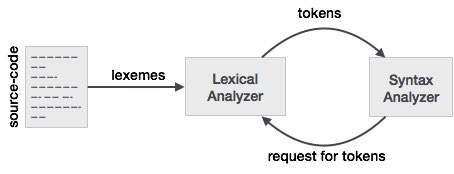
**Lexical analysis** is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.



**Tokens**

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

For example, in C language, the variable declaration line

int value = 100;

contains the tokens:

int (keyword), value (identifier), = (operator), 100 (constant) and ; (symbol).

**Specifications of Tokens**

Let us understand how the language theory undertakes the following terms:

**Alphabets**

Any finite set of symbols {0,1} is a set of binary alphabets, {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} is a set of Hexadecimal alphabets, {a-z, A-Z} is a set of English language alphabets.

**Strings**

Any finite sequence of alphabets is called a string. Length of the string is the total number of occurrence of alphabets, e.g., the length of the string tutorialspoint is 14 and is denoted by |tutorialspoint| = 14. A string having no alphabets, i.e. a string of zero length is known as an empty string and is denoted by ε (epsilon).

**Special Symbols**

A typical high-level language contains the following symbols:-

|  |  |
| --- | --- |
| Arithmetic Symbols | Addition(+), Subtraction(-), Modulo(%), Multiplication(\*), Division(/) |
| Punctuation | Comma(,), Semicolon(;), Dot(.), Arrow(->) |
| Assignment | = |
| Special Assignment | +=, /=, \*=, -= |
| Comparison | ==, !=, <, <=, >, >= |
| Preprocessor | # |
| Location Specifier | & |
| Logical | &, &&, |, ||, ! |
| Shift Operator | >>, >>>, <<, <<< |

**Language:**

A language is considered as a finite set of strings over some finite set of alphabets. Computer languages are considered as finite sets, and mathematically set operations can be performed on them. Finite languages can be described by means of regular expressions.

**Longest Match Rule**

When the lexical analyzer read the source-code, it scans the code letter by letter; and when it encounters a whitespace, operator symbol, or special symbols, it decides that a word is completed.

**For example:**

int intvalue;

While scanning both lexemes till ‘int’, the lexical analyzer cannot determine whether it is a keyword *int* or the initials of identifier int value.

The Longest Match Rule states that the lexeme scanned should be determined based on the longest match among all the tokens available.

The lexical analyzer also follows **rule priority** where a reserved word, e.g., a keyword, of a language is given priority over user input. That is, if the lexical analyzer finds a lexeme that matches with any existing reserved word, it should generate an error.

# Compiler Design - Semantic Analysis

We have learnt how a parser constructs parse trees in the syntax analysis phase. The plain parse-tree constructed in that phase is generally of no use for a compiler, as it does not carry any information of how to evaluate the tree. The productions of context-free grammar, which makes the rules of the language, do not accommodate how to interpret them.

For example

E → E + T

The above CFG production has no semantic rule associated with it, and it cannot help in making any sense of the production.

Semantics

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

CFG + semantic rules = Syntax Directed Definitions

For example:

int a = “value”;

should not issue an error in lexical and syntax analysis phase, as it is lexically and structurally correct, but it should generate a semantic error as the type of the assignment differs. These rules are set by the grammar of the language and evaluated in semantic analysis. The following tasks should be performed in semantic analysis:

* Scope resolution
* Type checking
* Array-bound checking

**Semantic Errors**

We have mentioned some of the semantics errors that the semantic analyzer is expected to recognize:

* Type mismatch
* Undeclared variable
* Reserved identifier misuse.
* Multiple declaration of variable in a scope.
* Accessing an out of scope variable.
* Actual and formal parameter mismatch.

**Attribute Grammar**

Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information. Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.

Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language. Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

**Example:**

E → E + T { E.value = E.value + T.value }

The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.

Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions. Based on the way the attributes get their values, they can be broadly divided into two categories : synthesized attributes and inherited attributes.

**Synthesized attributes**

These attributes get values from the attribute values of their child nodes. To illustrate, assume the following production**: S->ABC**

If S is taking values from its child nodes (A,B,C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

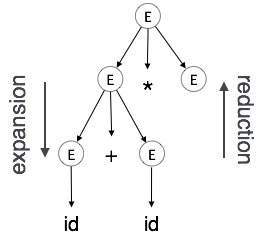
As in our previous example (E → E + T), the parent node E gets its value from its child node. Synthesized attributes never take values from their parent nodes or any sibling nodes.

Inherited attributes

In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production, **: S->ABC**

A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

**Expansion** : When a non-terminal is expanded to terminals as per a grammatical rule



**Reduction** : When a terminal is reduced to its corresponding non-terminal according to grammar rules. Syntax trees are parsed top-down and left to right. Whenever reduction occurs, we apply its corresponding semantic rules (actions).

Semantic analysis uses Syntax Directed Translations to perform the above tasks.

Semantic analyzer receives AST (Abstract Syntax Tree) from its previous stage (syntax analysis).

Semantic analyzer attaches attribute information with AST, which are called Attributed AST.

Attributes are two tuple value, <attribute name, attribute value>

For example:

int value = 5;

<type, “integer”>

<presentvalue, “5”>

For every production, we attach a semantic rule.

**S-attributed SDT**

|  |  |
| --- | --- |
| If an SDT uses only synthesized attributes, it is called as S-attributed SDT. These attributes are evaluated using S-attributed SDTs that have their semantic actions written after the production (right hand side). | S-attributed SDT |

As depicted above, attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

**L-attributed SDT**

|  |  |
| --- | --- |
| This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings.  In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes. As in the following production  S->ABC  S can take values from A, B, and C (synthesized). A can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right. | L-attributed SDT |

Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.

We may conclude that if a definition is S-attributed, then it is also L-attributed as L-attributed definition encloses S-attributed definitions

**What is Code Optimization?**

 Code Optimization is an approach for enhancing the performance of the code by improving it through the elimination of unwanted code lines and by rearranging the statements of the code in a manner that increases the execution speed of the code without any wastage of the computer resources.

**Advantages of Code Optimization-**

 Optimized code has faster execution speed

* Optimized code utilizes the memory efficiently
* Optimized code gives better performance

**Techniques for Code Optimization-**

 Compile Time Evaluation

1. Common sub-expression elimination
2. Dead Code Elimination
3. Code Movement
4. Strength Reduction

**1. Compile Time Evaluation-**

Two techniques that falls under compile time evaluation are-

**A) Constant folding-**

 As the name suggests, this technique involves folding the constants by evaluating the expressions that involves the operands having constant values at the compile time.

* **Example-**

Circumference of circle  = (22/7) x Diameter

Here, this technique will evaluate the expression 22/7 and will replace it with its result 3.14 at the compile time which will save the time during the program execution.

