UTM
UNIVERSITI TEKNOLOGI MALAYSIA

CHAPTER 4

Security in Operating System

Course Learning Outcome

After the completion of this module, students should be able to:

- Explain protection features provided by general purpose operating systems that is protecting memory, files and he execution environment
- Differentiate control access methods provided by the operating system
- Understand how user authentication works

OS Security

- **What is OS?**
- A program that acts as an **intermediary between user and computer hardware**
- **Purpose of OS:** provide an **environment in which a user can execute program**s in a convenient and efficient manner

OS Security : Overview

- **What is protection in OS?**
  - Prevention of **mischievous, intentional violation** of an access restriction by a user
  - Prevention of **erroneous program** from affecting the execution of other programs
  - Improve **reliability**

Security Breaches – Key Terms

- **Exposure** – Potential loss or harm due to a vulnerability.
  Example: Storing passwords in plain text exposes them to theft.
- **Vulnerability** – A flaw that can be exploited.
  Example: A buffer overflow in the login process.
- **Threat** – Circumstance or event that could cause harm.
  Example: Insider with admin privileges planning sabotage

Security Breaches

- Security breaches occur when the **confidentiality, integrity, or availability** of system resources is compromised.
- **Types of breaches:**
  - **Interruption** – Resource becomes unavailable (e.g., DoS attack).
  - **Interception** – Unauthorized access to data (e.g., packet sniffing).
  - **Modification** – Unauthorized alteration of data (e.g., file tampering).
  - **Fabrication** – Insertion of false data (e.g., fake log entries).
- **Example:**
  A malware-infected application modifies system DLLs to insert backdoors.

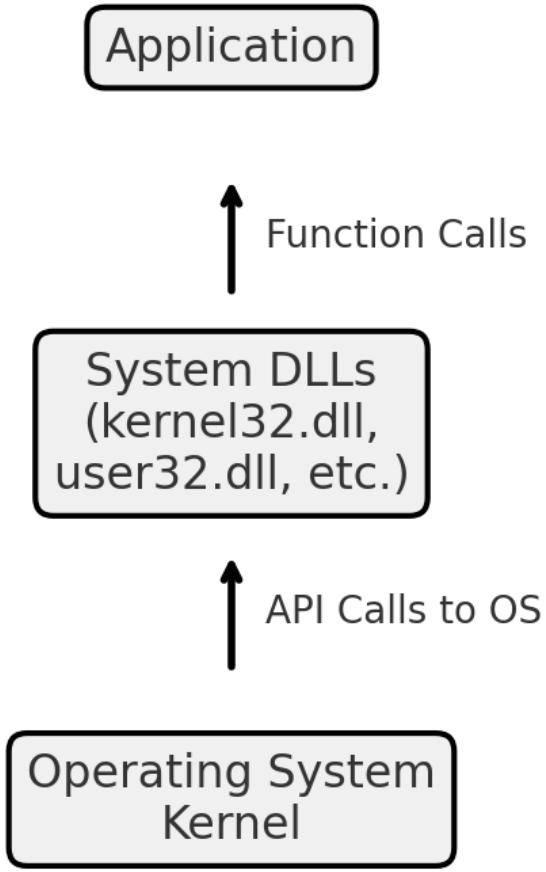
A **system DLL** (Dynamic Link Library) is a **shared library file** used by the operating system and applications to provide **common functions**.

- **Extension:** .dll in Windows.
- **Purpose:** Stores compiled code, resources, and routines that programs can call instead of having their own copy.
- **Location:** Typically found in C:\Windows\System32\ or C:\Windows\SysWOW64\ on Windows.
- **Examples:**
  - kernel32.dll – Handles memory management, I/O operations.
  - user32.dll – Manages user interface elements (windows, buttons).
  - advapi32.dll – Provides advanced Windows API services like registry and security.

Why they matter for OS Security?

- **Core functionality**: Many system processes depend on them.
- **Attack target**: If malware replaces or injects code into a system DLL, it can gain elevated privileges or run malicious code every time the DLL is loaded.
- **Protection mechanisms**:
  - Windows File Protection (WFP) and Windows Resource Protection (WRP) prevent unauthorized changes.
  - Digital signatures verify DLL integrity.

How Applications Use System DLLs in Windows



Why Focus on OS Security in General-Purpose Systems

- General-purpose OSs (Windows, Linux, macOS) are designed for diverse workloads and users, making them:
  - **Highly flexible** but with larger attack surfaces.
  - **Multi-user capable**, requiring strict access controls.
  - **Widely deployed**, making them popular targets for attackers.
- **For Example:**
  A misconfigured Linux server running Apache can allow remote attackers to exploit kernel vulnerabilities via malicious requests.

Recent Evolution in OS Security (2023–2025)

Modern OS security integrates both **software and hardware-based protections**

Year/Trend	Development	Example OS
2023	Hardware-enforced stack protection	Windows 11 using Intel CET / AMD Shadow Stack
2024	Memory-safe programming adoption	Windows kernel components rewritten in Rust
2024	Enhanced Secure Boot	Ubuntu 24.04 with improved shim and kernel signing
2025	AI-assisted real-time malware detection	macOS XProtect Remediator with ML-based heuristics

Security Goals

OS security aims to achieve the **CIA Triad**:

- **Confidentiality:** Ensure only authorized access to information.
  Example: File permissions to restrict sensitive data.
- **Integrity:** Prevent unauthorized modification of data.
  Example: Digital signatures on system updates.
- **Availability:** Keep systems operational for authorized users.
  Example: CPU scheduling fairness to avoid process starvation.

Example – Security Breach Scenarios

Goal Compromised	Scenario
Confidentiality	A malicious user reads the /etc/shadow password file in Linux.
Integrity	Malware alters system DLL files in Windows to insert a backdoor.
Availability	A process runs an infinite loop consuming all CPU cycles, making the system unresponsive.

Protected Objects of the General-Purpose Operating System

Objects that require protection:

- **Memory segments** (RAM allocated to programs)
- **I/O devices** (printers, network interfaces, disks)
- **Programs and processes** (running applications)
- **Files and directories**
- **System tables and instructions**
- **Authentication credentials** (passwords, keys)
- **The protection mechanism itself** (access control system)

Protection of Objects

Methods the OS uses to protect resources

- **Separation - keeping one user's objects separate from other users'**

  Physical – Different hardware for different security domains.
  Temporal – Processes run at separate times.
  Logical – Software-enforced isolation.
  Cryptographic – Data encryption to prevent unauthorized reading.

- **Access Control Mechanisms**:

  Directory-based
  Access Control Lists (ACLs)
  Access Control Matrices (ACMs)
  Capabilities

- **Granularity of Control**:

  Coarse-grained: Whole files or devices.
  Fine-grained: Individual records or bytes.

Principles of Separation

Type	Description	Example
Physical	Different processes use physically separate resources.	Dedicated printers for different security clearance levels.
Temporal	Processes with different security requirements run at different times.	Batch processing of classified and unclassified jobs in different time slots.
Logical	Programs are restricted to their permitted domain.	Virtual memory ensuring one process cannot access another process's memory.
Cryptographic	Data is encoded so that unauthorized processes cannot understand it.	Encrypting sensitive files before storing them in shared folders.

Granularity of Control

- **Granularity** refers to the **size of the object** on which access control is enforced.

  Fine-grained control – Access rights defined at small units (e.g., single variables in memory).

  Coarse-grained control – Access rights defined at large units (e.g., entire files).
- **Trade-off:**

  Finer granularity = **better security, more overhead**.

  Coarser granularity = **easier to implement, less overhead, but less flexible**.
- **Example:**

  File system level: Control at the **file** level (coarse).

  Database level: Control at the **record** or **column** level (fine).

Protection of Resources

- **OS Protection includes these:**
  - **Memory protection**
  - **File protection**
  - **General control of access to objects**
  - **User authentication**

Basis of Protection : Separation

- **Physical separation**

  Different processes use different physical objects
  Separate printers for different levels of security
- **Temporal separation**

  Processes with different security req. are executed at different times
- **Logical separation**

  Programs cannot access outside its permitted domain
  Illusion of single process operation
- **Cryptographic separation**

  Process conceal data and computations that its unintelligible to outside process

Is separation the only answer?

- Separation of **different forms may** be combined
- Separation can lead to **poor resource sharing**
- We want **users and objects** to be able to share algorithms and functions without compromising their security needs
- A practical OS provides protection by separation and ‘controlled’ sharing

OS can help via

- Do not protect

  OS with no protection
- Isolate

  Different processes running concurrently are unaware of the presence of each other
- Share all or share nothing

  Public (for all) or private (for owner only)
- Share via access limitation

  Ensuring only authorized access occurs
- Share by capabilities

  Dynamic creation of sharing rights – can depend on owner/ context of computation/object
- Limit use of an object

  To what extend is your privileges? E.g. Can view but not print

Memory and address protection

- Protection can be built into hardware that controls use of memory

- Include
  - Fence
  - Relocation
  - Base/bounds registers
  - Tagged architecture
  - Segmentation
  - Paging
  - Combined paging with segmentation

