

16. System Software

16.1 Purposes of an Operating System

I. Resource Management

- One OS task is to **maximize the utilization of four computer resources**:
 1. The CPU.
 2. Memory.
 3. The input/output (I/O) system.
 4. Disk.

CPU Optimization

- Involves **scheduling** for better utilization of CPU time.

I/O Optimization

- Involves the **direct memory access (DMA) controller** to allow I/O hardware to access the main memory **independently of the CPU**.
 - The CPU is fully utilized during I/O operations; DMA frees up the CPU to perform other tasks while the slower I/O operations are taking place.
- The DMA works as below:
 1. The DMA initiates the data transfers.
 2. The CPU performs other tasks while this data transfer occurs.
 3. Once the data transfer completes, an interrupt signal is sent to the CPU from the DMA.

Memory Optimization

- Moving frequently access instructions to **cache** for **faster recall**.
- Using **virtual memory** to swap memory to and from a disk.
- **Partitioning memory** to allow **multiprogramming**.
- Removing unused items from RAM by making a partition as available as soon as the process using it has terminated.

Disk Optimization

- **Disk caching** to hold data frequently transferred to/from the disk.
- **Compression utility** to decrease the size of a file stored on disk.
- **Defragmentation utility** to rearrange files to **contiguous disk space**.

II. User Interface Provision

- One OS task is to **hide the complexities of the hardware from the users** by:
 1. Using GUI rather than CLI.
 2. Using **device drivers** (simplifies the complexity of hardware interfaces).
 3. Simplifying the saving and retrieving of data from memory and storage.

4. Carrying out **background** utilities, such as virus scanning, which the users can "leave to its own devices."

III. Process Scheduling

- One OS task is to schedule processes to increase efficiency.
- The process of allocating the CPU's time on different processes based on their respective **priority**.

Terminologies

- **Process:** An instance of a program being executed by one or more threads.
- **Multitasking:** Allows computers to carry out more than one **process** at a time.
 - There are two types of multitasking:
 1. **Preemptive:** Processes are **forced to stop** after a certain amount of time.
 - Ensures no single task can monopolize the CPU.
 2. **Non-Preemptive:** Processes stop after it **voluntarily gives up control**.
- **Kernel:** The central component of an OS that runs in **kernel mode** (privileged mode) and directly **interacts with the hardware**.
 - It provides memory management, process management, device management, and system calls.
- **Low-Level Scheduling:** The process of selecting which process in the ready queue will be executed next by the CPU.
- **High-Level Scheduling:** The process of controlling the admission of new processes into the system.
 - It determines which jobs are loaded into memory for execution.
- **Starvation:** The situation when low-priority processes are indefinitely blocked from executing due to high-priority processes.

Why process scheduling is needed

- To allow **multi-tasking**.
- To allow **highest priority jobs** to be executed first.
- To keep the CPU busy all the time,
 - ...to ensure that all processes execute efficiently,
 - ...and to reduce waiting times for all processes.

Process States

1. **Ready:** The process is not currently being executed; It is in the queue waiting for the processor's attention.
2. **Running:** The process is being executed and has the processor's attention.
3. **Blocked:** The process is **waiting for an event** before it can continue running.
 - e.g., waiting for a user input.
4. **Terminated:** The process has finished execution and has been removed from the queue.

First Come First Served Scheduling (FCFS)

- The process added to a queue first is the process that leaves the queue first.
- **Benefit: starvation** won't occur → processes not ranked by priority.

Shortest Job First Scheduling (SJF)

- The process requiring the least CPU time is executed first.
 - It is **non-preemptive**, each process stops after its burst time.
- **Benefit: increased throughput** → more processes run in a unit time.

Shortest Remaining Time First (SRTF) Scheduling

- The process with the least **remaining burst time** is executed first.
 - It is **preemptive**, each process stops when a faster process arrives.