**B) Constant Propagation-**

 In this technique, if some variable has been assigned some constant value, then it replaces that variable with its constant value in the further program wherever it has been used during compilation, provided that its value does not get alter in between.

* **Example-**

pi = 3.14

radius = 10

Area of circle = pi x radius x radius

Here, this technique will substitute the value of the variables ‘pi’ and ‘radius’ at the compile time and then it will evaluate the expression 3.14 x 10 x 10 at the compile time which will save the time during the program execution.

**2. Common sub-expression elimination-**

 The expression which has been already computed before and appears again and again in the code for computation is known as a common sub-expression.

As the name suggests, this technique involves eliminating the redundant expressions to avoid their computation again and again. The already computed result is used in the further program wherever its required.

**Example-**

|  |  |
| --- | --- |
| **Code before Optimization** | **Code after Optimization** |
| S1 = 4 x i  S2 = a[S1]  S3 = 4 x j  S4 = 4 x i  **// Redundant  Expression**  S5 = n  S6 = b[S4] + S5 | S1 = 4 x i  S2 = a[S1]  S3 = 4 x j  S5 = n  S6 = b[S1] + S5 |

**3. Code Movement-**

 As the name suggests, this technique involves the movement of the code where the code is moved out of the loop if it does not matter whether it is present inside the loop or it is present outside the loop.

Such a code unnecessarily gets executed again and again with each iteration of the loop, thus wasting the time during the program execution.

**Example-**

|  |  |
| --- | --- |
| **Code before Optimization** | **Code after Optimization** |
| for ( int j = 0 ; j < n ; j ++)  {  x = y + z ;  a[j] = 6 x j;  } | x = y + z ;  for ( int j = 0 ; j < n ; j ++)  {  a[j] = 6 x j;  } |

**4. Dead code elimination-**

 As the name suggests, this technique involves eliminating the dead code where those statements from the code are eliminated which either never executes or are not reachable or even if they get execute, their output is never utilized.

**Example-**

|  |  |
| --- | --- |
| **Code before Optimization** | **Code after Optimization** |
| i = 0 ;  if (i == 1)  {  a = x + 5 ;  } | i = 0 ; |

**5. Strength reduction-**

As the name suggests, this technique involves reducing the strength of the expressions by replacing the expensive and costly operators with the simple and cheaper ones.

**Example-**

|  |  |
| --- | --- |
| **Code before Optimization** | **Code after Optimization** |
| B = A x 2 | B = A + A |

Here, the expression “A x 2” has been replaced with the expression “A + A” because the cost of multiplication operator is higher than the cost of addition operator.

The loader is a program which accepts an object code and prepare them for execution. The loader's target language is machine language, its source language is nearly machine langugae. loading is ultimately bound with storage management function of operating systems and is usually performed later than assembly or compilation .the period of executions of user program is called execution time .the period of translating user's source program is called assembly or compile time. load time refers to the period of loading and preparing object program for execution.

**Loading schemes: 1.Absolute loader. 2.Relocating loader. 3.Direct linking loader. 4.Dynamic Loading. 5.Dynamic linking.**

**Absolute loader**: The task of an absolute loader is virtually trivial.The loader simply accepts machine language code and places it into main memory specified by the assembler.

**Relocating loader:** The task of relocating loader is to avoid reassembling of of all subroutines when a subroutine is changed and to perform tasks of allocation and linking for programmer.

**Dynamic loading**: In order to overlay structure to work it is necessary for the module loader to load the various procedures as they are needed.There are many binders capable of processing and allocating overlay structure.the portion of the laoder that actually intercepts calls and loads necessary procedure is called overlay supervisor of simplly flipper.this overall scheme is called dynamic loading or load on call.

**Dynamic linking:** This is mechanism by which loading and linking of external references are postponed until execution time.This was made to sort out disadvantage of previous loading schemes like subroutine is referenced and never executed

**Implementation of Absolute loader:** The four loader functions are performed as following: 1.Allocation- By programmer 2.Linking- By programmer 3.Relocation- By assembler 4.Loading- By loader

**Loader Schemes:**

Based on the various functionalities of loader, there are various types of

loaders:

**1) “compile and go” loader:** in this type of loader, the instruction is read

line by line, its machine code is obtained and it is directly put in the main

memory at some known address. That means the assembler runs in one

part of memory and the assembled machine instructions and data is

directly put into their assigned memory locations. After completion of

assembly process, assign starting address of the program to the location

counter. The typical example is WATFOR-77, it’s a FORTRAN compiler

which uses such “load and go” scheme. This loading scheme is also

called as “assemble and go”.

**Advantages:**

• This scheme is simple to implement. Because assembler is placed at one

part of the memory and loader simply loads assembled machine

instructions into the memory.

**Disadvantages:**

• In this scheme some portion of memory is occupied by assembler which

is simply a wastage of memory. As this scheme is combination of

assembler and loader activities, this combination program occupies large

block of memory.

• There is no production of .obj file, the source code is directly converted

to executable form. Hence even though there is no modification in the

source program it needs to be assembled and executed each time, which

then becomes a time consuming activity.

• It cannot handle multiple source programs or multiple programs written

in different languages. This is because assembler can translate one source

language to other target language.

• For a programmer it is very difficult to make an orderly modulator

program and also it becomes difficult to maintain such program, and the

“compile and go” loader cannot handle such programs.

• The execution time will be more in this scheme as every time program

is assembled and then executed.

**Complier and GO to Scheme Dia**

|  |  |
| --- | --- |
| User program | Compile & go loader |

|  |
| --- |
|  |
|  |
| Compile & go loader |
|  |
| Execute code of user program |
|  |

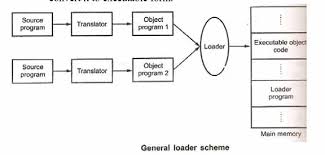
2) **General Loader Scheme:** in this loader scheme, the source program is converted to object program by some translator (assembler). The loader accepts these object modules and puts machine instruction and data in an executable form at their assigned memory. The loader occupies some portion of main memory.

**Advantages:**

• The program need not be retranslated each time while running it. This is because initially when source program gets executed an object program gets generated. Of program is not modified, then loader can make use of this object program to convert it to executable form.

• There is no wastage of memory, because assembler is not placed in the memory, instead of it, loader occupies some portion of the memory. And size of loader is smaller than assembler, so more memory is available to the user.

• It is possible to write source program with multiple programs and multiple languages, because the source programs are first converted to object programs always, and loader accepts these object modules to convert it to executable form.