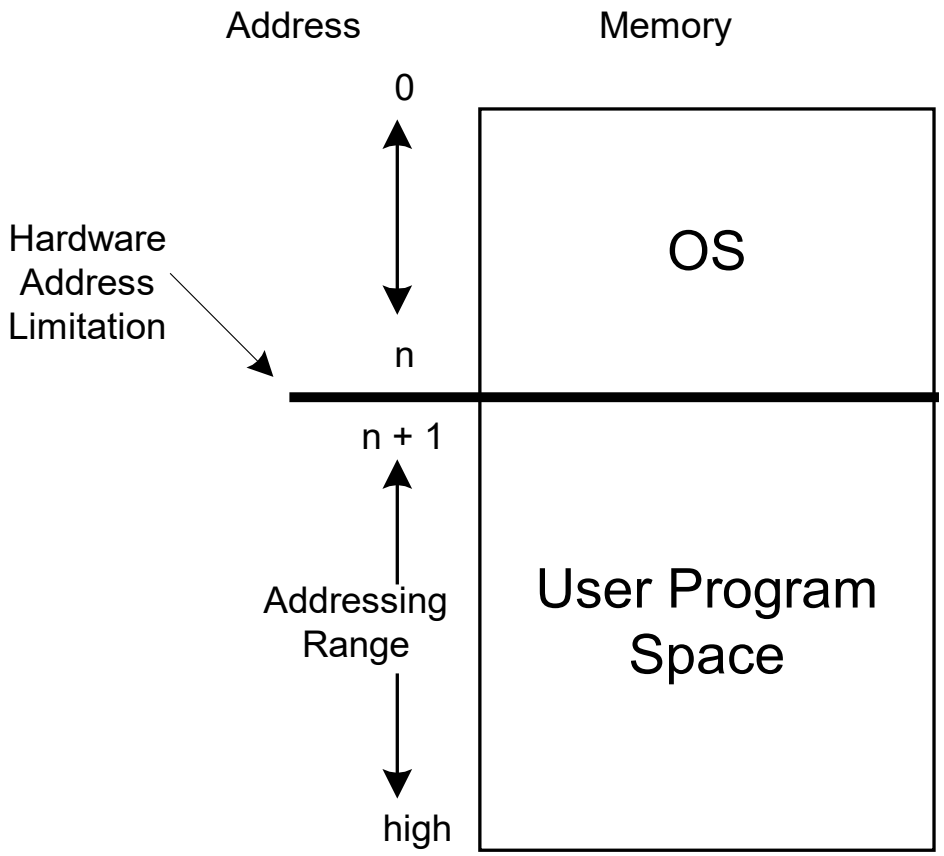
Memory and address protection prevents processes from:

- Overwriting the OS code or data.
- Accessing other processes' memory.
- Executing instructions they are not authorized to run.

In a multi-user or multi-tasking OS, **memory isolation** ensures stability, security, and integrity.

Memory Protection: Fence

- In a farm, a fence keeps the cows from grazing the neighbour's yard/land.
- In OS, same function. Keeps the users from ‘grazing’/destroying the OS resident portion of memory.
- A fence is a method to confine users to one side of a boundary.
- Usually, fence is implemented via a hardware register

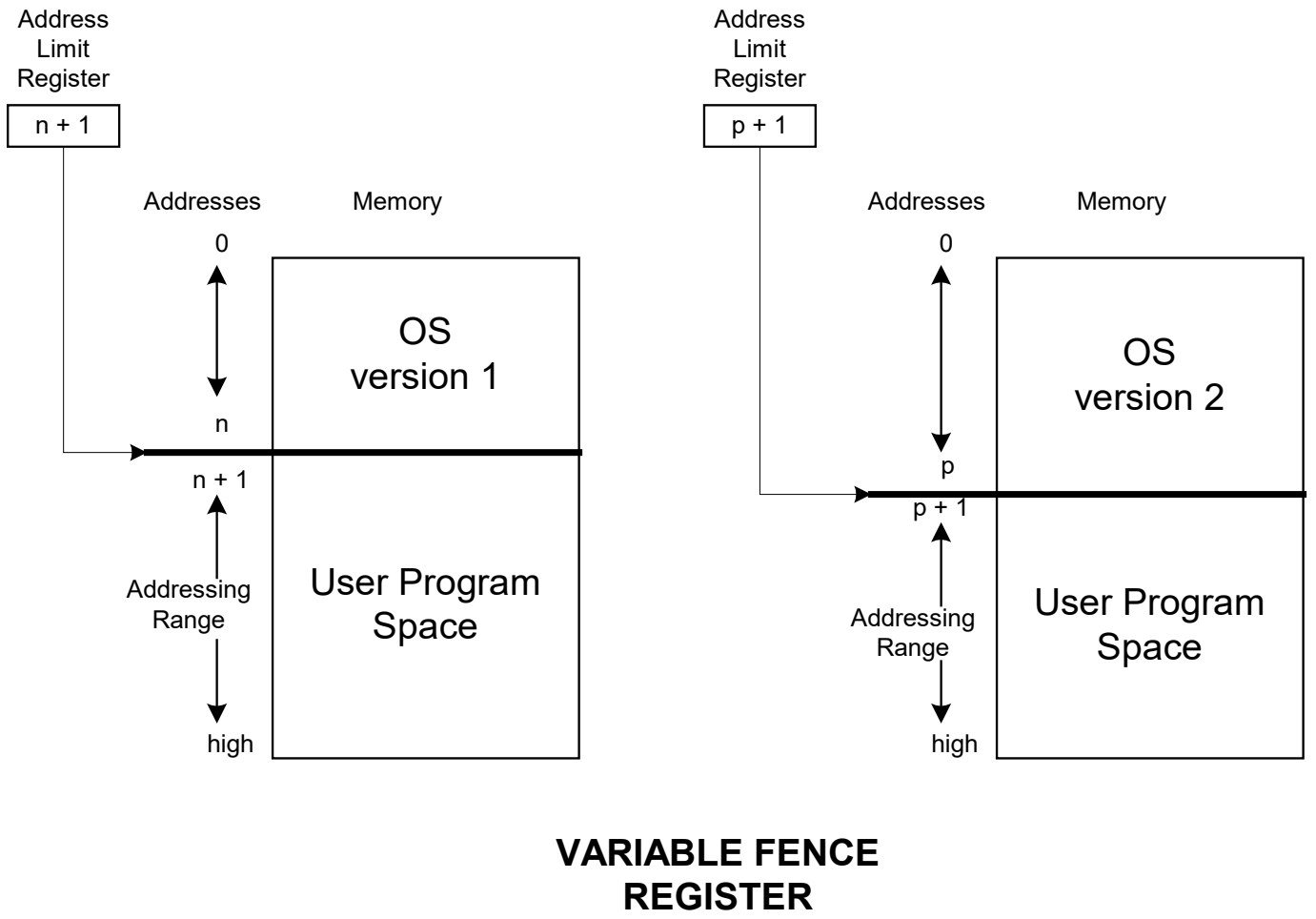


- Fixed fence is very **restrictive**
  - Amount of space predefined
  - Whether OS use it or not - its there
    - Waste of space may occur
  - OS have no space to grow

FIXED FENCE

Fence

- Fence register is another way
  - Fence is **not fixed** – can be changed
  - The register contains the end of OS
  - Address of user programs or modification will be compared to the register.
    - If address in user area – instruction executed
    - If address in OS area – error condition raised
  - Can protect OS from user but not user from another user
    - User cannot identify certain areas as off-limits



VARIABLE FENCE REGISTER

- The fence is **not fixed**—it changes depending on:
  - OS version.
  - Amount of OS code loaded.
  - Memory allocation needs.
- This flexibility helps avoid **wasting memory space**.

Memory Protection: Relocation

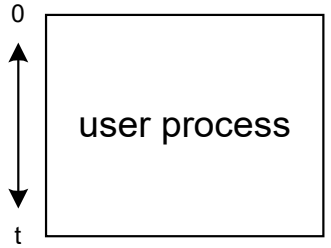
- OS do change (updates, dynamic loading, etc); hence programs must be written in a way **that does not depend on placement at a specific location** in memory.
- Relocation takes a program and puts it in **an appropriate place/address**, with the help of a **relocation factor**.

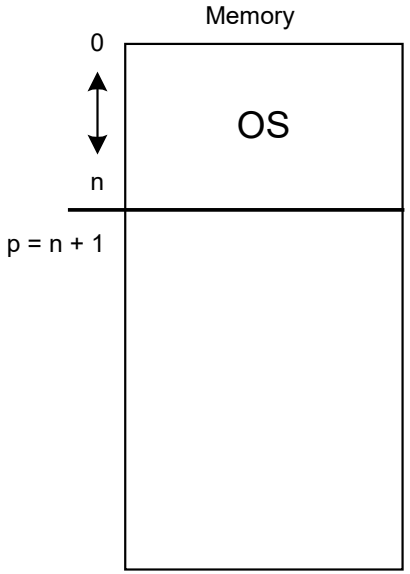
  The OS or hardware adds a relocation factor (offset) to all logical addresses generated by the program.

  This maps the program's logical address space to its actual physical memory location.
- A **fence register** can act as a hardware relocation device by shifting the starting address of user programs while preventing them from accessing OS space.

User Process (Logical View)

- The user program thinks it starts at address 0 and ends at address t.
- This is the **logical address space**.



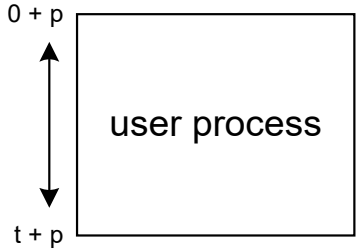


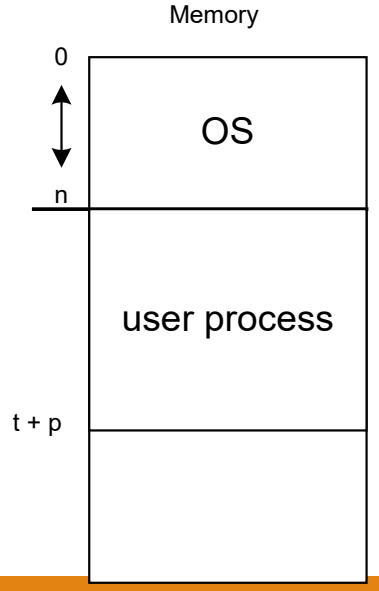
Physical Memory without Relocation

- OS is loaded at the bottom (address 0 to n).
- The **user program space** starts after OS, at physical address p = n + 1.
- Without relocation, the program's logical addresses would not match its physical location.

Relocated User Process (Logical View)

- Program addresses are now adjusted by adding **relocation factor p**.
- Logical address 0 becomes p in physical memory.
- Logical address t becomes t + p in physical memory.





Physical Memory with Relocation Applied

- OS is still in addresses 0 to n.
- The user process is relocated to start at physical address p.
- Every memory reference in the program is **offset by p**, ensuring correct access without rewriting the program.

