav Litvinov");
   strcpy(s1.dept, "MSIT-SE");
   s1.gender = 'M';

   s2 = s1;
```

We can nest structures inside structures. Some examples are:

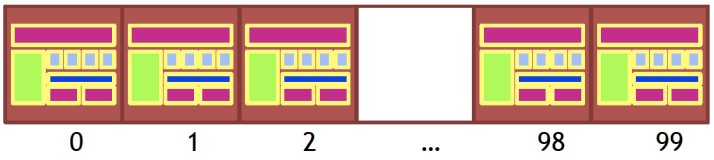
```
struct point {
  double x, y;
};

struct line {
  point p1, p2;
};

struct triangle {
  point p1, p2, p3;
};

struct point P;
struct line L;
struct triangle T;
```



An ordinary array: One type of data



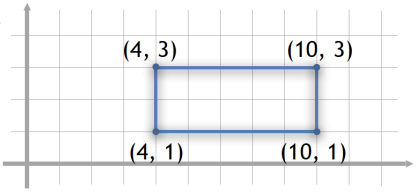
An array of structs: Multiple types of data in each array element



- We often use arrays of structures. Example:

```
struct Student class[100];

class[98].id = 811;
strcpy(class[98].name, "Stanislav Litvinov");
strcpy(class[98].dept, "MSIT-SE");
class[98].gender = 'M';

class[0] = class[98];
```

```
struct rectangle rect;

rect.vertex[0].x = 4;
rect.vertex[0].y = 1;

rect.vertex[1].x = 4;
rect.vertex[1].y = 3;

rect.vertex[2].x = 10;
rect.vertex[2].y = 3;

rect.vertex[3].x = 10;
rect.vertex[3].y = 1;
```



- Function pointer - a variable that holds the memory address of the callable object

- Declaring

Use brackets () to declare function pointer:

`void (*foo)(int)` - function pointer which points to the function that takes int as parameter and returns void

`void *foo(int)` - function which returns void pointer

- Initializing

```
void func(int x);

int main() {
void (*foo)(int);
foo = &func; // prefixing function name with an ampersand
foo = func; // or simply by naming it
}
```

- Invoking
  - Just as if you are calling a function:

```
foo(arg1, arg2);
```

  - Optionally dereference the function pointer before calling the function it points to:

```
(*foo)(arg1, arg2);
```

- Function pointers can be passed to another function as an argument - callback:

```
void download_file(const char *file, void
(*callback_function)(int status_code));
```

- Increasing flexibility of the functions and libraries!
  - qsort Example:

```
void qsort (void *base, size_t nmemb,
size_t size, int(*compar)(const void *,
const void *));
```

Function pointer allows anyone to specify how to sort the elements of the array without writing particular sorting algorithm.

Exercise: write your own **sorter** function and test it with **qsort**:

```
qsort (arg1, arg2, arg3, sorter);
```

- Event programming

Function pointers provide a way of passing instructions on how to do smth when event happens:

```
struct Button;
void button_click(Button *button,
void (*on_click)(Button *button));
```

function sets up a callback that is called when a click is detected on a button