Round Robin Scheduling

- A **fixed time slice** is given to each process; this is known as a **quantum**.
 - When a process arrives, it is pushed into the READY QUEUE; once a process exceeds its quantum, it is pushed into the BLOCKED QUEUE, then another process from the READY QUEUE is popped out.
 - **Context switching** is used to save the state of the pre-empted responses.
- **Benefit: starvation** won't occur → each process has the chance to run.

Interrupt Handling & OS Kernels

1. Interrupt received; other interrupts are disabled.
 - So the process that deals with the interrupt cannot itself be interrupted.
2. The state of the current process is saved on the **kernel stack**.
3. The source of the interrupt is identified; its priority is checked.
4. Using the **interrupt dispatch table**, the system jumps to the **interrupt service routine (ISR)**.
5. Once completed, the state of the interrupted process is restored.
6. Other interrupts are restored so that further interrupts can be dealt with.

IV. Memory Management

- One OS task is to determine **how memory is allocated** when many processes are competing with each other.

Single (Contiguous) Allocation

- All of the memory is made available to **a single application**

Paged Memory (Paging)

- Memory is split up into partitions of a **fixed size**, both physically and logically.
 - **"Frames"**: Physical memory blocks.
 - **"Pages"**: Logical memory blocks.
- A **page table** is used; it maps logical page address (index) into physical address.

- It stores page number, flag status, frame address, and the time of entry.
- The OS divides the memory into pages.

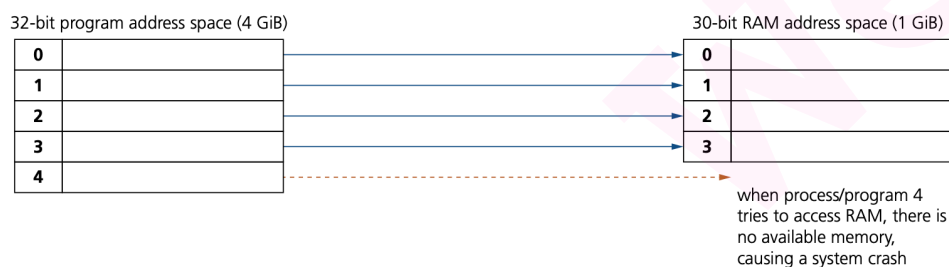
Segmented Memory (Segmentation)

- Logical memory is split up into partitions of **variable sizes** called "**segments**."
- A **segment map table** is used; It includes segment numbers and their user-defined offsets.
 - Physical address is derived from the segment numbers and offsets:

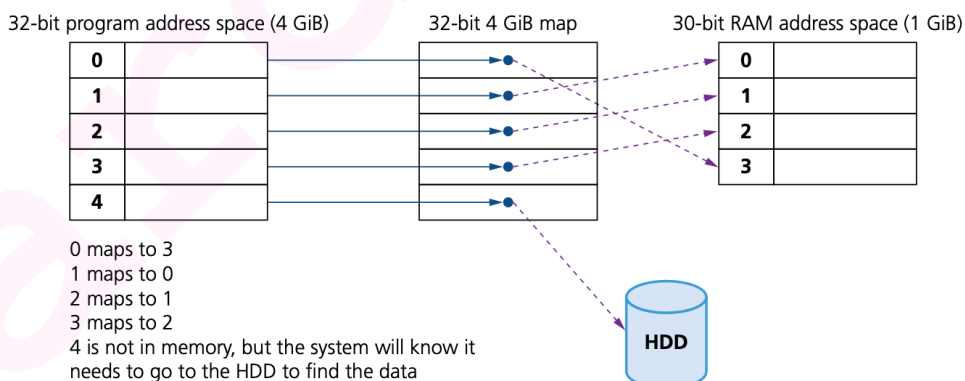
$$\text{Address of segment in physical memory space} = \text{segment number} + \text{offset value}$$
- Access time for segmentation is **slower** than paging.