Memory Protection: Base/Bounds Registers

- In a **multiuser, multiprogramming environment**, multiple processes from different users share main memory.
- To ensure **isolation** and **prevent interference**, the OS uses hardware registers:
- **Base Register**

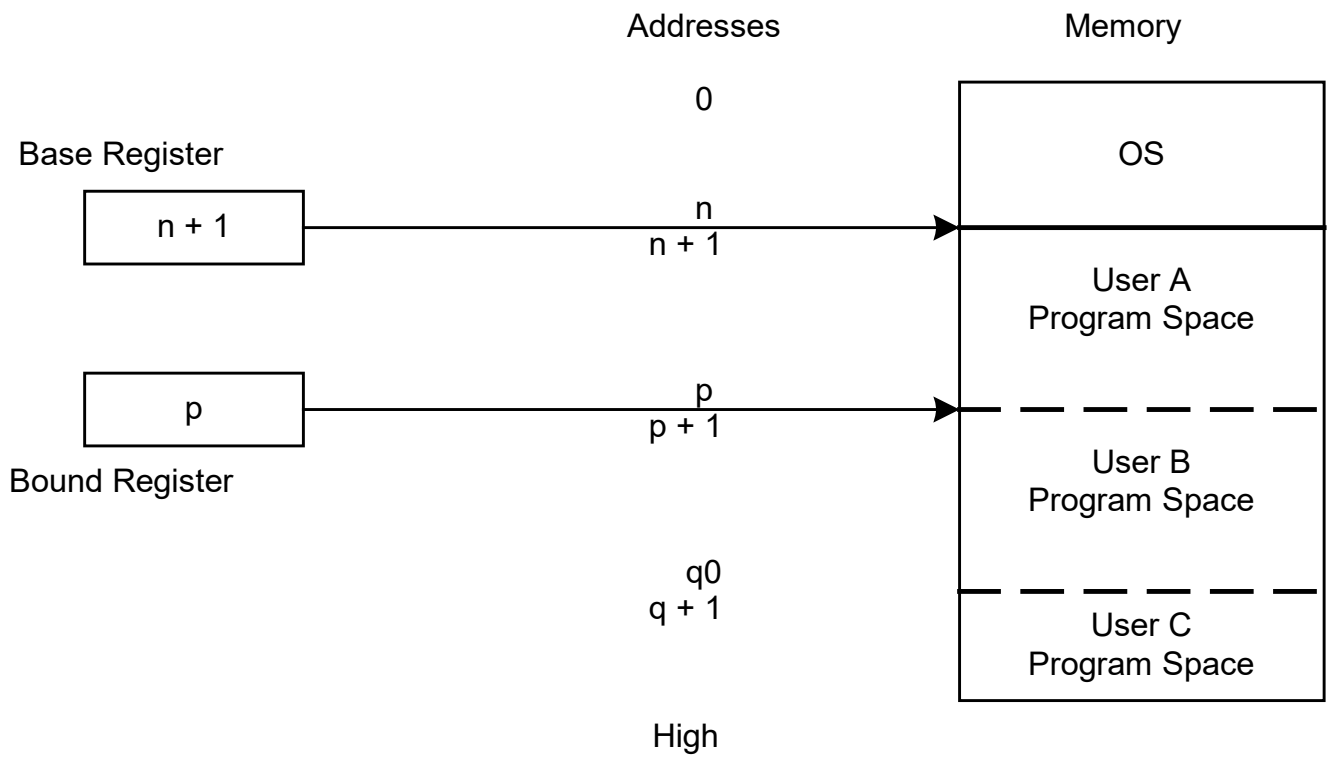
  Stores the **starting physical address** allocated to a process.
  Any logical address from the process is **added to the base** before accessing physical memory.
  Prevents the process from addressing memory **below** its allocated space.
- **Bounds Register**

  Stores the **size** or **upper limit** of the process's allocated memory space.
  Ensures the process cannot access memory **beyond** its assigned region.

Example
Let's say:
- Base = 1000
- Bound = 3000 (process has 2000 bytes allocated)
- Logical address = 250

Physical Address = Base + Logical = 1000 + 250 = **1250**.

If the logical address tries to go beyond 2000 (e.g., 2500), the hardware triggers a **trap** and prevents the access.



Benefits

- **Strong isolation** between processes.
- **Relocation** is easy — just change the base register for the process.
- **Dynamic protection** — values can be changed during context switching.

Limitations

- Only provides **one contiguous block** of memory for a process — no support for fragmented allocation.
- Not suitable for sharing non-contiguous segments between processes.

Modern Relevance

While pure base/bound is rare today, the concept survives in:

- **Virtual memory systems** (page tables implement similar boundary checks).
- **Sandboxing** environments (e.g., WASM memory bounds checks).
- **Hardware virtualization** (VMCS structures maintain base/bounds-like control fields).

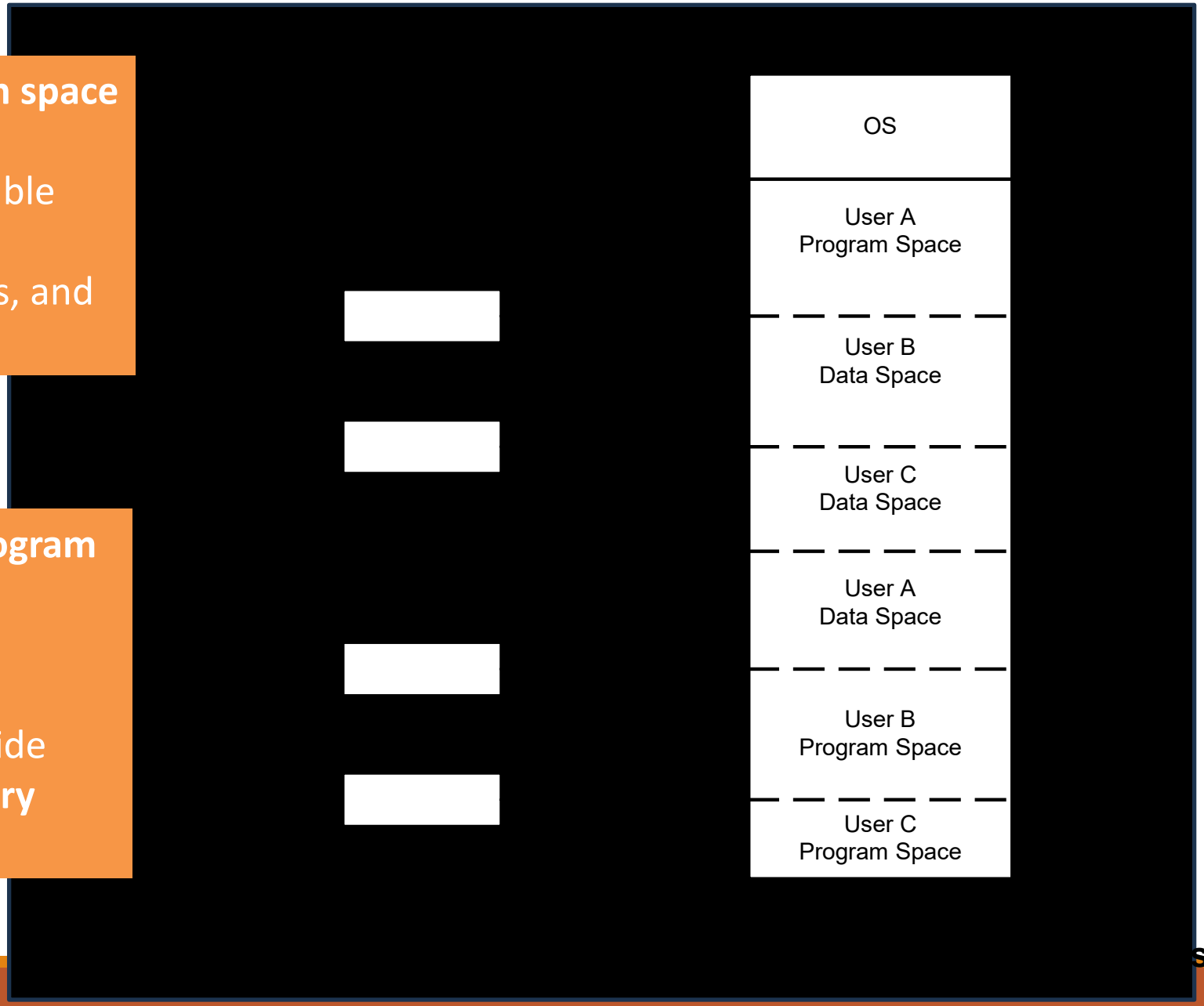
- Another pair of base/bound registers may be used for addresses inside a user's address space

  Program Base/Bound – for instructions only

  Data Base/Bound – for data access only
- Saves user from overwriting and destroying his program
- Also gives the ability to split program into 2 pieces and relocated separately
- In some systems, **two pairs** are used to provide **separate protection for code and data**, which improves security and flexibility.

- Each user (A, B, C) has **separate program space** and **data space** in memory.
- The program space contains the executable instructions.
- The data space contains variables, arrays, and other modifiable information.

- **Instruction Fetches**: Checked against **Program Base/Program Bound**.
- **Data Accesses**: Checked against **Data Base/Data Bound**.
- If a process tries to access memory outside either range, the hardware raises a **memory violation trap** and stops the process.



Memory Protection: Tagged Architecture

- **Problem with Base/Bound Registers:**

  They work at a **high granularity** — usually protecting entire contiguous memory regions.
  This means that once you give access to a process, it gets **all-or-none** access to that whole region.
  No way to allow access to some words in the block while protecting others.

Tagged Architecture Concept
Solution: Assign **access rights at the word level** instead of region level.

Tagged Architecture Concept

- Every memory word has **extra bits** called **tag bits**.
- These tag bits define **who can access it** and **what type of access is allowed**:

  R → Read only

  RW → Read and Write

  X → Execute only
- Tag bits are stored alongside the data word in memory.
- Access rights can only be changed by **privileged OS instructions**.

Tag	Word	Meaning
R	0001	Read-only data word
RW	0137	Read and write allowed
R	4091	Read-only word
X	(code)	Execute-only instruction

How It Works

1. When the CPU executes an instruction that accesses memory:
    1. It checks the **tag bits** for that memory word.
    2. If the access type matches the allowed rights, proceed.
    3. If not, raise a **protection fault**.
2. This check happens **every time** a word is accessed — high security but with some performance cost.

