

UNIT 4

Deadlocks

(4 Hrs) — Operating System (CACS251)

4.1 System Model

A **deadlock** is a situation in which a set of processes is **permanently blocked** because each process holds a resource and is waiting for another resource held by some other process in the set. None of them can proceed, so none will ever release its resources.

A process utilises a resource in exactly three steps: **request** (if the resource is not available the process must wait), **use**, and **release**. Resource types may be physical (printers, tape drives, memory space, CPU cycles) or logical (files, semaphores, database records, monitors). Each resource type may have several identical **instances**.

Everyday analogy — Traffic gridlock

Four cars enter a crossing from four sides at the same time. Each car needs the lane occupied by the car on its right. Nobody can move until someone backs up — a deadlock.

4.2 System Resources: Preemptable and Non-Preemptable

Preemptable resource	Non-preemptable resource
Can be taken away from the owning process with no ill effects	Cannot be taken away without causing the computation to fail
Example: main memory (contents can be swapped to disk), CPU	Example: CD/DVD recorder, printer in the middle of a job, tape drive, a locked file
Deadlocks involving them can be resolved by reallocation	Deadlocks usually involve these resources

Resource deadlocks concern **non-preemptable** resources (and also **reusable** ones such as printers vs. **consumable** ones such as messages and interrupts).

4.3 Conditions for Resource Deadlocks

Coffman *et al.* (1971) showed that a deadlock can occur **only if all four** of the following conditions hold simultaneously:

#	Condition	Meaning
1	Mutual exclusion	Each resource is either assigned to exactly one process or is available (non-sharable).
2	Hold and wait	A process holding resources granted earlier can request new resources.
3	No preemption	Resources previously granted cannot be forcibly taken away; the process must release them explicitly.
4	Circular wait	There is a circular chain of two or more processes, each waiting for a resource held by the next member of the chain.

Note: If even one of the four conditions is absent, deadlock is impossible. This observation is the basis of deadlock prevention.

4.4 Deadlock Modeling

Holt (1972) showed how the four conditions can be modelled using directed graphs called **Resource Allocation Graphs (RAG)**. The graph has two types of nodes: **processes** (drawn as circles) and **resources** (drawn as squares; dots inside a square represent instances). There are two types of edges:

- **Request edge** $P \rightarrow R$ — process P has requested resource R and is waiting.
- **Assignment edge** $R \rightarrow P$ — resource R has been allocated to process P.

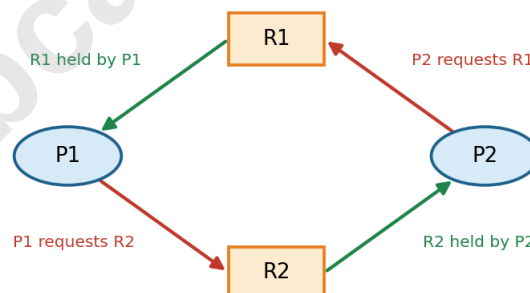


Figure 4.1 — Resource allocation graph containing a cycle (deadlock)

Here R1 is held by P1, which requests R2; R2 is held by P2, which requests R1. The cycle $P1 \rightarrow R2 \rightarrow P2 \rightarrow R1 \rightarrow P1$ is a deadlock.

Interpreting a RAG

- If the graph contains **no cycle** → no deadlock.
- If the graph contains a cycle and each resource type has **only one instance** → deadlock exists (cycle is necessary and sufficient).

- If the graph contains a cycle and some resource types have **several instances** → deadlock *may* exist (cycle is necessary but not sufficient).

Example — Two processes, two resources (program view)

Process A: request R; request S; use; release R; release S.

Process B: request S; request R; use; release S; release R.

If the scheduler interleaves as A requests R, B requests S, A requests S (blocks), B requests R (blocks) → deadlock. If A runs to completion first, or the requests are ordered the same way in both processes, no deadlock occurs.

4.5 The Ostrich Algorithm

The simplest approach: **stick your head in the sand and pretend there is no problem**. Deadlock is ignored under the assumption that it occurs rarely and that the cost of preventing or detecting it is higher than the cost of occasionally rebooting.

- Used by **UNIX and Windows** for most resources (a full process table, for instance, is not checked for deadlock).
- Reasonable trade-off between *convenience* and *correctness* — mathematicians want deadlock to be prevented at any cost; engineers ask how often it really happens.
- If deadlocks are rarer than other failures (crashes, hardware faults), the loss of convenience from prevention is not justified.

4.6 Methods of Handling Deadlocks

Method	Idea	Cost
1. Ignore the problem (Ostrich)	Assume deadlock never occurs	None; occasional reboot
2. Deadlock prevention	Structurally negate one of the four Coffman conditions	Low resource utilisation, reduced throughput
3. Deadlock avoidance	Carefully allocate resources only if the system stays in a safe state	Needs advance information; run-time overhead
4. Detection and recovery	Allow deadlock, detect it, then recover	Detection overhead and loss of work on recovery

4.7 Deadlock Prevention

Prevention ensures that at least one of the four necessary conditions can never hold.