Virtual Memory

- To solve the problem when RAM doesn't have enough space for a program:



- Disk/Secondary storage is used to extend the RAM,
 - ...so the CPU appears to be able to access more memory space than the available RAM.
- Only the data in use needs to be in main memory, so data can be swapped between RAM and virtual memory as necessary.
- Virtual memory is created **temporarily**, as demonstrated below:



- A drawback of virtual memory is **disk thrashing**:
 - Disk thrashing is a problem that occurs when there are **frequent transfers** between the main memory and the secondary memory.
 - As main memory fills up, more pages need to be swapped in and out of secondary memory.
 - This swapping leads to a very high rate of **hard disk head** movements.
 - Eventually, more time is spent swapping the pages than processing the data.

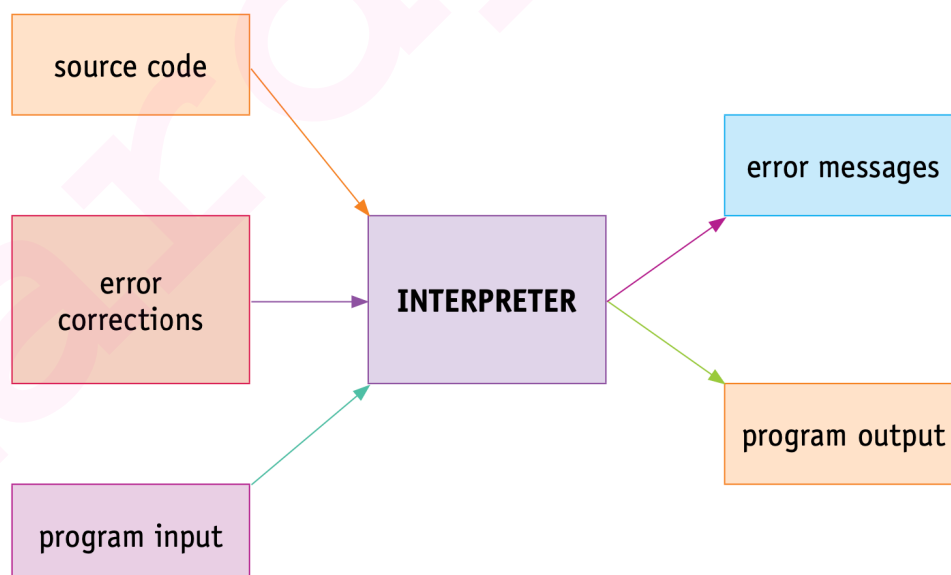
Page Replacement

- Techniques to select which page from RAM should be replaced by a page from the secondary memory when a *page fault* occurs.
 - Page fault:** When a new page is requested but is not in the main memory.
- There are four page replacement algorithms:
 - First in first out (FIFO):**
 - The oldest page at the front of the queue is replaced.
 - Drawback:* More page faults occur with more page frames.
 - Optimal page replacement (OPR):**
 - Anticipates the page that is referenced farthest in the future.
 - Drawback:* Impossible to implement :); used as a theoretical comparison.
 - Least recently used page replacement (LRU):**
 - The page that has not been used for the longest time is replaced.
 - Second chance page replacement:**
 - A "second-chance counter" is maintained for each frame, which increases by 1 when a page is referred to again when it's in the RAM.
 - A pointer cyclically points at each frame as a page fault occurs.
 - If the frame's second-chance counter is zero, it will be replaced.
 - Else, the counter decreases by 1 and the pointer increases by 1.
 - May refer to [Geeks4geeks's explanation](#).

16.2 Translation Software

Interpreters

- An interpreter must present in the system for the source code to be run.



- How interpreters translate a program:
 - An interpreter examines source code one statement at a time.
 - It checks each statement for errors,
 - ...if no error is found, the statement is executed,
 - ...if an error is found, this is reported and the interpreter halts.
 - Interpretation is repeated for every iteration in repeated sections of code.