Advantages

- **Fine-grained control**: Can protect individual memory words instead of large blocks.
- **Minimum sharing**: Give access only to the exact data needed for task completion.
- **Data classification**: Different types of data can have different protections, even if stored side-by-side.

Disadvantages

- **Complexity**: Requires major changes to the hardware and OS.
- **Cost**: More storage for tag bits and additional CPU logic.
- **Performance**: Slightly slower due to extra checking on every access.

Memory Protection: Segmentation

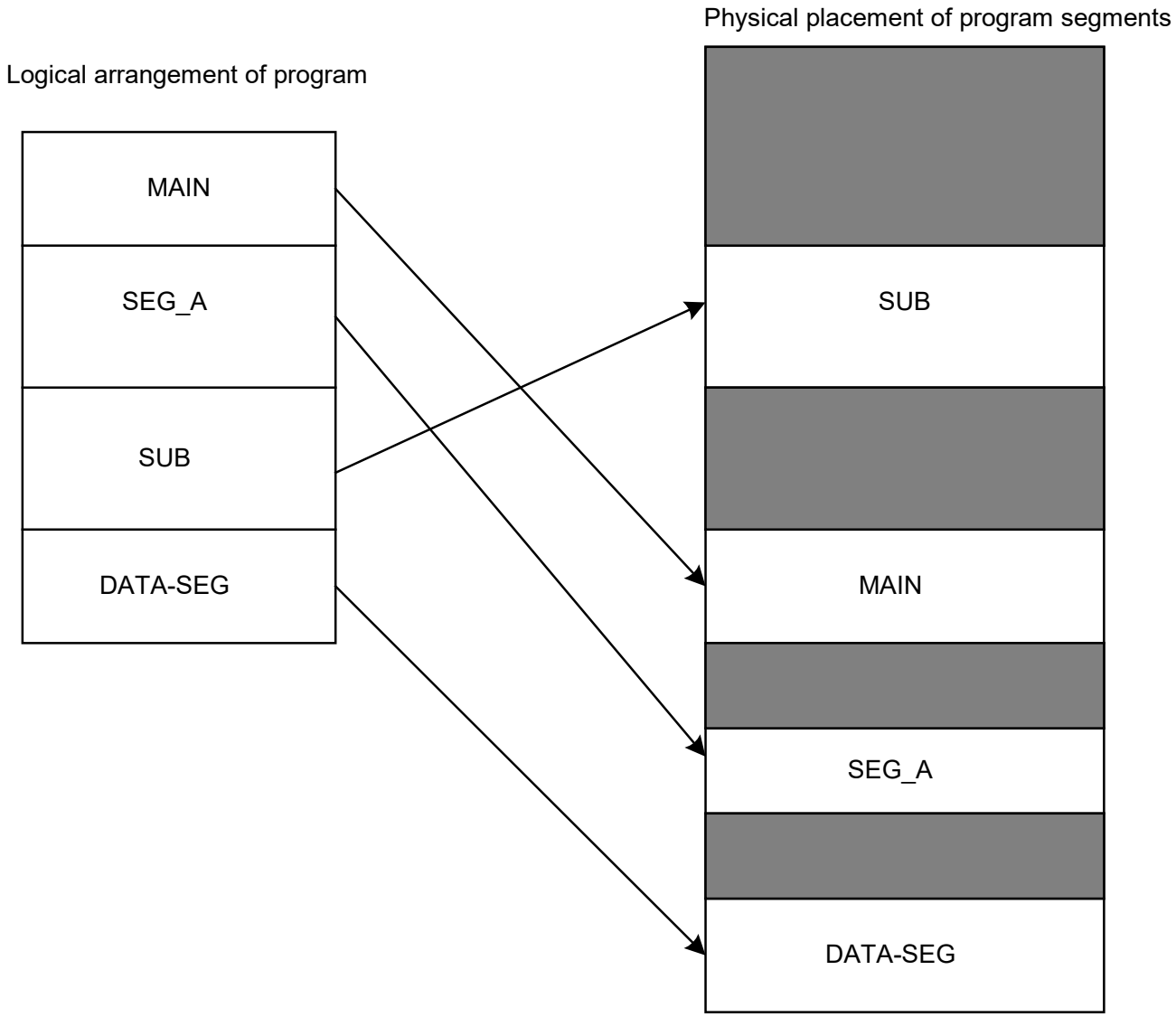
- Segmentation is dividing a program into separate pieces/segments having different access rights
- Each segment has a unique name
- Code within a segment addressed as <name, offset>

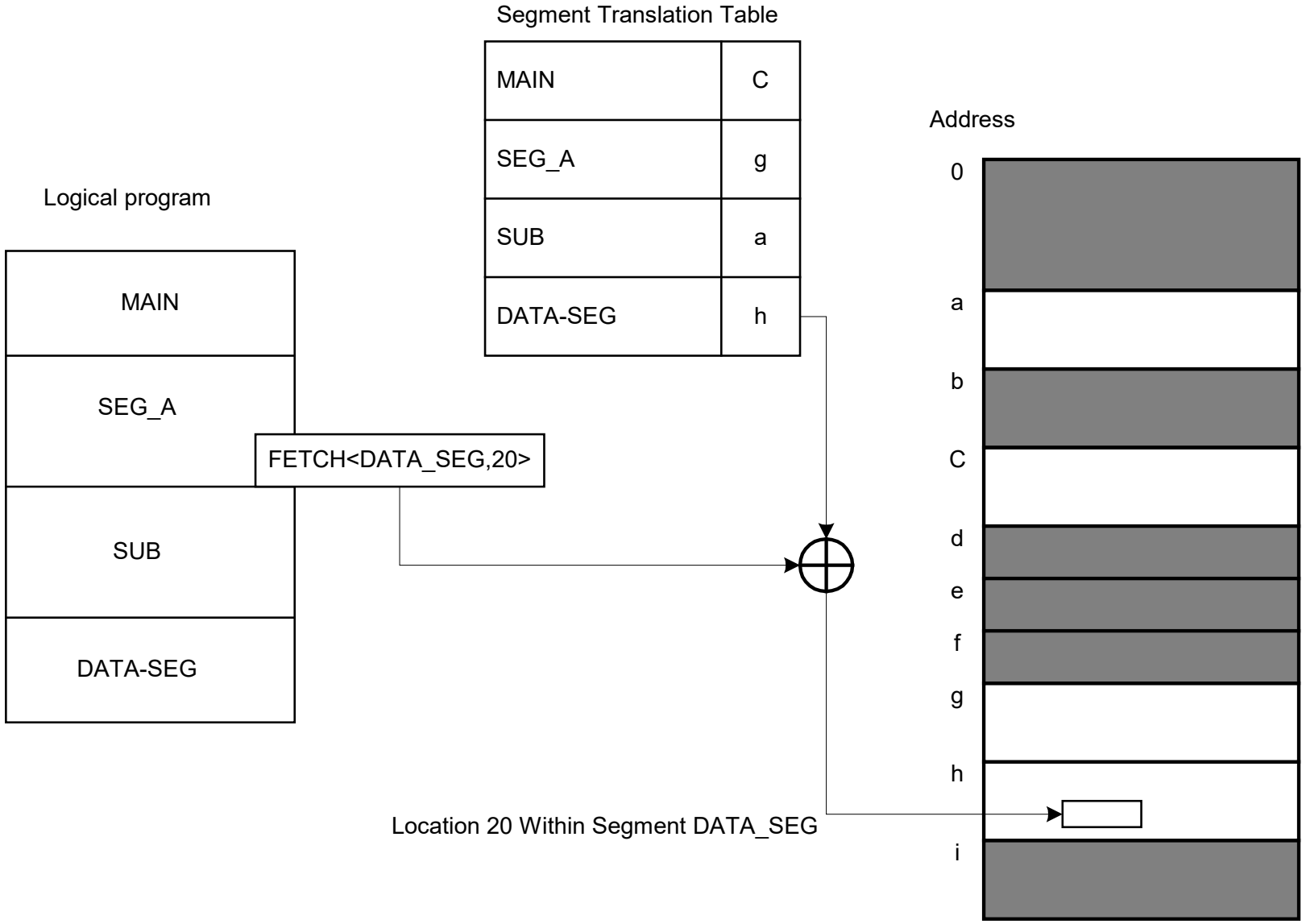
  Name – name of segment
  Offset – its location from the start of the segment

Segmentation

- Program is a long collection of segments and these segments may be placed in any available memory locations.
- OS must have a way arranging this

  Segment look-up tables
  Keeps segment name and true address in memory
  User need not know the address, just <name,offset>


Logical arrangement of program
MAIN
SEG_A
SUB
DATA-SEG
Physical placement of program segments
SUB
MAIN
SEG_A
DATA-SEG


Segment Translation Table
MAIN
C
SEG_A
g
SUB
a
DATA-SEG
h
Address
Logical program
MAIN
SEG_A
SUB
DATA-SEG
FETCH<DATA_SEG,20>
0
a
b
C
d
e
f
g
h
i
Location 20 Within Segment DATA_SEG

- **Benefit of segmentation**
  - Each address reference is checked for protection
  - Many different classes of data item can be assigned different levels of protection
  - 2 or more users can share access to a segment, with potentially different access rights
  - A user cannot generate an address or access to an unpermitted segment.

- Segmentation must offer protection while being efficient.