**3) Absolute Loader:** Absolute loader is a kind of loader in which relocated object files are created, loader accepts these files and places them at specified locations in the memory. This type of loader is called

absolute because no relocation information is needed; rather it is obtained from the programmer or assembler. The starting address of every module is known to the programmer, this corresponding starting address is stored in the object file, then task of loader becomes very simple and that is to simply place the executable form of the machine instructions at the locations mentioned in the object file. In this scheme, the programmer or assembler should have knowledge of memory management. The resolution of external references or linking of different subroutines are the issues which need to be handled by the programmer. The programmer should take care of two things: first thing is : specification of starting address of each module to be used. If some modification is done in some module then the length of that module may vary. This causes a change in the starting address of immediate next . modules, its then the programmer's duty to make necessary changes in the starting addresses of respective modules.

Second thing is ,while branching from one segment to another the absolute starting address of respective module is to be known by the programmer so that such address can be specified at respective JMP instruction. For example

Line no

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | Main | START | 1000 |  |
|  |  |  |  |  |
|  |  |  |  |  |
| 1 |  | JMP | 5000 |  |
| 16 |  | STORE  END | : INSTRUCTION AT LOCATION 2000 | |
| 1 |  | SUM | START 1000 |  |
| 2 |  |  |  |  |
| 20 |  | JMP | 2000 |  |
| 21 |  | END |  |  |

In this example there are two segments, which are interdependent. At line number 1 the assembler directive START specifies the physical starting address that can be used during the execution of the first segment MAIN.

Then at line number 15 the JMP instruction is given which specifies the physical starting address that can be used by the second segment. The assembler creates the object codes for these two segments by considering the stating addresses of these two segments. During the execution, the first segment will be loaded at address 1000 and second segment will be loaded at address 5000 as specified by the programmer. Thus the problem of linking is manually solved by the programmer itself by taking care of the mutually dependant dresses. As you can notice that the control is correctly transferred to the address 5000 for invoking the other segment, and after that at line number 20 the JMP instruction transfers the control to the location 2000, necessarily at location 2000 the instruction STORE of line number 16 is present. Thus resolution of mutual references and linking is done by the programmer. The task of assembler is to create the object codes for the above segments and along with the information such as starting address of the memory where actually the object code can be placed at the time of execution. The absolute loader accepts these object modules from assembler and by reading the information about their starting addresses, it will actually place (load) them in the memory at specified addresses.

The entire process is modeled in the following figure. i.e. **Process of Absolute Loading**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | Absolute Loader |  |
| SEGMENT1 |  | ASSEMBLER |  | Object code |  |
|  | Starting address | Object code for segment1 |
| ….. |  |  |  |  |  |
| SEGMENT2 |  | ASSEMBLER |  | Object code |  |
|  | Starting address | Object code for segment2 |
| ….. |  |  |  |  |  |
| SEGMENT n |  | ASSEMBLER |  | Object code |  |
|  | Starting address | Object code for segment n |

‘ Absolute Loader Main memory

Thus the absolute loader is simple to implement in this scheme-

l) Allocation is done by either programmer or assembler

2) Linking is done by the programmer or assembler

3) Resolution is done by assembler

4) Simply loading is done by the loader

As the name suggests, no relocation information is needed, if at all it is required then that task can be done by either a programmer or assembler

**Advantages:**

1. It is simple to implement

2. This scheme allows multiple programs or the source programs written different languages. If there are multiple programs written in different languages then the respective language assembler will convert it to the language and a common object file can be prepared with all the ad resolution.

3. The task of loader becomes simpler as it simply obeys the instruction regarding where to place the object code in the main memory.

4. The process of execution is efficient.

**Disadvantages:**

1. In this scheme it is the programmer's duty to adjust all the inter segment addresses and manually do the linking activity. For that, it is necessary for a programmer to know the memory management.

If at all any modification is done the some segments, the starting addresses of immediate next segments may get changed, the programmer has to take care of this issue and he needs to update the corresponding starting addresses on any modification in the source.

**Algorithm for absolute Loader**

Input: Object codes and starting address of program segments.

Output: An executable code for corresponding source program. This executable code is to be placed in the main memory

**Method: Begin**

For each program segment

do Begin

Read the first line from object module to obtain information about memory location. The starting address say S in corresponding object module is the memory location where executable code is to be placed.

Hence

Memory\_location = S

Line counter = 1; as it is first line While (! end of file)

For the curent object code

do Begin

1. Read next line

2. Write line into location S

3. S = S + 1

4. Line counter Line counter + 1

**Subroutine Linkage:** To understand the concept of subroutine linkages, first consider the following scenario:

"In Program A a call to subroutine B is made. The subroutine B is not written in the program segment of A, rather B is defined in some another program segment C"

Nothing is wrong in it. But from assembler's point of view while generating the code for B, as B is not defined in the segment A, the assembler can not find the value of this symbolic reference and hence it will declare it as an error. To overcome problem, there should be some

mechanism by which the assembler should be explicitly informed that segment B is really defined in some other segment C. Therefore whenever segment B is used in segment A and if at all B is defined in C, then B must -be declared as an external routine in A.

To declare such subroutine as external, we can use the assembler directive EXT. Thus the statement such as EXT B should be added at the beginning of the segment A. This actually helps to inform assembler that B is defined somewhere else. Similarly, if one subroutine or a variable is defined in the current segment and can be referred by other segments then those should be declared by using pseudo-ops INT. Thereby the assembler could inform loader that these are the subroutines or variables used by other segments. This overall process of establishing the relations between the subroutines can be conceptually called a\_ subroutine linkage.

For example

MAIN START

EXT B

.

.

.

CALL B

.

.

END

B START

.

.

RET

END

At the beginning of the MAIN the subroutine B is declared as external. When a call to subroutine B is made, before making the unconditional jump, the current content of the program counter should be stored in the system stack maintained internally. Similarly while returning from the subroutine B (at RET) the pop is performed to restore the program

counter of caller routine with the address of next instruction to be executed.

**Concept of relocations:**

Relocation is the process of updating the addresses used in the address sensitive instructions of a program. It is necessary that such a modification should help to execute the program from designated area of the memory.

The assembler generates the object code. This object code gets executed after loading at storage locations. The addresses of such object code will get specified only after the assembly process is over. Therefore, after loading,

**address of object code = Mere address of object code + relocation constant.**

There are two types of addresses being generated: Absolute address and, relative address. The absolute address can be directly used to map the object code in the main memory. Whereas the relative address is only after the addition of relocation constant to the object code address. This kind of adjustment needs to be done in case of relative address before actual execution of the code. The typical example of relative reference is :

addresses of the symbols defined in the Label field, addresses of the data which is defined by the assembler directive, literals, redefinable symbols.

Similarly, the typical example of absolute address is the constants which are generated by assembler are absolute.

The assembler calculates which addresses are absolute and which addresses are relative during the assembly process. During the assembly process the assembler calculates the address with the help of simple expressions.

