The Deadlock Problem

- A set of **blocked processes each holding** a resource and waiting to acquire a resource held by another process in the set.

- Example
  - System has 2 tape drives.
  - P_1 and P_2 each hold one tape drive and each needs another one.

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource.
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption:** a resource can be released only voluntarily by the process holding it, **after that process has completed its task**.

Deadlock Characterization

- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, ..., P_{n-1} is waiting for a resource that is held by P_n, and P_0 is waiting for a resource that is held by P_n.

Recovery from Deadlock: Resource Preemption

- Selecting a victim – minimize cost.
- Rollback – return to some safe state, restart process for that state.
- Starvation – same process may always be picked as victim, include number of rollback in cost factor.

Resource-Allocation Graph

A set of **vertices** V and a set of **edges** E.

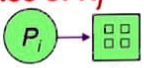
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- **request edge** – directed edge $P_1 \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

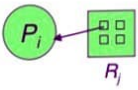
- **Process**

- **Resource Type with 4 instances**

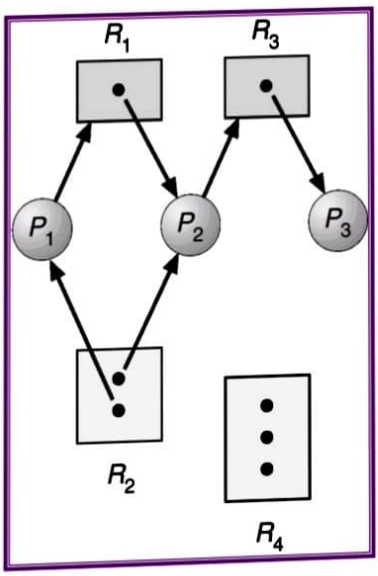
- P_i **requests instance of** R_j



- P_i **is holding an instance of** R_j



Example of a Resource Allocation Graph



A set of **vertices** ***V*** and a set of **edges** ***E***.

- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- **request edge** – directed edge $P_1 \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

Swapping vs Overlays

Swapping	Overlays
Swaps entire process between RAM and disk	Loads only required portions of a program into memory
Used to free up memory by moving inactive processes to disk	Used when program size is larger than available memory
Entire process is swapped in/out	Only specific sections (functions or modules) are loaded as needed
Used in modern OS for multitasking (e.g., Virtual Memory)	Used in early systems to handle large programs in limited memory
High, as it requires moving entire process	Lower, as only portions of the program are loaded when needed

Internal Fragmentation vs External Fragmentation

Internal Fragmentation	External Fragmentation
Unused memory within an allocated block	Free memory scattered across different blocks
Fixed-size partitioning or block allocation	Dynamic allocation of variable-sized memory blocks
If a process needs 5 KB but is given a 6 KB block, 1 KB is wasted	If a process of 6 KB is removed from a 10 KB block, 4 KB remains unusable if no process fits
Use paging to allocate exact memory needed	Compaction or segmentation to group free space together

Paging vs Segmentation

Paging	Segmentation
Divides memory into fixed-size pages	Divides memory into variable-sized segments based on logical divisions
Eliminates external fragmentation	Supports logical grouping of related data/code
Fixed page size (e.g., 4KB)	Variable segment size
Uses a **page table** to map logical to physical addresses	Uses a **segment table** with base and limit registers
Suffers from **internal fragmentation**	Suffers from **external fragmentation**
Used in modern operating systems for virtual memory	Used in systems requiring better logical organization

Definition of Dynamic Loading

Dynamic loading is a memory management technique where a program loads its modules (such as functions, libraries, or classes) into memory **only when they are needed** during execution, rather than loading everything at the start.

Strategies for Managing Memory in Kernel

1. **Contiguous Memory Allocation**
2. **Paging**
3. **Segmentation**
4. **Slab Allocation**
5. **Buddy System**
6. **Virtual Memory & Demand Paging**
7. **Kernel Memory Swapping**

Logical Address vs Physical Address

Logical Address	Physical Address
Address generated by the CPU during program execution.	Actual address in memory (RAM) after address translation.
Seen by the user and processes.	Managed by the OS and hardware (MMU).
Needs to be translated into a physical address by MMU.	No translation needed; directly used by the memory unit.
Independent of actual physical memory location.	Dependent on the available memory in RAM.
Used in **virtual memory** and **paging systems**.	Used to access actual data stored in RAM.
A program requests memory at **0x0045F000** (logical).	The MMU translates it to **0xABCDF000** (physical).
Provides process isolation (each process has its own logical space).	Physical addresses are controlled to prevent unauthorized access.