- **Problems with segmentation**
  - Protection problem – segment size
  - Efficiency problem – segment name
  - Memory utilization problem

- **Protection problem**

  A user can refer to a valid segment name with an offset beyond end of segment ➔ can read other parts of memory, hence unsafe
  <A,999> but segment A is 200 bytes long
  So must check generated address to verify that it's not beyond end of segment ➔ have segment length in look-up table
  Need more time and more resources

- Efficiency problem

  Segment name inconvenient to encode and will also slow look-up

  Compiler can translate name to numbers but problem when 2 procedures share the same segment

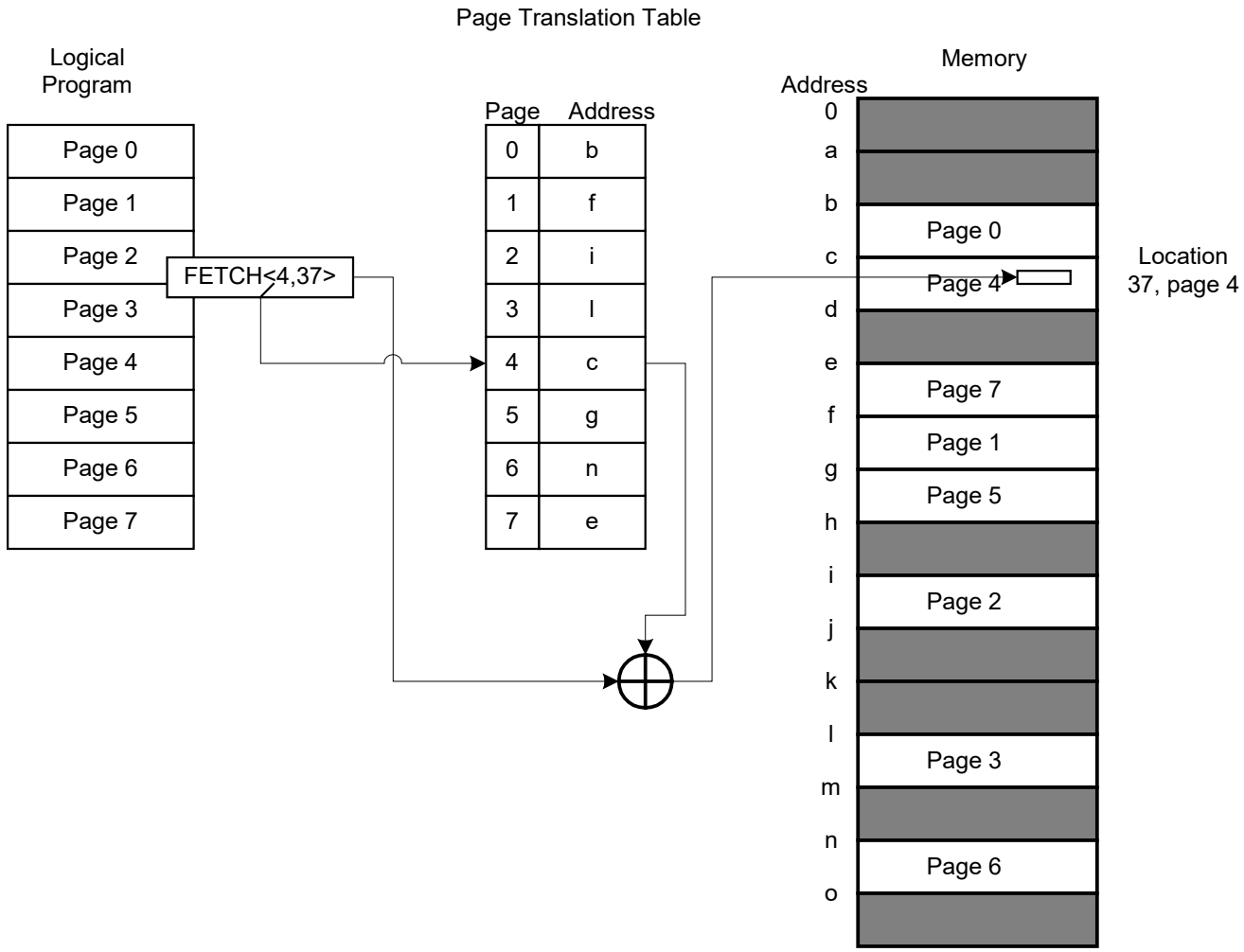
- Memory utilization problem

  Segmentation can cause fragmentation, hence lead to poor memory utilization

  Compaction can help but compacting and updating the look-up table takes time

Memory Protection: Paging

- Program divided to equal-sized pieces called **pages**
- Memory is divided to equal-sized units called **page frames**.
- Address: <page,offset>
- As with segmentation, OS maintains a table (page translation table) of user page numbers and their true addresses in memory



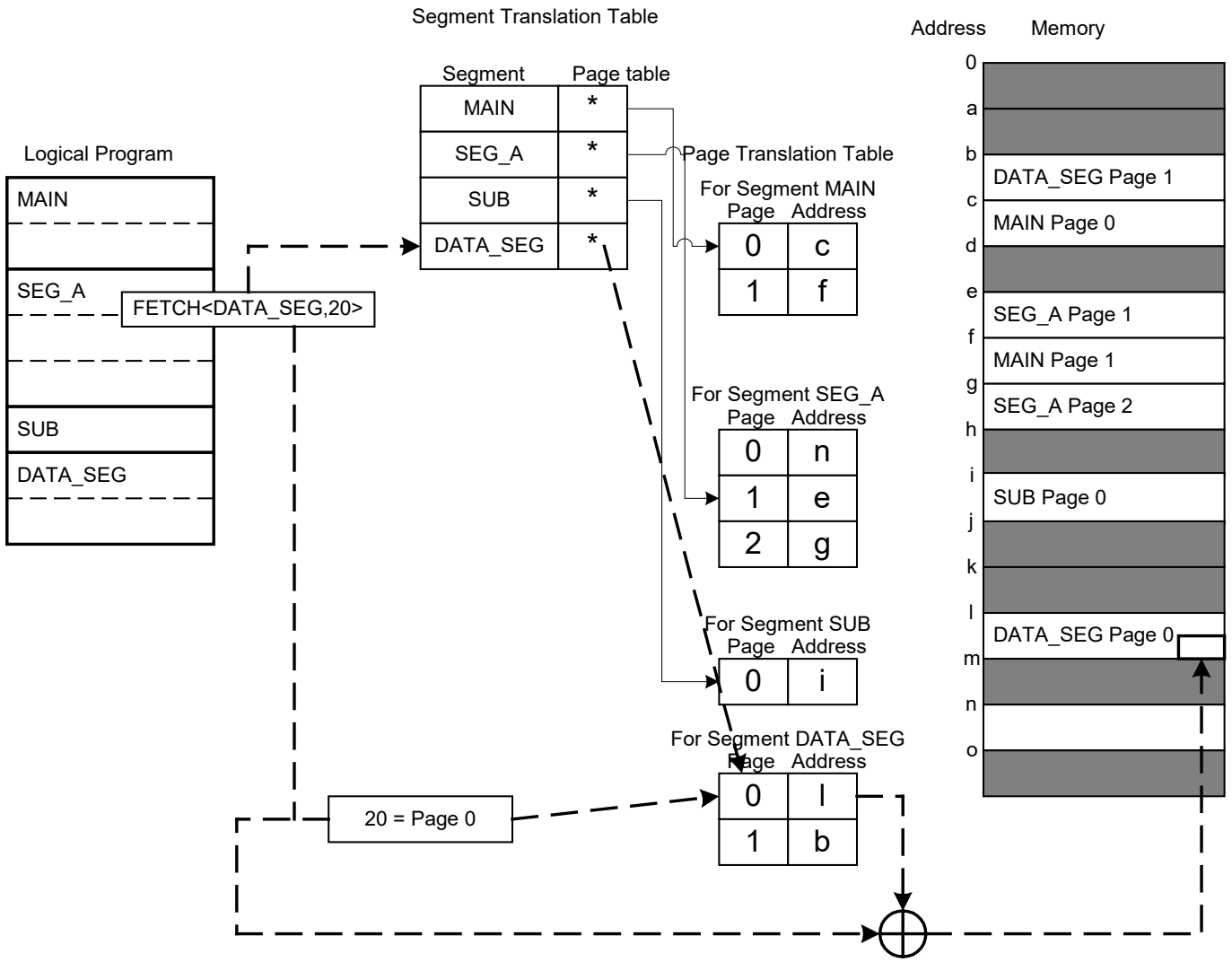
PAGING: PAGE ADDRESS TRANSLATION

- All pages same fixed size : no fragmentation
- No addressing beyond end of page : will be directed to next page
- *page size = 64 bytes ($64=2^6$),*
- *number of pages =10.*
- *Largest offset value 111111= 63 any larger will translate to next page*
- *Say <4,63> ➔ <5,0>*

- **Segments** are logical units, paging is not.
- When additional instruction are added
  - ✓ Subsequent instruction are pushed to lower addresses
  - ✓ Instructions at the end, becomes the start of a new page.
- Problem: no way of establishing if all values on a page should be protected at the same level

Memory Protection: Combined paging and segmentation

- Paging offers **implementation efficiency**
- Segmentation offers **logical protection**
- **Combined:** can use the best of both features



PAGED SEGMENTATION

Control of access to general objects

In OS security, **general objects** are any resources that a process or user might want to access, not limited to files or memory

Objects that needs protection:

- **Memory** – Main RAM segments, virtual memory pages.
- **Files or datasets** – Stored on disk or other storage devices.
- **Executing programs** – Code currently running in RAM.
- **Directories** – Collections of file references.
- **Hardware devices** – Printers, scanners, network cards.
- **Data structures** – Stacks, queues, tables.
- **OS components** – System tables, kernel instructions.
- **Authentication data** – Passwords, keys.
- **The protection mechanism itself** – e.g., ACL configuration

Goals in protecting objects

- **Check every access**, not just the first one.

