02. Operating Systems - Ultimate Master Textbook & Worked Examples Reference

> **Module Focus**: Comprehensive Theory + Step-by-Step Worked Numerical Examples under Every Pillar across All 5 Core Operating Systems Topics.

🏛️ Pillar 1: Process Management & CPU Scheduling

1.1 Process Architecture & Address Space Layout

A **Process** is a program in execution with an isolated virtual address space provided by the Operating System.

```
VIRTUAL ADDRESS SPACE LAYOUT
+-----------------------------------+ High Memory Address (0x7FFFFFFF...)
| Stack (Local Vars, Function Frames)| v (Grows Downward)
|                                   |
| Heap (Dynamic Memory: malloc/new)  | ^ (Grows Upward)
+-----------------------------------+
| Data Segment (Globals, Statics)    |
+-----------------------------------+
| Text Segment (Machine Instructions)| Low Memory Address (0x00000000...)
+-----------------------------------+
```

Process State Transitions:

```
         +-------------------------------------------------+
         |                                                 |
         v                                                 |
[New] -> [Ready] <--- (Time Quantum Expired / Preempted) --+
           |                                               |
           | (Scheduled by CPU Scheduler)                  |
           v                                               |
       [Running] -----------------------------------------+
           |
           +---> (I/O Request / Wait for Event) ---> [Waiting/Blocked]
           |                                               |
           |                                               |
           v                                               v
      [Terminated] <------- (I/O Completed) --------------+
```

1.2 Performance Metrics & Core Equations

- **Completion Time (CT)**: Timestamp at which a process finishes execution.
- **Turnaround Time (TAT)**: Total time elapsed from arrival to completion. $$TAT = CT - AT$$
- **Waiting Time (WT)**: Total time spent sitting in the Ready Queue. $$WT = TAT - BT$$
- **Response Time (RT)**: Time from arrival until the process is first assigned CPU execution. $$RT = \text{First Run Timestamp} - AT$$
- **CPU Utilization**: $\frac{\text{Total CPU Busy Time}}{\text{Total Elapsed Time}} \times 100\%$
- **Throughput**: $\frac{\text{Total Processes Completed}}{\text{Total Unit Time}}$

1.3 Worked Numerical Examples for ALL 8 CPU Scheduling Algorithms

Master Benchmark Dataset (for Algorithms 1–6):

Process	Arrival Time (AT)	Burst Time (BT)	Priority *(Lower Int = Higher Priority)*
P_1	0 ms	8 ms	3
P_2	1 ms	4 ms	1 (Highest)
P_3	2 ms	9 ms	4
P_4	3 ms	5 ms	2

---

1. FCFS (First-Come, First-Served)

- **Nature**: Non-preemptive. Executes processes strictly in order of arrival.

✏️ **Worked Numerical Step-by-Step Trace:**

- $t=0$: P_1 arrives $\implies$ runs $t=0 \to 8$. ($CT = 8$)
- $t=8$: P_2 runs $t=8 \to 12$. ($CT = 12$)
- $t=12$: P_3 runs $t=12 \to 21$. ($CT = 21$)
- $t=21$: P_4 runs $t=21 \to 26$. ($CT = 26$)

Gantt Chart:

```
| P1 (0 - 8) | P2 (8 - 12) | P3 (12 - 21) | P4 (21 - 26) |
```

📊 **Computation Table:**

Process	AT	BT	CT	$TAT = CT - AT$	$WT = TAT - BT$	RT
P_1	0	8	8	$8 - 0 = 8$	$8 - 8 = 0$	$0 - 0 = 0$
P_2	1	4	12	$12 - 1 = 11$	$11 - 4 = 7$	$8 - 1 = 7$
P_3	2	9	21	$21 - 2 = 19$	$19 - 9 = 10$	$12 - 2 = 10$

Process	AT	BT	CT	$TAT = CT - AT$	$WT = TAT - BT$	RT
P_4	3	5	26	$26 - 3 = 23$	$23 - 5 = 18$	$21 - 3 = 18$

- **Average Turnaround Time**: $\frac{8 + 11 + 19 + 23}{4} = \mathbf{15.25 \text{ ms}}$
- **Average Waiting Time**: $\frac{0 + 7 + 10 + 18}{4} = \mathbf{8.75 \text{ ms}}$

---

2. SJF (Shortest Job First - Non-Preemptive)

- **Nature**: Non-preemptive. Selects process with smallest total burst time among ready processes.