For example

LOADA(X)+5

The expression A(X) means the address of variable X. The meaning of the above instruction is that loading of the contents of memory location which is 5 more than the address of variable X. Suppose if the address of X is 50 then by above command we try to get the memory location

50+5=55. Therefore as the address of variable X is relative A(X) + 5 is also relative. To calculate the relative addresses the simple expressions are allowed. It is expected that the expression should possess at the most addition and multiplication operations. A simple exercise can be carried out to determine whether the given address is absolute or relative. In the expression if the address is absolute then put 0 over there and if address is relative then put lover there. The expression then gets transformed to sum of O's and l's. If the resultant value of the expression is 0 then expression is absolute. And if the resultant value of the expression is 1 then the expression is relative. If the resultant is other than 0 or 1then the

expression is illegal. For example:

|  |  |  |
| --- | --- | --- |
| **EXPRESSION** | **COMPUTATION** | **RELOCATION ATTRIBUTE** |
| A-B | 1-1 =0 | ABSOLUTE |
| A+B-C | 1+1-1=1 | RELATIVE |
| A-B+5 | 1-1+0=0 | ABSOLUTE |
| A+B | 1+1=2 | IILEGAL |

In the above expression the A, Band C are the variable names. The assembler is to c0l1sider the relocation attribute and adjust the object code by relocation constant. Assembler is then responsible to convey the information loading of object code to the loader. Let us now see how

assembler generates code using relocation information.

**Direct Linking Loaders**

In the above expression the A, Band C are the variable names. The

assembler is to c0l1sider the relocation attribute and adjust the object

code by relocation constant. Assembler is then responsible to convey the

information loading of object code to the loader. Let us now see how

assembler generates code using relocation information.

**Direct Linking Loaders**

The direct linking loader is the most common type of loader. This type of loader is a relocatable loader. The loader can not have the direct access to the source code. And to place the object code in the memory there are two situations: either the address of the object code could be absolute which then can be directly placed at the specified location or the address

can be relative. If at all the address is relative then it is the assembler who informs the loader about the relative addresses.

The assembler should give the following information to the loader

1) The length of the object code segment

2) The list of all the symbols which are not defined 111 the current segment but can be used in the current segment.

3) The list of all the symbols which are defined in the current segment but can be referred by the other segments.

The list of symbols which are not defined in the current segment but can be used in the current segment are stored in a data structure called USE table. The USE table holds the information such as name of the symbol, address, address relativity.

The list of symbols which are defined in the current segment and can be referred by the other segments are stored in a data structure called DEFINITION table. The definition table holds the information such as symbol, address.

**Overlay Structures and Dynamic Loading:**

Sometimes a program may require more storage space than the available one Execution of such program can be possible if all the segments are not required simultaneously to be present in the main memory. In such situations only those segments are resident in the memory that are actually needed at the time of execution But the question arises what will happen if the required segment is not present in the memory? Naturally the execution process will be delayed until the required segment gets loaded in the memory. The overall effect of this is efficiency of execution process gets degraded. The efficiency can then be improved by carefully selecting all the interdependent segments. Of course the assembler can not do this task. Only the user can specify such dependencies. The inter dependency of the segments can be specified by a tree like structure called static overlay structures. The overlay structure contain multiple root/nodes and edges. Each node represents the segment. The specification of required amount of memory is also essential in this structure. The two segments can lie simultaneously in the main memory if they are on the same path. Let us take an example to understand the concept. Various segments along with their memory requirements is as

shown below.

S1 (16K)

S3 (8k)

S2 (14k)

S4 (20k)

S6(16K)

S5 (12K)

S7(16K)

S8(12K)

S9(18K)

**Automatic Library Search:**

Previously, the library routines were available in absolute code but now the library routines are provided in relocated form that ultimately reduces their size on the disk, which in turn increases the memory utilization. At

execution time certain library routines may be needed. Keeping track of which library routines are required and how much storage is required by these routines, if at all is done by an assembler itself then the activity of

automatic library search becomes simpler and effective. The library routines can also make an external call to other routines. The idea is to make a list of such calls made by the routines. And if such list is made available to the linker then linker can efficiently find the set of required routines and can link the references accordingly.

For an efficient search of library routines it desirable to store all the calling routines first and then the called routines. This avoids wastage of time due to winding and rewinding. For efficient automated search of

library routines even the dictionary of such routines can be maintained. A table containing the names of library routines and the addresses where they are actually located in relocatable form is prepared with the help of

translator and such table is submitted to the linker. Such a table is **called subroutine directory**. Even if these routines have made any external calls the -information about it is also given in subroutine directory. The linker

searches the subroutine directory, finds the address of desired library routine (the address where the routine is stored in relocated form).Then linker prepares a load module appending the user program and necessary library routines by doing the necessary relocation. If the library routine contains the external calls then the linker searches the subroutine directory finds the address of such external calls, prepares the load module by resolving the external references. Linkage Editor: The execution of any program needs four basic functionalities and those are allocation, relocation, linking and loading. As we have also seen in direct linking loader for execution of any program each time these four functionalities need to be performed. But performing all these functionalities each time is time and space consuming task. Moreover if the program contains many subroutines or functions and the program needs to be executed repeatedly then this activity becomes annoyingly complex .Each time for execution of a program, the allocation, relocation linking and -loading needs to be done. Now doing these activities each time increases the time and space complexity. Actually, there is no need to redo all these four activities each time. Instead, if the results of some of these activities are stored in a file then that file can be used by other activities. And performing allocation, relocation, linking and loading can be avoided each time. The idea is to separate out these activities in separate groups. Thus dividing the essential four functions in groups reduces the overall time complexity of loading process. The program which performs allocation, relocation and linking is called binder. The binder performs relocation, creates linked executable text and stores this text in a file in some systematic manner. Such kind of module prepared by the binder execution is called

load module. This load module can then be actually loaded in the main memory by the loader. This loader is also called as module loader. If the binder can produce the exact replica of executable code in the load module then the module loader simply loads this file into the main memory which ultimately reduces the overall time complexity. But in this process the binder should knew the current positions of the main memory.

Even though the binder knew the main memory locations this is not the only thing which is sufficient. In multiprogramming environment, the region of main memory available for loading the program is decided by

the host operating system. The binder should also know which memory area is allocated to the loading program and it should modify the relocation information accordingly. The binder which performs the linking function and produces adequate information about allocation and relocation and writes this information along with the program code in the file is called linkage editor. The module loader then accepts this rile as input, reads the information stored in and based on this information about allocation and relocation it performs the task of loading in the main memory. Even though the program is repeatedly executed the linking is done only once. Moreover, the flexibility of allocation and relocation helps efficient utilization of the main memory.