  E.g. revocations should apply immediately
- **Enforce least privileged**

  A subject should have access to the smallest number of objects necessary to perform some task ➔ only what is needed
- **Verify acceptable usage**

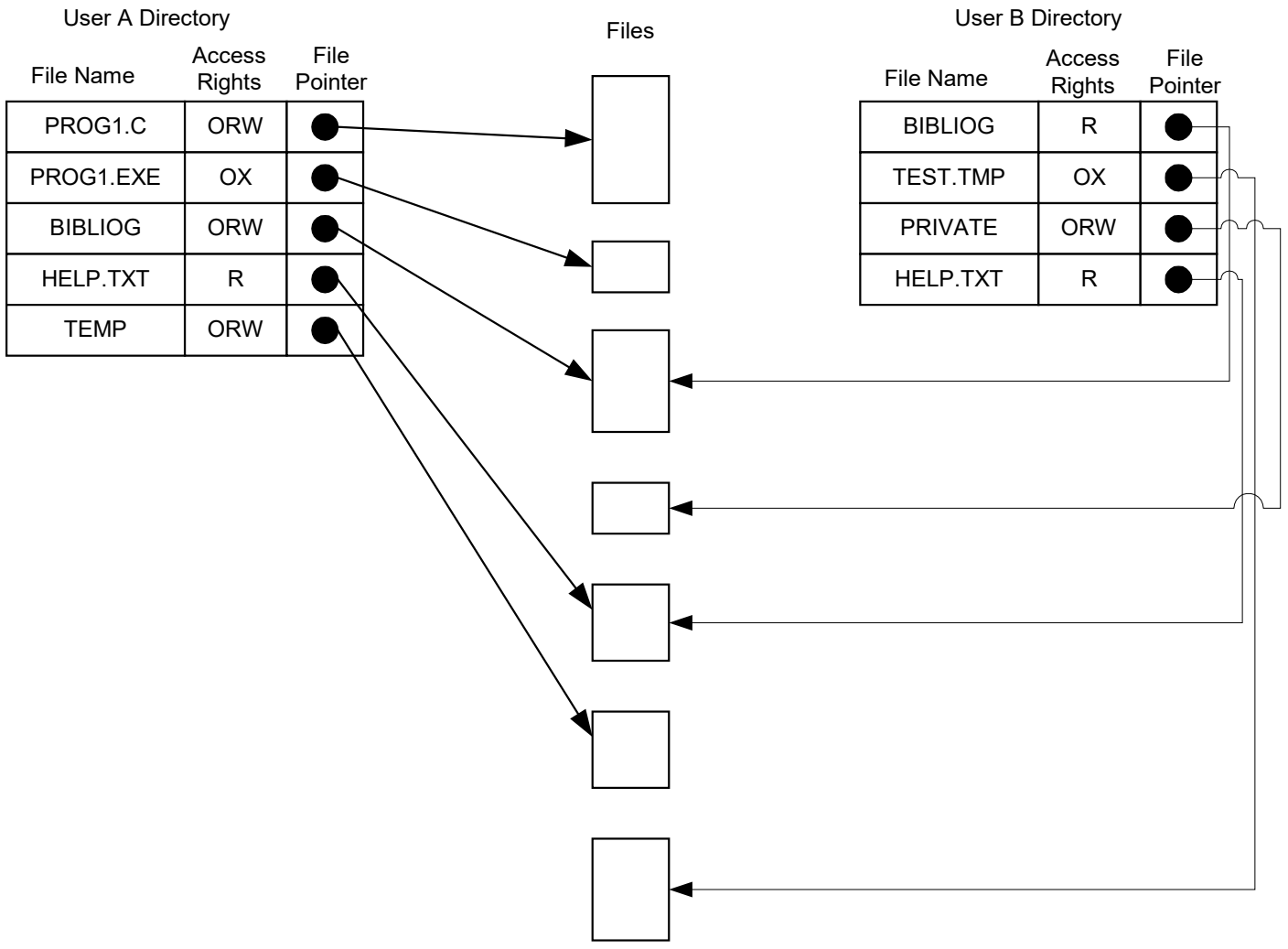
  A subject may use the object only in certain legitimate and appropriate ways (only push, pop, clear on a stack, for instances).

Protection mechanisms for general objects

- Directory
- Access Control List
- Access Control Matrix
- Capability
- Procedure-oriented Access Control

G.O.Protection: Directory

- This mechanism works like a file directory
- Each file has a unique owner who has access rights control over it
- Each user has a file directory, which list all the files the user has access to
- User cannot write to a file directory – to preserve integrity.
- This is done by OS
- Rights to a file include read, write, execute and owner

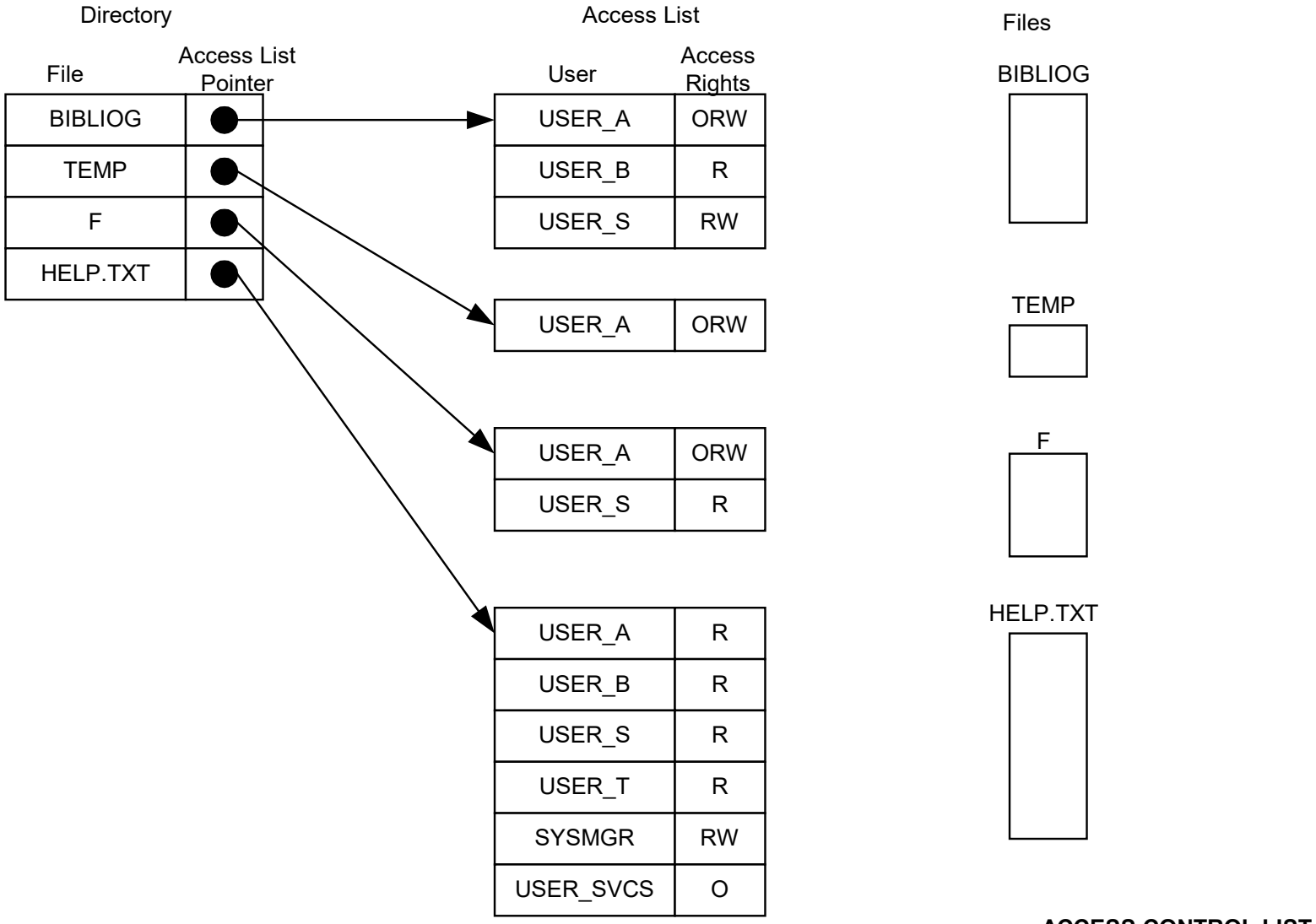


DIRECTORY ACCESS

- Problems
  - List becomes too large if many shared objects are accessible to all users
    - Have entry for objects even if user doesn't use it
    - Deletion must reflect on all directories
  - Difficulty in revoking access
    - Must go thru all directories to revoke rights to file
  - Pseudonyms: when files are renamed
    - 2 owners have different file with same name
      - Solution: owner's designation with file name (owner:filename)
    - When file are renamed (P→Q), then re-request P, problems of conflicting access rights Q-r P-rw

G.O.P.: Access Control List

- One list for each object.
- The list shows all subjects who should have access to the object and what their access is.
- A major advantage with ACLs is that a default access can be assigned to group of subjects as well as specific rights to individual subjects.
- Often used, e.g. Windows NT.



ACCESS CONTROL LIST

G.O.P: Access Control Matrix

- A list of access rights for user and objects
- A list of triples

  Subject
  Object
  Rights
- Problem: searching large numbers of triplets is inefficient

G.O.P: Access Control Matrix

	BIBLIOG	TEMP	F	HELP.TXT	C-COMP	SYS_CLCK	PRINTER
USER_A	ORW	ORW	ORW	R	X	R	W
USER_B	R	-	-	R	X	R	W
USER_S	RW	-	R	R	X	R	W
USER_T	-	-	-	R	X	R	W
SYSMGR	-	-	-	RW	OX	OX	O
USER_SVC	-	-	-	O	X	R	W

ACCESS CONTROL MATRIX

G.O.P: Capability

- An **unforgeable token** that gives the possessor certain rights to an object

- **Users** can

  create objects and specify acceptable access rights R,W,X
  Create new objects and define type of accesses unknown to the system

    E.g. access right to transfer/propagate object
    A → *OT* → B → *O* → C

Modern Access Control Developments (2023–2025)

Development	Description	Example
Role-Based Access Control (RBAC)	Assign rights based on roles instead of individual users.	Linux SELinux roles, Windows Active Directory roles.
Attribute-Based Access Control (ABAC)	Access based on attributes of user, object, and environment.	AWS IAM policies with conditions.
Zero Trust Access Control	Continuous verification, least privilege at all times.	Microsoft Entra ID Conditional Access.
Policy-as-Code	Writing access rules as code for automation.	Open Policy Agent (OPA).

UTM
UNIVERSITI TEKNOLOGI MALAYSIA

File Protection Mechanisms

- **Basic forms of protection**
- **Single file permissions**
- **Per-object and per-user protection**

File Protection Mechanisms

- **File protection mechanisms** prevent unauthorized reading, writing, executing, or deleting of files.