✏️ **Worked Numerical Step-by-Step Trace:**

- $t=0$: Only P_1 is ready. P_1 runs $t=0 \to 8$. ($CT=8$).
- $t=8$: $P_2(BT=4), P_3(BT=9), P_4(BT=5)$ are all in Ready Queue.
- Smallest BT is $P_2(4) \implies$ runs $t=8 \to 12$. ($CT=12$).
- $t=12$: Ready: $P_4(5), P_3(9)$.
- Smallest BT is $P_4(5) \implies$ runs $t=12 \to 17$. ($CT=17$).
- $t=17$: Ready: $P_3(9) \implies$ runs $t=17 \to 26$. ($CT=26$).

Gantt Chart:

```
| P1 (0 - 8) | P2 (8 - 12) | P4 (12 - 17) | P3 (17 - 26) |
```

📊 **Computation Table:**

Process	AT	BT	CT	TAT	WT	RT
P_1	0	8	8	8	0	0
P_2	1	4	12	11	7	7
P_3	2	9	26	24	15	15
P_4	3	5	17	14	9	9

- **Average Turnaround Time**: $\frac{8 + 11 + 24 + 14}{4} = \mathbf{14.25 \text{ ms}}$
- **Average Waiting Time**: $\frac{0 + 7 + 15 + 9}{4} = \mathbf{7.75 \text{ ms}}$

---

3. SRTF (Shortest Remaining Time First - Preemptive SJF)

- **Nature**: Preemptive. Preempts current process if a new process arrives with smaller remaining burst time.

✏️ **Worked Numerical Step-by-Step Trace:**

- $t=0$: Run P_1. Remaining $BT(P_1) = 8$.
- $t=1$: $P_2(BT=4)$ arrives. Remaining $BT(P_1) = 7$. Since $4 < 7$, preempt P_1. Run P_2.

- $t=2$: $P_3(BT=9)$ arrives. $BT(P_2)$ remaining $= 3$. Continue P_2.
- $t=3$: $P_4(BT=5)$ arrives. $BT(P_2)$ remaining $= 2$. Continue P_2.
- $t=5$: P_2 completes ($CT=5$). Ready: $P_4(5), P_1(7), P_3(9)$. Run P_4.
- $t=10$: P_4 completes ($CT=10$). Ready: $P_1(7), P_3(9)$. Run P_1.
- $t=17$: P_1 completes ($CT=17$). Ready: $P_3(9)$. Run P_3.
- $t=26$: P_3 completes ($CT=26$).

Gantt Chart:

```
| P1 (0-1) | P2 (1-5) | P4 (5-10) | P1 (10-17) | P3 (17-26) |
```

📊 **Computation Table:**

Process	AT	BT	CT	TAT	WT	RT
P_1	0	8	17	17	$17 - 8 = 9$	$0 - 0 = 0$
P_2	1	4	5	4	$4 - 4 = 0$	$1 - 1 = 0$
P_3	2	9	26	24	$24 - 9 = 15$	$17 - 2 = 15$
P_4	3	5	10	7	$7 - 5 = 2$	$5 - 3 = 2$

- **Average Turnaround Time**: $\frac{17 + 4 + 24 + 7}{4} = \mathbf{13.0 \text{ ms}}$
- **Average Waiting Time**: $\frac{9 + 0 + 15 + 2}{4} = \mathbf{6.5 \text{ ms}}$

---

4. Non-Preemptive Priority Scheduling

- **Nature**: Non-preemptive. Selects process with highest priority (lowest integer value).