Condition attacked	Technique	Drawback
Mutual exclusion	Make resources sharable where possible (read-only files); spool printers so that only the spooler daemon requests the printer	Not all resources can be shared or spooled (e.g., a database record being updated)
Hold and wait	Require processes to request all resources at once before starting, or to release all held resources before requesting new ones	Low resource utilisation; processes may not know their needs in advance; starvation of processes needing popular resources
No preemption	If a process holding resources requests one that is unavailable, all its held resources are preempted (implicitly released); it restarts only when it can regain all	Can be applied only to resources whose state can be saved and restored (CPU registers, memory); not to printers
Circular wait	Impose a global numbering of resources ; processes must request resources in increasing order only	Finding a suitable ordering is hard; resources may be requested in an unnatural order

Example — Resource ordering

Assign numbers: 1 = scanner, 2 = printer, 3 = tape drive. A process holding the printer (2) may next request only the tape drive (3), never the scanner (1). Because every process climbs the numbers, a cycle can never form.

4.8 Deadlock Avoidance

Avoidance requires that the system has some **additional a priori information** — e.g., each process declares the **maximum number** of resources of each type that it may need. The resource-allocation state is defined by the number of available and allocated resources and the maximum demands. The system grants a request only if the resulting state is **safe**.

4.8.1 Safe and Unsafe States

- A state is **safe** if there exists a **safe sequence** $\langle P_1, P_2, \dots, P_n \rangle$ — an ordering in which every process can obtain its maximum remaining needs using the currently available resources plus those released by all earlier processes in the sequence — so all can finish.
- A state is **unsafe** if no such sequence exists. An unsafe state *may* lead to deadlock (not necessarily); a deadlocked state is always unsafe. Safe $\rightarrow$ no deadlock.

4.8.2 Banker's Algorithm (Dijkstra, 1965)

The Banker's algorithm is named by analogy to a banker who never allocates cash so that he can no longer satisfy the needs of all customers. It works for **multiple instances** of each resource type. Let n = number of processes, m = number of resource types.

Data structure	Size	Meaning
Available	vector [m]	Number of available instances of each resource type
Max	$n \times m$	$\text{Max}[i][j]$ = maximum demand of process i for resource j
Allocation	$n \times m$	$\text{Allocation}[i][j]$ = instances of j currently held by i
Need	$n \times m$	$\text{Need}[i][j] = \text{Max}[i][j] - \text{Allocation}[i][j]$ (remaining need)

Safety Algorithm

- **Step 1:** Let $\text{Work} = \text{Available}$ and $\text{Finish}[i] = \text{false}$ for all i .
- **Step 2:** Find a process i such that $\text{Finish}[i] = \text{false}$ and $\text{Need}[i] \leq \text{Work}$. If none exists, go to step 4.
- **Step 3:** $\text{Work} = \text{Work} + \text{Allocation}[i]$; $\text{Finish}[i] = \text{true}$; go to step 2.
- **Step 4:** If $\text{Finish}[i] = \text{true}$ for all i the state is **safe**; otherwise it is unsafe.

Resource-Request Algorithm

When process P_i makes a request vector $\text{Request}[i]$: (1) if $\text{Request} \leq \text{Need}$, continue, else error (process exceeded its maximum); (2) if $\text{Request} \leq \text{Available}$, continue, else P_i must wait; (3) *pretend* to allocate: $\text{Available} -= \text{Request}$; $\text{Allocation}[i] += \text{Request}$; $\text{Need}[i] -= \text{Request}$; (4) run the safety algorithm — if the new state is safe the allocation is made, otherwise the pretended allocation is undone and P_i waits.

Example — Banker's algorithm

Five processes P_0 – P_4 and three resource types A, B, C with total instances $A = 10$, $B = 5$, $C = 7$. Snapshot:

Process	Allocation A B C	Max A B C	Need = Max – Alloc A B C	Available A B C
P0	0 1 0	7 5 3	7 4 3	3 3 2
P1	2 0 0	3 2 2	1 2 2	

Process	Allocation A B C	Max A B C	Need = Max – Alloc A B C	Available A B C
P2	3 0 2	9 0 2	6 0 0	
P3	2 1 1	2 2 2	0 1 1	
P4	0 0 2	4 3 3	4 3 1	

Solution — Is the system in a safe state?

Work = Available = (3, 3, 2).