- Basic forms of protection include:
  - **All-None Protection**
    - Files are either fully public or fully private. Example: Early IBM OS allowed open access unless password-protected.
  - **Group Protection**
    - Users are grouped, and access rights are assigned per group. Common in Unix/Linux systems using owner/group/others model.

All-None Protection

- An early file protection method used in some older systems (e.g., early IBM OS).
- Protection based on trust and ignorance (open only with filename that you know)
- Not acceptable.

Problems with this approach

- **Lack of trust** in large systems:

  Anyone with the file name can open public files.
  Difficult to limit access only to specific trusted users.
- **Complexity** for password-protected files:

  Each file required a password entry every time it was accessed.
  Managing multiple file passwords was cumbersome.
- **Security weakness**:

  File listings reveal file names, removing the “ignorance” barrier.

Group Protection

- Files are protected based on **group membership**.
- A group is a set of users with a **common relationship** (e.g., project team).
- Each user belongs to **one group** (in the simplest model).
- Files are assigned permissions for:
  1. **Owner** (user who created it) - **U**
  2. **Group** (trusted collaborators) - **G**
  3. **Others** (everyone else - **O**

u+r+w+x, g+r+w-x, o+r-w-x

Advantages

- **Easy to implement**:

  OS identifies users by User ID (UID) and Group ID (GID) at login.
  Each file stores its owner's UID and GID in the directory.
- Simple for collaboration within a group.
- More fine-grained than all-none protection.

Limitation

- In the basic model, a user can only belong to one group at a time — limits flexibility for multi-project work (modern systems support multiple groups per user).

Feature	All-None Protection	Group Protection
Access Control Method	All users can access (public) or none (password-protected)	Access based on owner, group, and others
Granularity	Very coarse – only 2 levels (public/private)	Medium – 3 main levels with specific rights
Ease of Use	Simple – just know filename or password	Moderately easy – based on group membership
Security Strength	Weak – public files easily accessed; passwords may be guessed	Stronger – group members only, others restricted
Example OS Usage	Early IBM OS	UNIX/Linux, modern OS group-based permissions
Advantages	Easy to understand, no complex permissions	Supports collaboration, more control than all-none
Disadvantages	No fine-grained control, file listings expose names, password management overhead	Limited if only 1 group per user; needs management of group membership

Single Permission

- **Single permission** means the access control is applied **per file**, not for a group of files or a directory.

- Types of single file permissions:
  - **Password or other token**
  - **Temporary acquired permission**

Passwords and Tokens

- In the **Password or Token** method, each file has its own **unique password or access token.**
- This method can be used to control **specific types of access**, such as:
  - **Read-only**
  - **Write-only**
  - **Read/Write**
  - **Full control**

Passwords and Tokens

- A password is set for a file by the file owner or administrator.
- Anyone attempting to access that file must **enter the correct password** or present the required token.
- The system verifies the input before allowing access.
- Tokens may be:
  - **Physical devices** (e.g., USB key, smart card)
  - **Digital keys** (e.g., software license keys)
  - **Biometric templates** (if used in a secure container)

- **Write-only password**: Protects the file so only certain people can modify it
- **Any access password**: Needed for both read and write operations.
- The concept is similar to **having a unique group** for each file:
  - "Group" = all users who know the password for that file.

Real-World Example

- **Microsoft Office**: Password-protected Word/Excel files.
- **PDF Security**: Restrict opening or editing with separate passwords.
- **Encrypted Archives**: ZIP/RAR files with passwords for extraction.

Advantages

- **Fine-grained control**: Different passwords for different files.
- **File-level security**: No dependency on broader group permissions.
- **Easy sharing**: Just share the password with trusted individuals without changing system-level settings.

Disadvantages

- **Password management**: Too many passwords to remember/manage for multiple files.
- **Security risk**: If the password is leaked, anyone can access the file.
- **Revocation difficulty**: To remove a user’s access, you must change the password and inform all other authorized users.
- **User inconvenience**: Frequent prompts for passwords can slow down workflows.

Temporary acquired permission

- **SetUID** (suid) is a permission flag in Unix/Linux-like operating systems.
- When applied to an **executable file**, it **temporarily grants the user who runs the program the file owner's privileges** during the program's execution.
- This allows a non-privileged user to perform a **specific privileged action** without being given full access rights permanently.
- Sometimes a user needs to perform a task that requires higher privileges than they normally have.
- Instead of giving them permanent elevated rights, **SetUID** allows a **controlled privilege escalation** just for the duration of that program.
- This reduces the security risk compared to making the user a full system administrator.

- **Normal Execution (No SetUID)**

  The program runs with the privileges of the user executing it.

- **With SetUID**

  The **program runs with the privileges of the file owner**, not the executor.

  If the file owner is root (superuser), the program executes with root privileges.

- **Example Scenario**:

  The */usr/bin/passwd* program in Linux allows users to change their own password.

  The password file *(/etc/shadow*) is owned by root and not directly accessible by normal users.

  passwd has the SetUID bit set, so when a user runs it, the program temporarily runs with root privileges, allowing it to safely update */etc/shadow*.

Setting and Identifying SetUID

Command to set SetUID bit:

chmod u+s filename

Checking if a file has SetUID set:

ls -l /usr/bin/passwd-rwsr-xr-x 1 root root 54256 Jan 1 12:34 /usr/bin/passwd

Here, the **s** in -rws indicates the SetUID bit for the user (owner) category.

Breakdown of the Output

-rwsr-xr-x 1 root root 54256 Jan 1 12:34 /usr/bin/passwd

File Permissions (-rwsr-xr-x)

- - → Regular file
- rws → Owner (root) has **read**, **write**, and **setuid** permissions
- r-x → Group has **read** and **execute** permissions
- r-x → Others have **read** and **execute** permissions

Special Bit: s in rws

- The s in place of the owner's execute bit means the **setuid** bit is set.
- This allows the program to run with the **file owner's privileges** (in this case, root), even when executed by another user.
- This is **critical** for passwd, because changing a password requires access to system files like /etc/shadow, which are normally restricted.

Ownership

- root root → Owned by user root and group root

Size and Timestamp

- 54256 → File size in bytes
- Jan 1 12:34 → Last modification date and time

File Path

- /usr/bin/passwd → Location of the executable

Advantages

- **Controlled Privilege Escalation:** Grants higher privileges only during program execution.
- **Security Boundaries Maintained:** User doesn't get permanent superuser rights.
- **Enables Necessary Access:** Allows certain critical system tasks to be performed by normal users.

Disadvantages / Security Risks

- **Exploitable if the Program is Insecure:**

If a SetUID program has vulnerabilities (e.g., buffer overflow), an attacker might exploit it to gain permanent root access.

- **Privilege Misuse:**

Poorly written SetUID programs may unintentionally allow actions outside their intended scope.

- **Limited to Supported OS Types:**

Not all operating systems have SetUID functionality (primarily Unix/Linux-based).

Feature	Password or Token	Temporary Acquired Permission (SetUID)
Definition	Each file has a unique password or token for access control.	A user temporarily gains the file owner's access rights while running the file.
How It Works	User must enter the file's password or present a token to gain access. Passwords/tokens can control read, write, or full access.	If the setuid bit is set on an executable, running it gives the executor the file owner's privileges during execution.
Advantages	Fine-grained, file-level security; easy to share access without changing system-level settings.	Allows controlled privilege escalation; lets users perform specific tasks without full access to all files.
Disadvantages	Password management burden, risk if leaked, revocation requires changing password and informing all authorized users.	Can be exploited if the program is insecure; limited to specific OS types that support setuid; requires careful coding.
Real-World Examples	Password-protected Word/Excel documents; password-protected ZIP/PDF files; encrypted archive files.	'passwd' command in Unix/Linux lets normal users change their password by temporarily accessing system password file.

User Authentication in OS

- User authentication is the process the OS uses to **verify the identity** of a person or process before granting access to resources.
- It ensures that only legitimate users can access files, applications, and system services.
- **Analogy:** It's like showing your ID to a security guard before entering a secure building.

Steps in Authentication

- **Identification** – The user claims an identity.

  Example: Entering a username.

- **Authentication** – The system verifies the claim.

  Example: Checking the password against the stored hash.

- **Authorization** (after authentication) – The OS assigns permissions based on the user's profile.

Authentication Factors

Authentication methods are usually based on one or more of these **factors.**

Factor Type	Description	Example
Something you know	A secret known only to the user.	Password, PIN
Something you have	A physical or digital object.	Smart card, security token
Something you are	A biological trait.	Fingerprint, facial recognition
Somewhere you are	Geographical location.	GPS-based login restrictions
Something you do	Behavioral patterns.	Typing rhythm, swipe pattern

Multi-Factor Authentication (MFA) combines two or more of these factors for higher security.

Common Methods in OS

- Identification and Authentication
- Password-based authentication
- One Time Passwords (OTP)
- Biometric Authentication
- Token-based Authentication

Password-based Authentication

- The OS stores **hashed** and sometimes **salted** passwords.
- At login, the entered password is hashed and compared to the stored hash.
- Common in all OS types (Windows, Linux, macOS).
- **Weaknesses:**

  Vulnerable to brute force, dictionary, and phishing attacks.
  Users often choose weak passwords.