✏️ **Worked Numerical Step-by-Step Trace:**

- $t=0$: Only P_1 is ready ($\text{Pri}=3$). Runs $t=0 \to 8$. ($CT=8$).
- $t=8$: Ready: $P_2(\text{Pri}=1), P_4(\text{Pri}=2), P_3(\text{Pri}=4)$.
- Highest priority is $P_2 \implies$ runs $t=8 \to 12$. ($CT=12$).
- $t=12$: Ready: $P_4(\text{Pri}=2), P_3(\text{Pri}=4) \implies P_4$ runs $t=12 \to 17$. ($CT=17$).
- $t=17$: $P_3(\text{Pri}=4)$ runs $t=17 \to 26$. ($CT=26$).

Gantt Chart:

```
| P1 (0 - 8) | P2 (8 - 12) | P4 (12 - 17) | P3 (17 - 26) |
```

📊 **Computation Table:**

Process	AT	BT	Priority	CT	TAT	WT	RT
P_1	0	8	3	8	8	0	0

Process	AT	BT	Priority	CT	TAT	WT	RT
P_2	1	4	1	12	11	7	7
P_3	2	9	4	26	24	15	15
P_4	3	5	2	17	14	9	9

- **Average Turnaround Time**: $\mathbf{14.25 \text{ ms}}$
- **Average Waiting Time**: $\mathbf{7.75 \text{ ms}}$

---

5. Preemptive Priority Scheduling

- **Nature**: Preemptive. Preempts running process if a process with strictly higher priority arrives.

✏️ **Worked Numerical Step-by-Step Trace:**

- $t=0$: Run $P_1(\text{Pri}=3)$.
- $t=1$: $P_2(\text{Pri}=1)$ arrives. Priority $1 < 3 \implies$ Preempt P_1. Run P_2.
- $t=2$: $P_3(\text{Pri}=4)$ arrives. $P_2(\text{Pri}=1)$ higher priority. Continue P_2.
- $t=3$: $P_4(\text{Pri}=2)$ arrives. $P_2(\text{Pri}=1)$ higher priority. Continue P_2.
- $t=5$: P_2 finishes ($CT=5$). Ready: $P_4(2), P_1(3), P_3(4)$. Highest is $P_4 \implies$ run P_4.
- $t=10$: P_4 finishes ($CT=10$). Ready: $P_1(3), P_3(4) \implies$ run P_1.
- $t=17$: P_1 finishes ($CT=17$). Ready: $P_3(4) \implies$ run P_3.
- $t=26$: P_3 finishes ($CT=26$).

Gantt Chart:

```
| P1 (0-1) | P2 (1-5) | P4 (5-10) | P1 (10-17) | P3 (17-26) |
```

📊 **Computation Table:**

Process	AT	BT	Priority	CT	TAT	WT	RT
P_1	0	8	3	17	17	9	0
P_2	1	4	1	5	4	0	0
P_3	2	9	4	26	24	15	15
P_4	3	5	2	10	7	2	2

- **Average Turnaround Time**: $\mathbf{13.0 \text{ ms}}$
- **Average Waiting Time**: $\mathbf{6.5 \text{ ms}}$

---

6. Round Robin (RR) Scheduling ($Time\ Quantum\ q = 4\text{ ms}$)

- **Nature**: Preemptive circular queue scheduling.

✏️ **Worked Numerical Step-by-Step Trace & Ready Queue Tracking:**

- $t=0$: Ready Queue: $[P_1]$. Run P_1 for $q=4\text{ms}$ ($t=0 \to 4$).
- During $t=1$: P_2 arrives.
- During $t=2$: P_3 arrives.
- During $t=3$: P_4 arrives.
- $t=4$: P_1 quantum expires (rem $BT=4$). Re-queued behind arrived processes.
- Ready Queue: $[P_2, P_3, P_4, P_1]$. Run $P_2(BT=4)$ for $t=4 \to 8$.
- $t=8$: P_2 finishes ($CT=8$). Ready Queue: $[P_3, P_4, P_1]$. Run $P_3(BT=9)$ for $q=4\text{ms}$ ($t=8 \to 12$).
- $t=12$: P_3 quantum expires (rem $BT=5$). Ready Queue: $[P_4, P_1, P_3]$. Run $P_4(BT=5)$ for $q=4\text{ms}$ ($t=12 \to 16$).
- $t=16$: P_4 quantum expires (rem $BT=1$). Ready Queue: $[P_1, P_3, P_4]$. Run $P_1(rem\ BT=4)$ for $t=16 \to 20$.
- $t=20$: P_1 finishes ($CT=20$). Ready Queue: $[P_3, P_4]$. Run $P_3(rem\ BT=5)$ for $q=4\text{ms}$ ($t=20 \to 24$).
- $t=24$: P_3 quantum expires (rem $BT=1$). Ready Queue: $[P_4, P_3]$. Run $P_4(rem\ BT=1)$ for $t=24 \to 25$.
- $t=25$: P_4 finishes ($CT=25$). Ready Queue: $[P_3]$. Run $P_3(rem\ BT=1)$ for $t=25 \to 26$.
- $t=26$: P_3 finishes ($CT=26$).