Direct linking: As we have seen in overlay structure certain selective subroutines can be resident in the memory. That means it is not necessary to resident all the subroutines in the memory for all the time. Only necessary routines can be present in the main memory and during execution the required subroutines can be loaded in the memory. This process of postponing linking and loading of external reference until execution is called dynamic linking. For example suppose the subroutine main calls A,B,C,D then it is not desirable to load A,B,C and D along with the main in the memory. Whether A, B, C or D is called by the main or not will be known only at the time of execution. Hence keeping these routines already before is really not needed. As the subroutines get

executed when the program runs. Also the linking of all the subroutines has to be performed. And the code of all the subroutines remains resident in the main memory. As a result of all this is that memory gets occupied

unnecessarily. Typically 'error routines' are such routines which can be invoked rarely. Then one can postpone the loading of these routines during the execution. If linking and loading of such rarely invoked external references could be postponed until the execution time when it was found to be absolutely necessary, then it increases the efficiency of overhead of the loader. In dynamic linking, the binder first prepares a load module in which along with program code the allocation and relocation information is stored. The loader simply loads the main module in the main memory. If any external ·reference to a subroutine comes, then the execution is suspended for a while, the loader brings the required subroutine in the main memory and then the execution process is resumed. Thus dynamic linking both the loading and linking is done

dynamically.

**Advantages**

1. The overhead on the loader is reduced. The required subroutine will be load in the main memory only at the time of execution.

2. The system can be dynamically reconfigured.

**Disadvantages** The linking and loading need to be postponed until the execution. During the execution if at all any subroutine is needed then the process of execution needs to be suspended until the required subroutine

gets loaded in the main memory.

**Bootstrap Loader**: As we turn on the computer there is nothing meaningful in the main memory (RAM). A small program is written and stored in the ROM. This program initially loads the operating system from secondary storage to main memory. The operating system then takes the overall control. This program which is responsible for booting up the system is **called bootstrap loader**. This is the program which must be executed first when the system is first powered on. If the program starts from the location x then to execute this program the program counter of this machine should be loaded with the value x. Thus the task of setting

the initial value of the program counter is to be done by machine hardware. The bootstrap loader is a very small program which is to be fitted in the ROM. The task of bootstrap loader is to load the necessary portion of the operating system in the main memory .The initial address at which the bootstrap loader is to be loaded is generally the lowest (may be at 0th location) or the highest location. .

Concept of Linking: As we have discussed earlier, the execution of

program can be done with the help of following steps

1. Translation of the program(done by assembler or compiler)

2. Linking of the program with all other programs which are needed for

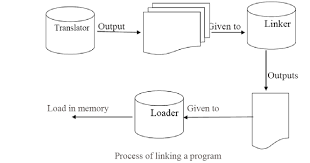
execution. This also involves preparation of a program called load module.

3. Loading of the load module prepared by linker to some specified

memory location.

The output of translator is a program called object module. The linker processes these object modules binds with necessary library routines and prepares a ready to execute program. Such a program is called binary

program. The "binary program also contains some necessary information about allocation and relocation. The loader then load s this program into memory for execution purpose.



Various tasks of linker are -

1. Prepare a single load module and adjust all the addresses and subroutine references with respect to the offset location.

2. To prepare a load module concatenate all the object modules and adjust all the operand address references as well as external references to the offset location.

3. At correct locations in the load module, copy the binary machine instructions and constant data in order to prepare ready to execute module.

The linking process is performed in two passes. Two passes are necessary because the linker may encounter a forward reference before knowing its address. So it is necessary to scan all the DEFINITION and USE table at least once. Linker then builds the Global symbol table with the help of USE and DEFINITION table. In Global symbol table name of each externally referenced symbol is included along with its address relative to beginning of the load module. And during pass 2, the addresses of external references are replaced by obtaining the addresses from global symbol table.

# Memory Management

Main Memory refers to a physical memory that is the internal memory to the computer. The word main is used to distinguish it from external mass storage devices such as disk drives. Main memory is also known as RAM. The computer is able to change only data that is in main memory. Therefore, every program we execute and every file we access must be copied from a storage device into main memory.

All the programs are loaded in the main memeory for execution. Sometimes complete program is loaded into the memory, but some times a certain part or routine of the program is loaded into the main memory only when it is called by the program, this mechanism is called **Dynamic Loading**, this enhance the performance.

Also, at times one program is dependent on some other program. In such a case, rather than loading all the dependent programs, CPU links the dependent programs to the main executing program when its required. This mechanism is known as **Dynamic Linking**.

## Swapping

A process needs to be in memory for execution. But sometimes there is not enough main memory to hold all the currently active processes in a timesharing system. So, excess process are kept on disk and brought in to run dynamically. Swapping is the process of bringing in each process in main memory, running it for a while and then putting it back to the disk.

## Contiguous Memory Allocation

In contiguous memory allocation each process is contained in a single contiguous block of memory. Memory is divided into several fixed size partitions. Each partition contains exactly one process. When a partition is free, a process is selected from the input queue and loaded into it. The free blocks of memory are known as *holes*. The set of holes is searched to determine which hole is best to allocate.

## Memory Protection

Memory protection is a phenomenon by which we control memory access rights on a computer. The main aim of it is to prevent a process from accessing memory that has not been allocated to it. Hence prevents a bug within a process from affecting other processes, or the operating system itself, and instead results in a segmentation fault or storage violation exception being sent to the disturbing process, generally killing of process.

## Memory Allocation

Memory allocation is a process by which computer programs are assigned memory or space. It is of three types :

1. **First Fit:**

The first hole that is big enough is allocated to program.

1. **Best Fit:**

The smallest hole that is big enough is allocated to program.

1. **Worst Fit:**

The largest hole that is big enough is allocated to program.

## Fragmentation

Fragmentation occurs in a dynamic memory allocation system when most of the free blocks are too small to satisfy any request. It is generally termed as inability to use the available memory.

In such situation processes are loaded and removed from the memory. As a result of this, free holes exists to satisfy a request but is non contiguous i.e. the memory is fragmented into large no. Of small holes. This phenomenon is known as **External Fragmentation.**

Also, at times the physical memory is broken into fixed size blocks and memory is allocated in unit of block sizes. The memory allocated to a space may be slightly larger than the requested memory. The difference between allocated and required memory is known as **Internal fragmentation** i.e. the memory that is internal to a partition but is of no use.

## Paging

A solution to fragmentation problem is Paging. Paging is a memory management mechanism that allows the physical address space of a process to be non-contagious. Here physical memory is divided into blocks of equal size called **Pages**. The pages belonging to a certain process are loaded into available memory frames.

### Page Table

A Page Table is the data structure used by a virtual memory system in a computer operating system to store the mapping between *virtual address* and *physical addresses.*

Virtual address is also known as Logical address and is generated by the CPU. While Physical address is the address that actually exists on memory.