Main Function of Memory Management Unit (MMU)

1. Translates **logical addresses** to **physical addresses**.
2. Handles **paging and segmentation**.
3. Enforces **memory protection** and access control.
4. Supports **virtual memory** by managing page tables.

Purpose of Paging the Page

1. **Eliminates external fragmentation** by dividing memory into fixed-size pages.
2. **Allows efficient memory allocation** for processes.
3. **Supports virtual memory** by enabling page swapping.
4. **Improves memory utilization** by loading only needed pages.

- A **demand-paging system is similar to a paging system with swapping**

- A **lazy swapper** is used. It never swaps a page into memory unless that page will be needed.

- A swapper manipulate entire processes, **whereas a pager** is concerned with the individual pages of process.

How to satisfy a request of size ***n*** from a list of free holes?

- **First-fit**: Allocate the ***first*** hole that is big enough

- **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole

- **Worst-fit**: Allocate the ***largest*** hole; must also search entire list
  - Produces the largest leftover hole

First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Major Problems in Implementing Demand Paging

1. **Page Fault Overhead** – Frequent page faults slow down execution.
2. **Increased Memory Access Time** – Requires address translation, adding delay.
3. **Thrashing** – Excessive page swapping reduces system performance.
4. **Storage Overhead** – Requires **page tables** and **swap space management**.
5. **Replacement Policy Complexity** – Choosing which page to evict affects efficiency.
6. **Security Concerns** – Improper handling may expose data during swapping.

How Does a System Detect Thrashing?

- Monitors the **high page fault rate** and **low CPU utilization**.
- Uses the **working set model** to check if a process has enough frames.
- If adding more frames reduces page faults, thrashing is detected.

Use of Valid and Invalid Bits in Paging

- **Valid Bit (1):** The page is in memory and can be accessed.
- **Invalid Bit (0):** The page is not in memory or belongs to another process, causing a page fault if accessed.

Definition of Reference String

- A **sequence of memory page accesses** used to analyze page replacement algorithms.
- Example: `{1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7}`

Effective Access Time (EAT)

- The average time required to access memory, considering **TLB hit/miss rates** and **page faults**.
- Formula:

$$EAT = (1 - p) \times \text{Memory Access Time} + p \times (\text{Page Fault Overhead})$$

where **p** = page fault rate. ↓

Global vs Local Page Replacement Algorithm

Global Page Replacement	Local Page Replacement
Allows a process to replace any page in memory.	A process can only replace its own pages.
Dynamic (frames are shared among processes).	Fixed (each process has a set number of frames).
Can cause thrashing if one process takes too many frames.	Prevents thrashing but may lead to inefficient frame usage.
LRU, FIFO, Clock (Global versions).	Working Set, Local LRU.

Demand Paging vs Pure Paging

Demand Paging	Pure Paging
Loads pages only when needed.	Loads all pages into memory before execution.
High initially but reduces over time.	No page faults after loading all pages.
More efficient; only required pages are loaded.	May waste memory if some pages are never used.
Slower initially but improves.	Faster execution start, but may use more memory.

Examples

- **Paging:** Used in **Windows and Linux** virtual memory.
- **Segmentation:** Used in **Intel x86 protected mode**.
- **Paging with Segmentation:** Used in **Multics, IBM OS/2**, where segmentation defines logical units, and paging manages memory allocation.

Significance of LDT and GDT

1. **LDT ensures process isolation** by maintaining private segment descriptors for each process.
2. **GDT provides system-wide access** to shared segments like kernel memory and task state segments.
3. **LDT allows each process to have its own memory layout**, improving security and flexibility.
4. **GDT remains constant**, reducing overhead when switching processes.
5. **Together, LDT and GDT enable segmentation-based memory management** in systems like Intel x86.

Methods for Handling Deadlock

1. **Deadlock Prevention** – Ensures at least one of the four necessary conditions (Mutual Exclusion, Hold and Wait, No Preemption, Circular Wait) is never satisfied.
2. **Deadlock Avoidance** – Uses algorithms like **Banker's Algorithm** to allocate resources safely.
3. **Deadlock Detection and Recovery** – Detects deadlocks using resource allocation graphs and recovers by **process termination** or **resource preemption**.
4. **Ignoring Deadlocks (Ostrich Algorithm)** – Used in systems where deadlocks are rare and handling them is costly (e.g., most OSes like Linux and Windows).