Gantt Chart:

```
| P1 (0-4) | P2 (4-8) | P3 (8-12) | P4 (12-16) | P1 (16-20) | P3 (20-24) | P4 (24-25) | P3 (25-26) |
```

📊 **Computation Table:**

Process	AT	BT	CT	$TAT = CT - AT$	$WT = TAT - BT$	RT
P_1	0	8	20	$20 - 0 = 20$	$20 - 8 = 12$	$0 - 0 = 0$
P_2	1	4	8	$8 - 1 = 7$	$7 - 4 = 3$	$4 - 1 = 3$
P_3	2	9	26	$26 - 2 = 24$	$24 - 9 = 15$	$8 - 2 = 6$
P_4	3	5	25	$25 - 3 = 22$	$22 - 5 = 17$	$12 - 3 = 9$

- **Average Turnaround Time**: $\frac{20 + 7 + 24 + 22}{4} = \mathbf{18.25 \text{ ms}}$
- **Average Waiting Time**: $\frac{12 + 3 + 15 + 17}{4} = \mathbf{11.75 \text{ ms}}$

---

7. Multilevel Queue Scheduling

- **Nature**: Memory divided into static queues based on process type (System vs User). Fixed priority preemption between queues.

✏️ **Worked Numerical Step-by-Step Trace:**

- **Queue 1 (System Processes - Priority 1, High)**: RR ($q=2\text{ms}$). Processes: $P_{sys1} (BT=3, AT=0)$, $P_{sys2} (BT=2, AT=0)$.
- **Queue 2 (User Processes - Priority 2, Low)**: FCFS. Processes: $P_{user1} (BT=4, AT=0)$.
- Queue 1 runs first (Absolute Priority):
- $t=0 \to 2$: P_{sys1} runs. (Rem $BT = 1$).
- $t=2 \to 4$: P_{sys2} runs. (P_{sys2} completes, $CT=4$).
- $t=4 \to 5$: P_{sys1} runs. (P_{sys1} completes, $CT=5$).
- Queue 1 is now empty. CPU switches to Queue 2:
- $t=5 \to 9$: P_{user1} runs FCFS. (P_{user1} completes, $CT=9$).

Gantt Chart:

```
| Psys1 (0-2) | Psys2 (2-4) | Psys1 (4-5) | Puser1 (5-9) |
```

📊 **Computation Table:**

Process	Queue	AT	BT	CT	TAT	WT
P_{sys1}	System (Q_1)	0	3	5	5	$5 - 3 = 2$
P_{sys2}	System (Q_1)	0	2	4	4	$4 - 2 = 2$
P_{user1}	User (Q_2)	0	4	9	9	$9 - 4 = 5$

---

8. Multilevel Feedback Queue (MLFQ) Scheduling

- **Nature**: Dynamic movement between queues based on CPU burst execution duration (Demotes CPU-bound processes, promotes aging processes).