- Interpretation has to be repeated **every time the program is run**.

Stages of Compilation

1. Lexical Analysis

- Removes unnecessary characters (e.g., comments and white spaces).
- **Tokenize** the program by using a **keyword table** and a **symbol table**.
 - **Keyword table** stores *reserved words used, operators used, and their matching tokens*.
 - **Symbol table** stores *identifier name used, their data type and role (e.g., variable, constant, array, procedure), and their location*.

2. Syntax Analysis

- Checks that the rules of **grammar/syntax** have been obeyed.
 - The Backus-Naur Form is being used to parse the syntax.
- Produces an **error report**.

3. Code Generation

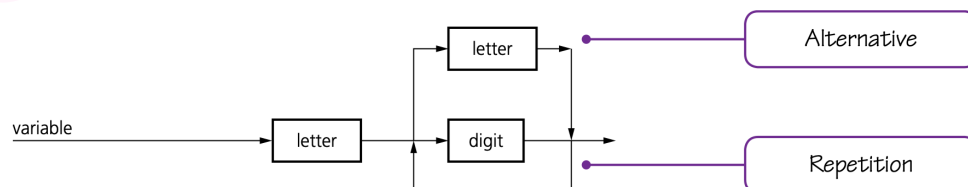
- Produces the **object code** (either machine code or intermediate code).
 - **Intermediate code**: A **machine-independent** code that is converted to machine code when the program is loaded.

4. Optimization

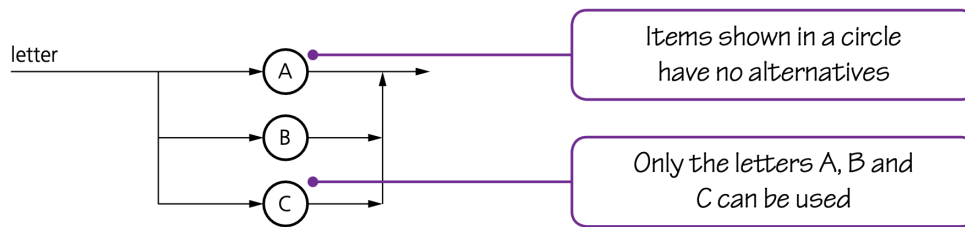
- Removes **redundant codes**.
- Reorganizes code to make it more efficient.
- Reduces time, memory, storage, and CPU use of a program.

Syntax Diagrams

- A syntax diagram visually represents the valid structure of an expression.
- The diagram below specifies the structure for a "variable," which shall be:
 - A letter at the start,
 - Any numbers of letters and digits followed.



- Note that any expression, including the "letter" and "digit," shall be specified, as shown in the following diagram.



Back-Naur Form (BNF)

Back-Naur Form syntax

- "< >" → used to enclose an item (e.g., "digit").
- "::=" → separates an item from its definition.
- "|" → indicates a choice between items.

- A simple variable definition can be represented below.

```
<variable> ::= <letter> | <digit>;
<letter>  ::= A | B | C
<digit>   ::= 1 | 2 | 3
```

- To indicate repetition, recursive definitions shall be used.

```
<variable> ::= <letter> <rest>
<rest>    ::= <letter> <rest> | <digit> <rest> | <digit> | <letter>
```

Reverse Polish Notation (RPN)

- A method of representing an arithmetical or logical expression without the **use of brackets** or special punctuation.
 - It uses a **prefix notation**: a variable is placed after the variables it acts on.

$$A - B * C \Rightarrow A B C * - \text{ (RPN)}$$

- How a RPN equation can be evaluated:
 - The expression is read from left to right, one item at a time.
 - Each element is checked to see if it is an operator or a value.
 - Values are pushed onto a stack until an operator is found.
 - The operator is applied to the last two values on the stack, **and** the result is pushed back onto the stack.
 - This repeats until a single value remains, which is the solution.