## Segmentation

Segmentation is another memory management scheme that supports the user-view of memory. Segmentation allows breaking of the virtual address space of a single process into segments that may be placed in non-contiguous areas of physical memory.

### Segmentation with Paging

Both paging and segmentation have their advantages and disadvantages, it is better to combine these two schemes to improve on each. The combined scheme is known as 'Page the Elements'. Each segment in this scheme is divided into pages and each segment is maintained in a page table. So the logical address is divided into following 3 parts :

* Segment numbers(S)
* Page number (P)
* The displacement or offset number (D)

# What is Virtual Memory?

Virtual Memory is a space where large programs can store themselves in form of pages while their execution and only the required pages or portions of processes are loaded into the main memory. This technique is useful as large virtual memory is provided for user programs when a very small physical memory is there.

In real scenarios, most processes never need all their pages at once, for following reasons :

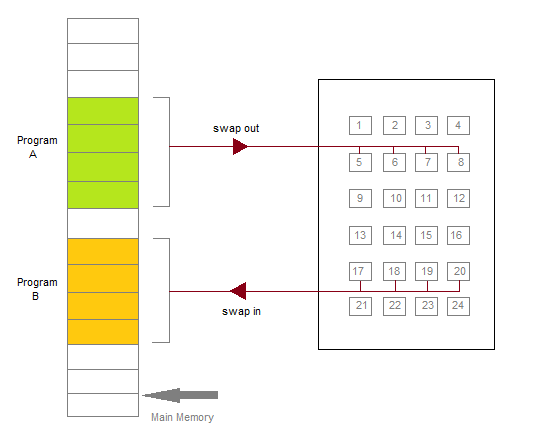
* Error handling code is not needed unless that specific error occurs, some of which are quite rare.
* Arrays are often over-sized for worst-case scenarios, and only a small fraction of the arrays are actually used in practice.
* Certain features of certain programs are rarely used.

### Benefits of having Virtual Memory

1. Large programs can be written, as virtual space available is huge compared to physical memory.
2. Less I/O required, leads to faster and easy swapping of processes.
3. More physical memory available, as programs are stored on virtual memory, so they occupy very less space on actual physical memory.

## What is Demand Paging?

The basic idea behind demand paging is that when a process is swapped in, its pages are not swapped in all at once. Rather they are swapped in only when the process needs them(On demand). This is termed as lazy swapper, although a pager is a more accurate term.



Initially only those pages are loaded which will be required the process immediately.

The pages that are not moved into the memory, are marked as invalid in the page table. For an invalid entry the rest of the table is empty. In case of pages that are loaded in the memory, they are marked as valid along with the information about where to find the swapped out page.

When the process requires any of the page that is not loaded into the memory, a page fault trap is triggered and following steps are followed,

1. The memory address which is requested by the process is first checked, to verify the request made by the process.
2. If its found to be invalid, the process is terminated.
3. In case the request by the process is valid, a free frame is located, possibly from a free-frame list, where the required page will be moved.
4. A new operation is scheduled to move the necessary page from disk to the specified memory location. ( This will usually block the process on an I/O wait, allowing some other process to use the CPU in the meantime. )
5. When the I/O operation is complete, the process's page table is updated with the new frame number, and the invalid bit is changed to valid.
6. The instruction that caused the page fault must now be restarted from the beginning.

There are cases when no pages are loaded into the memory initially, pages are only loaded when demanded by the process by generating page faults. This is called **Pure Demand Paging**.

The only major issue with Demand Paging is, after a new page is loaded, the process starts execution from the beginning. Its is not a big issue for small programs, but for larger programs it affects performance drastically.

## Page Replacement

As studied in Demand Paging, only certain pages of a process are loaded initially into the memory. This allows us to get more number of processes into the memory at the same time. but what happens when a process requests for more pages and no free memory is available to bring them in. Following steps can be taken to deal with this problem :

1. Put the process in the wait queue, until any other process finishes its execution thereby freeing frames.
2. Or, remove some other process completely from the memory to free frames.
3. Or, find some pages that are not being used right now, move them to the disk to get free frames. This technique is called **Page replacement** and is most commonly used. We have some great algorithms to carry on page replacement efficiently.

Basic Page Replacement

* Find the location of the page requested by ongoing process on the disk.
* Find a free frame. If there is a free frame, use it. If there is no free frame, use a page-replacement algorithm to select any existing frame to be replaced, such frame is known as **victim frame**.
* Write the victim frame to disk. Change all related page tables to indicate that this page is no longer in memory.
* Move the required page and store it in the frame. Adjust all related page and frame tables to indicate the change.
* Restart the process that was waiting for this page.

### FIFO Page Replacement

* A very simple way of Page replacement is FIFO (First in First Out)
* As new pages are requested and are swapped in, they are added to tail of a queue and the page which is at the head becomes the victim.
* Its not an effective way of page replacement but can be used for small systems.

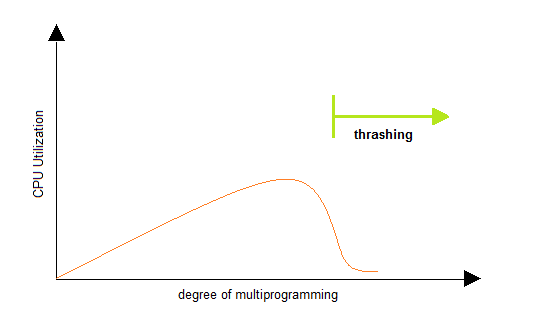
LRU Page Replacement

Below is a video, which will explain LRU Page replacement algorithm in details with an example.

### Thrashing

A process that is spending more time paging than executing is said to be thrashing. In other words it means, that the process doesn't have enough frames to hold all the pages for its execution, so it is swapping pages in and out very frequently to keep executing. Sometimes, the pages which will be required in the near future have to be swapped out.

Initially when the CPU utilization is low, the process scheduling mechanism, to increase the level of multiprogramming loads multiple processes into the memory at the same time, allocating a limited amount of frames to each process. As the memory fills up, process starts to spend a lot of time for the required pages to be swapped in, again leading to low CPU utilization because most of the processes are waiting for pages. Hence the scheduler loads more processes to increase CPU utilization, as this continues at a point of time the complete system comes to a stop.



To prevent thrashing we must provide processes with as many frames as they really need "right now".