✏️ **Worked Numerical Step-by-Step Trace:**

- **Queue 0 (Q_0)**: Round Robin ($q = 8\text{ ms}$)
- **Queue 1 (Q_1)**: Round Robin ($q = 16\text{ ms}$)
- **Queue 2 (Q_2)**: FCFS

Process Execution Trace: Process P_1 with $BT = 35\text{ ms}$ arrives at $t=0$: 1. **Enters Q_0**: Runs for 8ms slice ($t=0 \to 8$). Remaining $BT = 27\text{ms}$. - Did not finish within $q=8\text{ms} \implies$ Demoted to Q_1. 2. **Enters Q_1**: Runs for 16ms slice ($t=8 \to 24$). Remaining $BT = 11\text{ms}$. - Did not finish within $q=16\text{ms} \implies$ Demoted to Q_2. 3. **Enters Q_2**: Runs under FCFS for remaining 11ms ($t=24 \to 35$). - Finishes execution at $t=35\text{ms}$! ($CT=35$, $TAT=35$, $WT=0$).

🏛️ Pillar 2: Memory Management & ALL 7 Page Replacement Algorithms Masterclass

2.1 Paging & Address Translation Math

Structure of Virtual and Physical Addresses:

- **Logical Address**: Consists of Page Number (p) and Page Offset (d).
- **Physical Address**: Consists of Frame Number (f) and Frame Offset (d).
- **Key Equations**: $$\text{Page Size} = 2^d \text{ bytes}$$ $$\text{Number of Pages} = 2^p = \frac{\text{Logical Address Space}}{\text{Page Size}}$$ $$\text{Page Table Size} = \text{Number of Pages} \times \text{Page Table Entry Size}$$

✏️ **Worked Numerical Example 1 (Logical Address Translation):**

Given a 32-bit Logical Address Space with a Page Size of $4 \text{ KB} = 4096 \text{ bytes} = 2^{12} \text{ bytes}$. Page Table Entry size $= 4 \text{ bytes}$. Given Logical Address $= \text{0x00003024} = 12324_{10}$. Page Table Mapping: Page $3 \to$ Frame 7.

1. **Calculate Bit Split**:
2. Offset bits $d = 12$ bits (since $2^{12} = 4096$).
3. Page Number bits $p = 32 - 12 = 20$ bits.
4. **Extract Page Number (p) and Offset (d)**:
5. $p = \lfloor 12324 / 4096 \rfloor = \mathbf{3}$
6. $d = 12324 \pmod{4096} = \mathbf{36} = \mathbf{0x024}$
7. **Calculate Physical Address**:
8. Frame Number $f = 7$.
9. $\text{Physical Address} = (f \times \text{Page Size}) + d = (7 \times 4096) + 36 = 28672 + 36 = \mathbf{28708_{10}} = \mathbf{0x00007024}$.

✏️ **Worked Numerical Example 2 (TLB Effective Access Time):**

- **TLB Hit Ratio (h)**: $90\% = 0.90$
- **TLB Access Time (t_{tlb})**: 20 ns
- **Main Memory Access Time (t_{mem})**: 100 ns

$$\text{Effective Access Time (EAT)} = h \cdot (t_{\text{tlb}} + t_{\text{mem}}) + (1 - h) \cdot (t_{\text{tlb}} + 2 t_{\text{mem}})$$ $$\text{EAT} = 0.90 \cdot (20 + 100) + 0.10 \cdot (20 + 200) = 0.90(120) + 0.10(220) = 108 + 22 = \mathbf{130 \text{ ns}}$$

2.2 Complete Breakdown & Trace Benchmark for ALL 7 Page Replacement Algorithms

Master Benchmark Setup: - **Reference String**: `7, 0, 1, 2, 0, 3, 0, 4, 2, 3` - **Frame Capacity**: **3 Frames** (Initially empty: `[-, -, -]`).

---

1. FIFO (First-In, First-Out)

- **Policy**: Replaces the oldest page in memory.
- **Defect**: Suffers from **Belady's Anomaly** (increasing frame allocation can increase page fault rate).