Hashing is a one-way mathematical transformation:

- Input: Your password (e.g., "MySecureP@ss2025")
- Output: A fixed-length **hash value** (e.g., "7c6a180b36896a0a8c02787eeafb0e4c")

Adding a Salt

- **Salt** = A random value added to the password **before hashing**
- Each user gets a unique salt
- Stored alongside the hash in the password file.
- Ensures even identical passwords produce different hashes

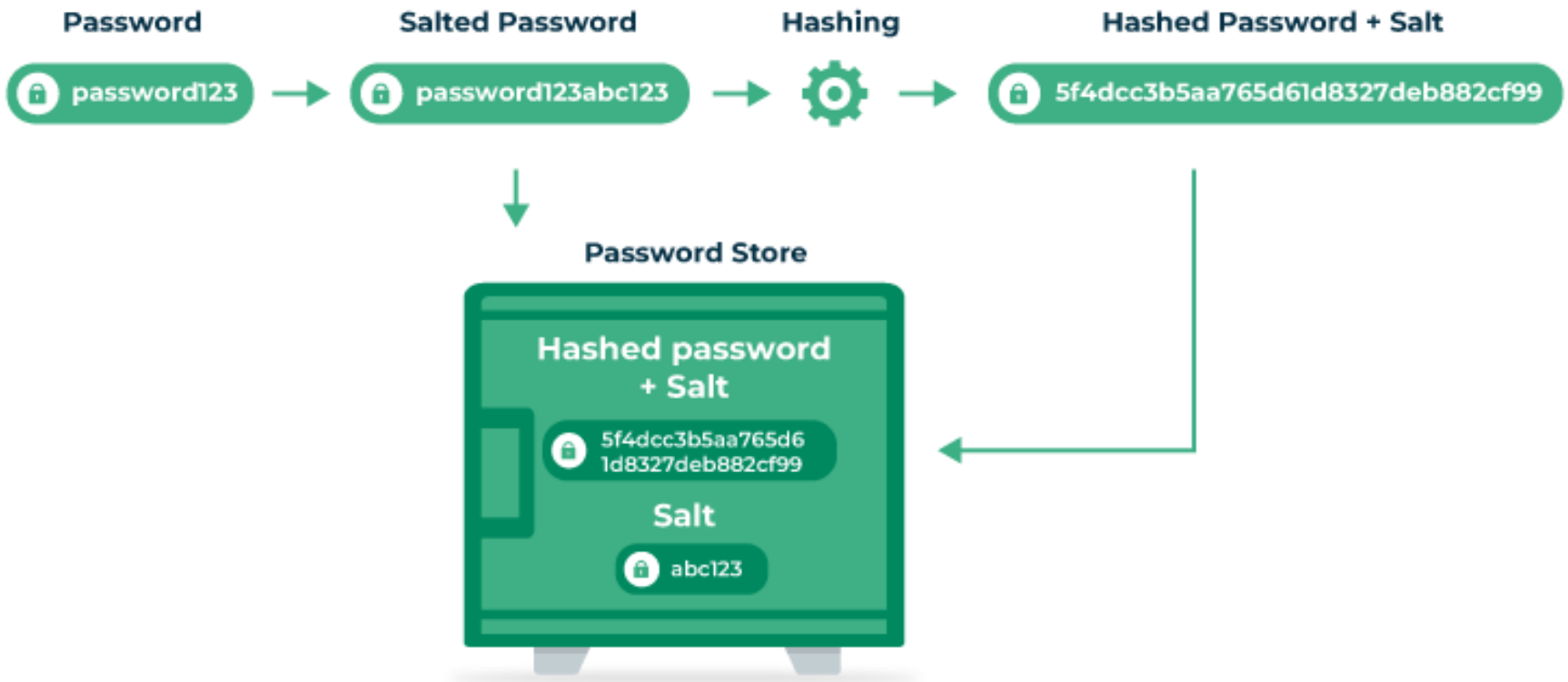
Password: "123456“
Salt: "k9&2zQ“
Hashed value = Hash("123456k9&2zQ") = e93ac7b65f1...

User A and User B may both have "123456" as a password, but **their salts differ**, so their hashes are different.

How the OS Uses Salt and Hash

- **When creating or changing a password:**
  - ✓ Generate a random salt.
  - ✓ Concatenate salt + password.
  - ✓ Hash the result using a secure algorithm.
  - ✓ Store the **hash** and the **salt** in the password database.

- **When logging in:**
  - ✓ Retrieve the stored salt for that user.
  - ✓ Append salt to the entered password.
  - ✓ Hash this combination.
  - ✓ Compare the computed hash with the stored hash — if they match, authentication succeeds.



https://www.geeksforgeeks.org/techtips/what-is-password-salting/

Example – Linux /etc/shadow file entry

alice:6saltstring$hashedpassword:19000:0:99999:7:::

- 6 → SHA-512 is used.
- saltstring → The salt.
- hashedpassword → The salted hash of the password

Types of Attacks Mitigated by Password Salting

- Salting helps protect against several types of attacks on stored password hashes, including:
  - **Rainbow Table Attacks**: Attackers use precomputed tables of hash values to quickly reverse-engineer passwords. Salting ensures that each password has a unique hash, making precomputed tables ineffective.
  - **Brute Force Attacks**: These attacks involve trying every possible password until the correct one is found. Salting doesn't directly prevent brute force attacks, but it does make them more time-consuming by introducing a unique value for each password.
  - **Precomputed Hash Attacks**: Attackers who have access to a hash database may attempt to compare stored hashes against precomputed hashes for common passwords. With salting, even if common passwords are used, they will have different hashes due to the unique salts, rendering precomputed attacks useless.

Modern Enhancements

- **Pepper**: A secret value added to the password before hashing, stored separately from the database (e.g., in the OS secure storage).
- **Key Stretching**: Using algorithms like bcrypt or Argon2 that make hashing intentionally slow and memory-intensive, making brute force attacks much harder.

One Time Passwords (OTP)

- Valid for only one login session or transaction.
- Generated by:
  - Hardware tokens.
  - Mobile apps (Google Authenticator).
  - SMS (less secure).
- Often used as a **second factor** in MFA.

Two common OTP generation methods are used by OS authentication systems

- Time-based OTP (TOTP)
- HMAC-based OTP (HOTP)

Two Main OTP Approaches

1.Time-based OTP (TOTP)

- Based on the current time and a shared secret.
- The OTP changes every fixed interval (e.g., 30 seconds).
- Example: Google Authenticator, Microsoft Authenticator.

2.HMAC-based OTP (HOTP)

- Based on a counter that increments with each OTP generation.
- Example: Some hardware tokens used for VPN logins.

Time-based OTP (TOTP)

- Relies on the current time as a moving factor.
- Both the OS (or authentication server) and the user's token/app share:
  - **A secret key** (stored securely on both sides)
  - **The same time reference**
- Steps:
  1. User initiates login (e.g., SSH to a Linux server).
  2. The OS prompts for username + password.
  3. After verifying the password, the OS prompts for OTP.
  4. User opens an authenticator app (Google Authenticator, Authy) which calculates OTP:
     - OTP = Hash(secret_key + current_time_interval)
  5. User enters the OTP.
  6. OS calculates the expected OTP using its own stored secret and current time.
  7. If they match → Access granted.

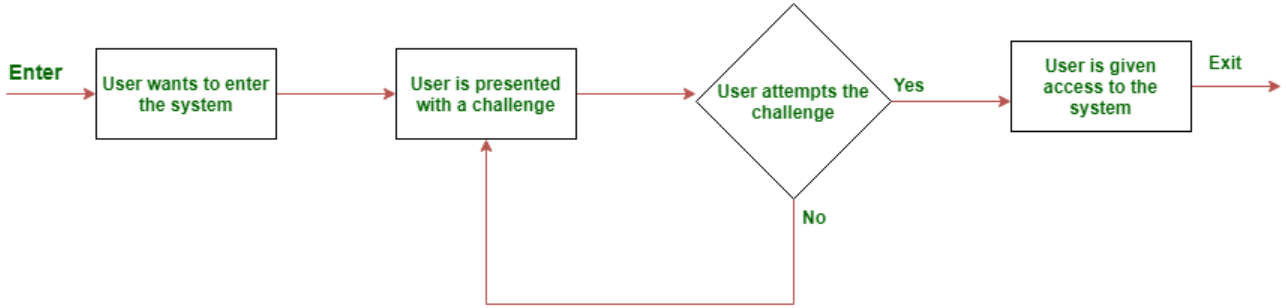
HMAC-based OTP (HOTP)

- Uses a counter instead of the current time.
- Each time an OTP is generated, the counter increments.
- Steps:
  1. OS and token share a secret key and counter.
  2. Each login attempt generates:
     - OTP = HMAC(secret_key + counter)
  3. OS compares OTP from user with its own calculation.
  4. If correct, increments counter on both sides.

**HMAC (Hash-based Message Authentication Code)

Challenge-Response Authentication (CRA)

- OTP (One-Time Password) is **a form of Challenge-Response Authentication**, especially when used in dynamic contexts like multi-factor authentication (MFA) or transaction verification.



It's a security mechanism where:

- The **server issues a challenge** (e.g., a random number or token).
- The **client must respond correctly,** often by computing or retrieving a value based on the challenge.
- The goal is to **prove identity** without transmitting reusable secrets.

Diagran source: https://www.geeksforgeeks.org/computer-networks/challenge-response-authentication-mechanism-cram/

Biometric Authentication

- Uses unique biological traits (fingerprints, retina scans, facial features).
- Increasingly common in OS logins (e.g., Windows Hello, Touch ID).
  Must store and process biometric data securely.

Token-based Authentication

- Requires a physical or virtual device (USB key, smart card, NFC card).
- The token proves the user's identity during login.

Attacks on Authentication in OS

- **Brute Force Attack** – Trying all possible passwords.
- **Dictionary Attack** – Trying commonly used passwords.
- **Credential Stuffing** – Using stolen credentials from another breach.
- **Social Engineering** – Tricking the user into revealing credentials.
- **Keylogging** – Capturing keystrokes to steal passwords.

How to choose a good password?

- Use other characters besides A-Z
- Choose long passwords
- Avoid actual names or words
- Choose an unlikely password
- Change regularly
- Don’t write it down
- Don’t tell anyone else

Recent Developments in OS Security

- **Secure Boot & Trusted Platform Module (TPM)** – Prevent boot-time tampering.
- **Mandatory Access Control (MAC)** in systems like SELinux and AppArmor.
- **Virtualization-based Security (VBS)** for isolation of sensitive components.
- **Patch Guard & Kernel Integrity Checks** to prevent rootkits.
- **AI-driven Threat Detection** for adaptive security monitoring.

UTM
UNIVERSITI TEKNOLOGI MALAYSIA

Summary

- **OS Security** : Measures and mechanisms designed to protect the operating system, its components, and managed data from unauthorized access, misuse, or attacks.
- **Common Security Breaches**: Interruption, Interception, Modification, Fabrication
- **Protected Objects** (Memory, Files, I/O Devices, etc) and **Protection Methods** (Authentication, Access control, Memory protection, File protection mechanisms)