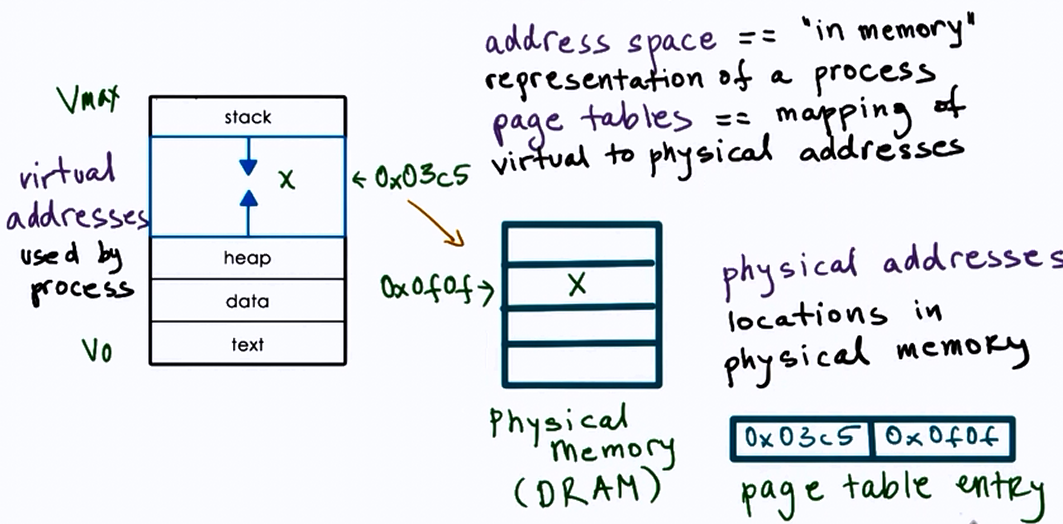
# Process and Process Management

**Process**: Instance of an executing program.

* State of execution
  + program counter, stack pointer
* Parts and temporary holding area
  + data, register state, occupies state in memory
* May require special hardware
  + I/O devices

Process is a state of a program when executing and loaded in memory (active state) as opposed to application (static state).

## What does a process look like?



### Type of state

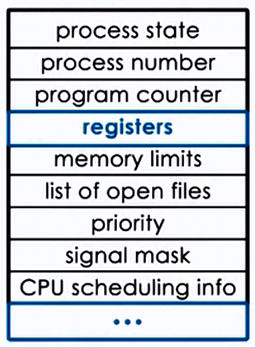
* Text and Data
  + static state when process loads first
* Heap
  + dynamically created during execution
* Stack
  + grows and shrinks
  + LIFO queue (used to store task checkpoints to resume the original process after switching from another.)

## How does the OS know what a process is doing?

Using:

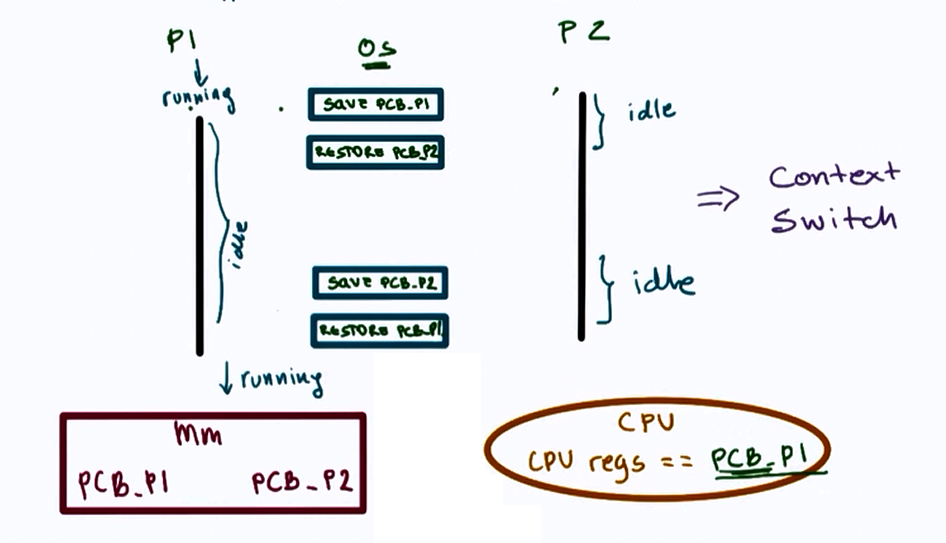
* Program counter
* CPU registers
* Stack pointer

## Process Control Block (PCB)



* PCB created when process is created
* Certain fields are updated when process state change e.g. memory mapping
* or other fields that change very frequently e.g. Program Counter

## How is PCB used ?



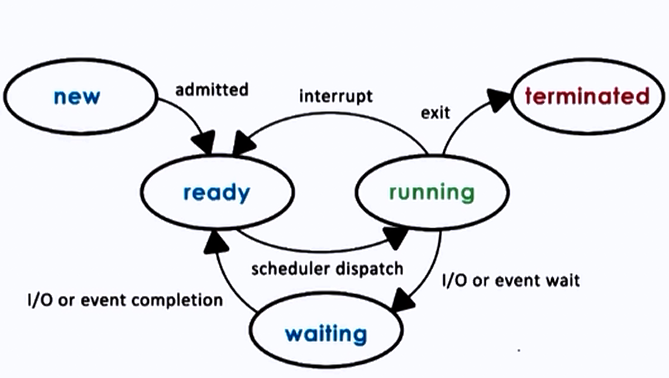
## Context Switch

* Mechanism used to switch from the context of one process to another in the CPU.
* They are expensive!
  + direct costs: no of cycles for load and store instructions.
  + indirect costs: **COLD** cache (read more [here](http://stackoverflow.com/questions/22756092/what-does-it-mean-by-cold-cache-and-warm-cache-concept))
    - Therefore limit frequency how context switching is done.

When a cache is **HOT**, most process data is in the cache so the process performance will be at its best.

Sometimes there are situations where we have to Context Switch (higher priority process, timesharing, etc.)

## Process Lifecycle



CPU is able to execute a process when the process is in Running or Ready state.

## Process Creation

#### Mechanisms:

* fork :
  + copies the parent PCB into new child PCB
  + child contains execution at instruction after fork
* exec :
  + replace child image
  + load new program and start from first instruction

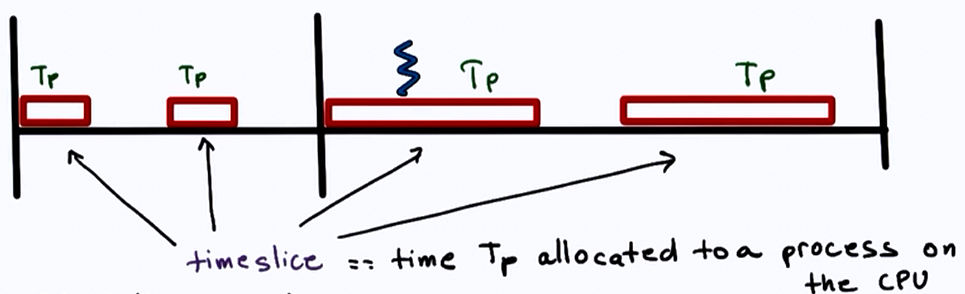
## What is the role of CPU scheduler?