✏️ Step-by-Step Execution Trace:

Step	Ref Page	Frame 0	Frame 1	Frame 2	Status	Evicted Page
1	`7`	**7**	-	-	**FAULT**	None
2	`0`	7	**0**	-	**FAULT**	None
3	`1`	7	0	**1**	**FAULT**	None
4	`2`	**2**	0	1	**FAULT**	`7`
5	`0`	2	0	1	**HIT**	None
6	`3`	2	**3**	1	**FAULT**	`0`
7	`0`	2	3	**0**	**FAULT**	`1`
8	`4`	**4**	3	0	**FAULT**	`2`
9	`2`	4	**2**	0	**FAULT**	`3`
10	`3`	4	2	**3**	**FAULT**	`0`

- **Total FIFO Faults**: $\mathbf{9 \text{ Page Faults}}$

💡 Belady's Anomaly Numerical Proof:

For Reference String `1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5`: - With **3 Frames** $\implies \mathbf{9 \text{ Page Faults}}$. - With **4 Frames** $\implies \mathbf{10 \text{ Page Faults}}$ *(Faults increase despite adding memory!)*.

---

2. LRU (Least Recently Used)

- **Policy**: Replaces the page that has not been used for the longest period in the past.
- **Property**: Stack Algorithm (Immune to Belady's Anomaly).

✏️ Step-by-Step Execution Trace:

Step	Ref Page	Frame 0	Frame 1	Frame 2	Status	Recency Order (Oldest $\to$ Newest)
1	`7`	**7**	-	-	**FAULT**	[7]
2	`0`	7	**0**	-	**FAULT**	[7, 0]

Step	Ref Page	Frame 0	Frame 1	Frame 2	Status	Recency Order (Oldest $\to$ Newest)
3	`1`	7	0	**1**	**FAULT**	[7, 0, 1]
4	`2`	**2**	0	1	**FAULT**	[0, 1, 2] *(Evict 7)*
5	`0`	2	0	1	**HIT**	[1, 2, 0]
6	`3`	2	0	**3**	**FAULT**	[2, 0, 3] *(Evict 1)*
7	`0`	2	0	3	**HIT**	[2, 3, 0]
8	`4`	**4**	0	3	**FAULT**	[3, 0, 4] *(Evict 2)*
9	`2`	4	0	**2**	**FAULT**	[0, 4, 2] *(Evict 3)*
10	`3`	4	**3**	2	**FAULT**	[4, 2, 3] *(Evict 0)*

- **Total LRU Faults**: $\mathbf{8 \text{ Page Faults}}$

---

3. Optimal Page Replacement (OPT / MIN)

- **Policy**: Replaces the page that will not be used for the longest period of time in the **future**.

✏️ **Step-by-Step Execution Trace:**

Step	Ref Page	Frame 0	Frame 1	Frame 2	Status	Future Next Access
1	`7`	**7**	-	-	**FAULT**	Never
2	`0`	7	**0**	-	**FAULT**	Step 5
3	`1`	7	0	**1**	**FAULT**	Never
4	`2`	**2**	0	1	**FAULT**	Evict `7` (Never used again)
5	`0`	2	0	1	**HIT**	Step 7
6	`3`	2	0	**3**	**FAULT**	Evict `1` (Never used again)
7	`0`	2	0	3	**HIT**	None
8	`4`	**4**	0	3	**FAULT**	Evict `2` (Used at Step 9 vs 3 at Step 10)
9	`2`	4	0	**2**	**FAULT**	Evict `3`
10	`3`	4	0	**3**	**FAULT**	Evict `2`

- **Total OPT Faults**: $\mathbf{7 \text{ Page Faults}}$ *(Theoretical Benchmark Minimum)*

---

4. Second Chance (Clock Algorithm)

- **Policy**: Circular queue with a reference bit R.
- Access $\implies R = 1$.
- Replacement $\implies$ Check clock pointer: If $R=1 \implies$ reset $R=0$ and advance pointer. If $R=0 \implies$ replace page.