P1: Need (1,2,2) ≤ (3,3,2) ✓ → Work = (3,3,2) + (2,0,0) = **(5,3,2)**

P3: Need (0,1,1) ≤ (5,3,2) ✓ → Work = (5,3,2) + (2,1,1) = **(7,4,3)**

P4: Need (4,3,1) ≤ (7,4,3) ✓ → Work = (7,4,3) + (0,0,2) = **(7,4,5)**

P2: Need (6,0,0) ≤ (7,4,5) ✓ → Work = (7,4,5) + (3,0,2) = **(10,4,7)**

P0: Need (7,4,3) ≤ (10,4,7) ✓ → Work = (10,4,7) + (0,1,0) = **(10,5,7)**

Safe sequence: **<P1, P3, P4, P2, P0>** — the system is in a **safe state**.

Example (continued) — P1 requests (1, 0, 2). Can it be granted?

Solution

Request (1,0,2) ≤ Need[1] = (1,2,2) ✓ and ≤ Available (3,3,2) ✓.

Pretend allocation: Available = (2,3,0); Allocation[P1] = (3,0,2); Need[P1] = (0,2,0).

Safety check: P1 (0,2,0) ≤ (2,3,0) → Work = (5,3,2); P3 → (7,4,3); P4 → (7,4,5); P0 → (7,5,5); P2 → (10,5,7). A safe sequence **<P1, P3, P4, P0, P2>** exists.

Therefore the request can be **granted immediately**.

Limitations of the Banker's algorithm: processes rarely know their maximum needs in advance; the number of processes is not fixed; resources may suddenly become unavailable (a printer breaks); and the algorithm is expensive — $O(m \times n^2)$.

4.9 Deadlock Detection

If neither prevention nor avoidance is used, the system may enter a deadlock. Then the system must (1) **detect** the deadlock and (2) **recover** from it.

4.9.1 Resource Allocation Graph (single instance of each resource)

Reduce the RAG to a **wait-for graph** by removing resource nodes and collapsing edges: an edge $P_i \rightarrow P_j$ means P_i is waiting for a resource held by P_j . The system is deadlocked **if and only if the wait-for graph contains a cycle**. A cycle-detection algorithm (depth-first search) is run periodically; it takes $O(n^2)$ operations.

4.9.2 Detection with several instances of each resource type

Uses data structures similar to the Banker's algorithm (Available, Allocation and a Request matrix) but no Max/Need. The algorithm marks processes that can finish given current Available; any process that cannot be marked at the end is **deadlocked**. *When to run the algorithm:* at every resource request that cannot be immediately granted (early but costly), at regular intervals (e.g., every hour), or when CPU utilisation drops below a threshold (say 40%).

4.10 Recovery from Deadlock

(a) Process termination

- **Abort all deadlocked processes** — breaks the deadlock certainly but at great expense (partial results are lost).
- **Abort one process at a time** until the cycle is eliminated — overhead of re-running detection after each abort. Victim chosen by: priority, computation time so far and remaining, resources held, resources needed, whether interactive or batch, number of processes to terminate.

(b) Resource preemption

Successively preempt some resources from processes and give them to others until the deadlock cycle is broken. Three issues: **selecting a victim** (minimise cost), **rollback** (return the process to a safe state and restart it) and **starvation** (ensure the same process is not always picked; include number of rollbacks in the cost factor).

(c) Recovery through rollback (checkpointing)

Processes periodically write their state to a **checkpoint** file. When a deadlock is detected, a process that owns a needed resource is **rolled back** to an earlier checkpoint taken before it acquired that resource, and the resource is reassigned.

Related concepts

Term	Meaning
Livelock	Processes keep changing state in response to each other but make no progress (like two people repeatedly stepping aside in a corridor); the CPU is busy but no useful work is done.
Starvation	A process waits indefinitely because others are always given preference; unlike deadlock, the system as a whole continues to work. Cure: aging, FIFO policy.
Two-phase locking	A database technique: acquire all locks first (growing), then release (shrinking); combined with abort-and-restart to avoid deadlock.
Communication deadlock	Two processes each wait for a message from the other; solved with timeouts and retransmission.

Key Takeaways

- ✓ **Deadlock**: a set of processes each waiting for a resource held by another in the set.
- ✓ Four necessary conditions: **mutual exclusion, hold and wait, no preemption, circular wait**.
- ✓ A resource allocation graph with a cycle indicates deadlock (certain for single-instance resources).
- ✓ Four strategies: **ignore (Ostrich)**, **prevent** (break a condition), **avoid** (Banker's algorithm, safe states) and **detect and recover**.
- ✓ Safe state $\Rightarrow$ no deadlock; unsafe state $\Rightarrow$ deadlock possible. Banker's: $\text{Need} = \text{Max} - \text{Allocation}$; find a safe sequence.
- ✓ Recovery: terminate processes, preempt resources, or roll back to checkpoints.

— End of Unit 4 —