CPU scheduler determines which one of the currently ready processes will be dispatched to the CPU to start running, and how long it should run for.

OS must :

* preempt => interrupt and save current context
* schedule => run scheduler to choose next process
* dispatch => dispatch process 2 switch into its context

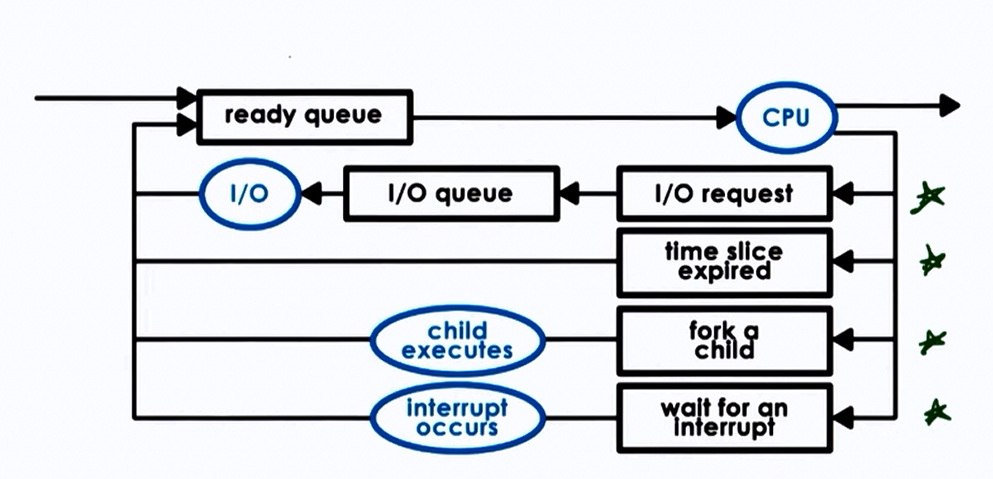
## Scheduling design decisions



* What are the appropriate timeslice values?
* Metrics to choose next process to run?

## I/O

A process can make way in the ready queue in a number of ways.



## Can process interact?

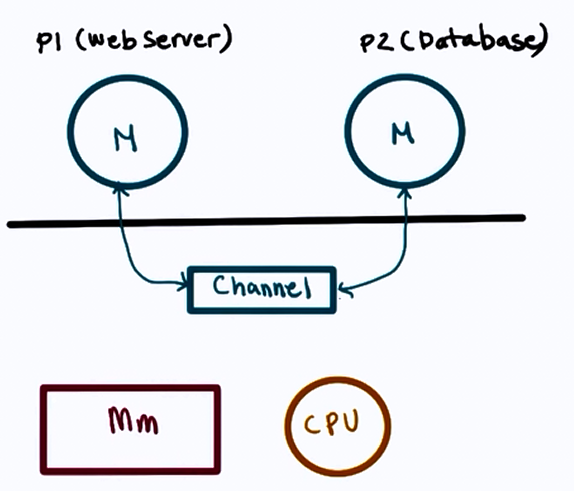
#### Inter Process communication:

IPC mechanisms:

* transfer data/info between address space
* maintain protection and isolation
* provide flexibility and performance

Two types of IPC models:

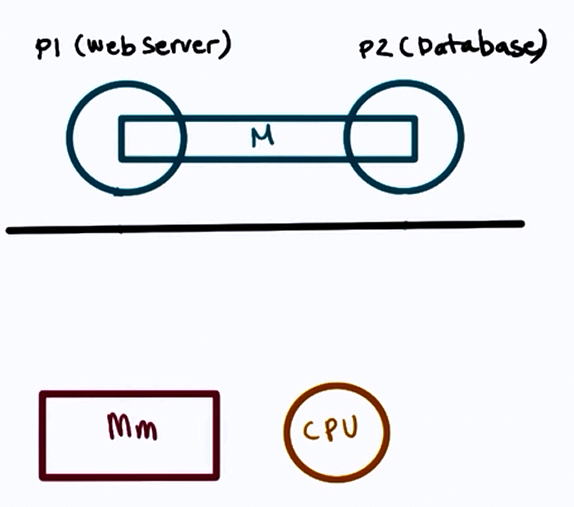
#### 1. ****Message Passing IPC****



* OS provides communication channel line shared buffer
* Processes can write(send), read(receive) msg to/from channel

**Advantages**: OS manages the channel  
**Disadvantages**: Overheads

#### 2. ****Shared Memory IPC****



* OS establishes a shared channel and maps it into each processes' address space
* Processes directly write(send), read(receive) msg to/from this memory

**Advantages**: OS is out of the way after establishing the shared channel  
**Disadvantages**: Re-implementing a lot of code that could have been done by the OS

Overall, **shared memory** based communication is better if mapping memory between two processes is amortized over a large number of messages.

II copy **Process Management**

**Process**: Process is the job which is the under Execution. Process is also known as the Running Job. For Execution there must be a System call which call the Processor or CPU for Performing any Operation. Process includes reading data from a File, Writing a Data from a File, Printing a document means to Say any Type of Operation is known as the Process. Every Process has some Attributes Like

1) **Process ID or Identification Number**.- Process id is given by the CPU when we Request for an Operation. Process id is also known as a unique identification Number which is Available when we request for a service. As we know there are many types of Operation those are performed on the Computer. So that for identification means which Process will be executed.

2)   **Process Name**. : Name Specify the Description of the Process. Name of Operation which is performed by the Process. For      Example Move the Mouse, Click on My Computer, Play the Song etc.

3)   **Process State (Ready, Active, Wait or Suspend)**. The Process has Some States, State Specify the Process State means      whether a Process is running or not, Whether a Process wait for CPU etc.

There are three Types of States

1)   **Ready**: Process wills Ready State, when we completed all the Input and Outputs. After giving the input, we wait for the      Execution. After the Completion of user Interaction means after all the Inputs and Outputs.

2)   **Active**: Active Means Process is running under the CPU.

3)   **Wait**: When a Process is waiting for the Input and Outputs from the user then this is called as on Wait State.

Process State will give you the Information about the Status of the Process.

4)   **Process Resources**: For Running a Process, then there are many Resources used. For entering some data then we must use  the Keyboard and for Sending data to the Computer CPU has used. So that For Performing any Operation, what types of  Resources. All the Devices those are attached with the Computer are known as Resources of the System.

5)   **Scheduling Information**. : Scheduling is used when there is Many Process those are running at a Time. Which Process will      be executed by the CPU? So that we use the Scheduling, determines the Time of CPU, Means CPU Time Divided into the various Processes.