✏️ **Step-by-Step Execution Trace:**

Step	Ref Page	Frame Array `[F0, F1, F2]`	Ref Bits `[R0, R1, R2]`	Clock Pointer	Status	Evicted Page
1	`7`	`[7, -, -]`	`[1, 0, 0]`	1	**FAULT**	None
2	`0`	`[7, 0, -]`	`[1, 1, 0]`	2	**FAULT**	None
3	`1`	`[7, 0, 1]`	`[1, 1, 1]`	0	**FAULT**	None
4	`2`	`[2, 0, 1]`	`[1, 0, 0]`	1	**FAULT**	`7` *(Clared R0,R1,R2=0)*
5	`0`	`[2, 0, 1]`	`[1, 1, 0]`	1	**HIT**	None
6	`3`	`[2, 0, 3]`	`[1, 1, 1]`	0	**FAULT**	`1`
7	`0`	`[2, 0, 3]`	`[1, 1, 1]`	0	**HIT**	None
8	`4`	`[4, 0, 3]`	`[1, 0, 0]`	1	**FAULT**	`2` *(Cleared R0,R1,R2=0)*
9	`2`	`[4, 2, 3]`	`[1, 1, 0]`	2	**FAULT**	`0`
10	`3`	`[4, 2, 3]`	`[1, 1, 1]`	2	**HIT**	None

- **Total Second Chance Faults**: $\mathbf{8 \text{ Page Faults}}$

---

5. Enhanced Second Chance Algorithm (2-Bit Pair)

- **Policy**: Uses pair (R, M) where $R = \text{Reference Bit}$, $M = \text{Modify/Dirty Bit}$.
- **Class Hierarchy**:
- `(0, 0)` : Best candidate (Not recently used, clean).
- `(0, 1)` : Second choice (Not recently used, but modified - requires writeback).
- `(1, 0)` : Third choice (Recently used, clean).
- `(1, 1)` : Worst candidate (Recently used and modified).

✏️ **Worked Example Trace:**

Reference sequence with operations: `7(R), 0(W), 1(R), 2(W), 0(R), 3(W), 0(R), 4(W), 2(R), 3(R)` on 3 Frames. - Step 1-3: Load `7(1,0)` , `0(1,1)` , `1(1,0)` . Frames full. - Step 4 (`2(W)`): Scan for Class $(0,0) \implies$ none. Scan for Class $(0,1)$ clearing $R=0$. Finds `7(0,0)` $\implies$ **Evict 7**, load `2(1,1)` . - Step

5 (`0(R)`): Hit on `0` . Set $R=1 \implies (1,1)$. - Step 6 (`3(W)`): Scan for $(0,0) \implies$ Finds `1(0,0)` $ \implies$ **Evict 1**, load `3(1,1)` . - Total Faults: **8 Page Faults**.

---

6. LFU (Least Frequently Used)

- **Policy**: Maintains execution access frequency counter per page. Replaces page with smallest counter. Tie-breaker: FIFO.

✏️ **Step-by-Step Execution Trace:**

Step	Ref Page	Frames `[F0, F1, F2]`	Frequencies `(Page: Count)`	Status	Evicted Page
1	`7`	`[7, -, -]`	7:1	**FAULT**	None
2	`0`	`[7, 0, -]`	7:1, 0:1	**FAULT**	None
3	`1`	`[7, 0, 1]`	7:1, 0:1, 1:1	**FAULT**	None
4	`2`	`[2, 0, 1]`	0:1, 1:1, 2:1	**FAULT**	`7` *(FIFO tie-break count 1)*
5	`0`	`[2, 0, 1]`	0:2, 1:1, 2:1	**HIT**	None
6	`3`	`[2, 0, 3]`	0:2, 2:1, 3:1	**FAULT**	`1` *(Tie-break count 1)*
7	`0`	`[2, 0, 3]`	0:3, 2:1, 3:1	**HIT**	None
8	`4`	`[4, 0, 3]`	0:3, 3:1, 4:1	**FAULT**	`2` *(Tie-break count 1)*
9	`2`	`[4, 0, 2]`	0:3, 4:1, 2:1	**FAULT**	`3` *(Tie-break count 1)*
10	`3`	`[3, 0, 2]`	0:3, 2:1, 3:1	**FAULT**	`4` *(Tie-break count 1)*

- **Total LFU Faults**: $\mathbf{9 \text{ Page Faults}}$

---

7. MFU (Most Frequently Used)

- **Policy**: Replaces page with highest access frequency counter (assumes low-count page was just brought in and will be needed). Tie-breaker: FIFO.

✏️ **Step-by-Step Execution Trace:**

- Step 1-3: Load `7` , `0` , `1` (Faults 1, 2, 3). Frequencies: 7:1, 0:1, 1:1.
- Step 4 (`2`): Highest freq tie 1 $\implies$ FIFO evicts `7` . Load `2` . (Fault 4). Freqs: 0:1, 1:1, 2:1.
- Step 5 (`0`): Hit. Frequencies: 0:2, 1:1, 2:1.
- Step 6 (`3`): Highest freq is `0` (count 2) $\implies$ **Evict 0**. Load `3` . (Fault 5). Freqs: 1:1, 2:1, 3:1.
- Step 7 (`0`): Highest freq tie 1 $\implies$ FIFO evicts `1` . Load `0` . (Fault 6). Freqs: 2:1, 3:1, 0:1.
- Step 8 (`4`): Evicts `2` (FIFO tie). Load `4` . (Fault 7).

- Step 9 (`2`): Evicts `3` (FIFO tie). Load `2`. (Fault 8).
- Step 10 (`3`): Evicts `0` (FIFO tie). Load `3`. (Fault 9).
- **Total MFU Faults**: $\mathbf{9 \text{ Page Faults}}$

📊 Master Page Replacement Benchmark Summary Table

Algorithm	Fault Count (`7,0,1,2,0,3,0,4,2,3`)	Hardware Overhead	Belady's Anomaly?	Production OS Implementation
Optimal (OPT)	**7 (Minimum)**	Theoretical (Needs Future)	**Immune**	Theoretical Benchmark
LRU	**8**	High (Counter / Stack per ref)	**Immune**	Approximated via Clock
Second Chance (Clock)	**8**	Low (1 Bit per frame)	**Vulnerable**	**Linux / Windows Page Replacer**
Enhanced Second Chance	**8**	Low (2 Bits per frame)	**Vulnerable**	Windows Memory Manager
FIFO	**9**	Minimal (Queue pointer)	**Vulnerable**	Rarely used standalone
LFU	**9**	High (Counter registers)	**Vulnerable**	Cache Layering
MFU	**9**	High (Counter registers)	**Vulnerable**	Specialized DB engines

---

🏛️ Pillar 3: Deadlocks & Banker's Algorithm Masterclass

3.1 The 4 Coffman Conditions for Deadlock

Deadlock occurs if and only if all 4 conditions hold simultaneously: 1. **Mutual Exclusion**: At least one resource is held in a non-shareable mode. 2. **Hold and Wait**: A process holds at least one resource and waits for additional resources held by others. 3. **No Preemption**: Resources cannot be preempted forcibly; released only voluntarily. 4. **Circular Wait**: Set of waiting processes $\{P_0, P_1, \dots, P_n\}$ such that P_0 waits for P_1, P_1 for P_2, ..., and P_n for P_0.

3.2 Banker's Algorithm Data Structures & Formulas

Let $n = \text{number of processes}$, $m = \text{number of resource types}$. - **$\text{Available}[m]$**: Available instances of each resource type. - **$\text{Max}[n][m]$**: Maximum resource demand of each process. - **$**

\text{Allocation}[n][m]$**: Currently allocated instances to each process. - **$\text{Need}[n][m]$: Remaining resource need of each process. $$\text{Need}[i][j] = \text{Max}[i][j] - \text{Allocation}[i][j]$$

3.3 Worked Numerical Example 1: Dijkstra's Safety Algorithm

Benchmark System State:

5 Processes (P_0, P_1, P_2, P_3, P_4) and 3 Resource Types ($A=10, B=5, C=7$).

Process	Allocation (A, B, C)	Max (A, B, C)	Need Matrix ($Need = Max - Alloc$)
P_0	0, 1, 0	7, 5, 3	$(7-0, 5-1, 3-0) = \mathbf{(7, 4, 3)}$
P_1	2, 0, 0	3, 2, 2	$(3-2, 2-0, 2-0) = \mathbf{(1, 2, 2)}$
P_2	3, 0, 2	9, 0, 2	$(9-3, 0-0, 2-2) = \mathbf{(6, 0, 0)}$
P_3	2, 1, 1	2, 2, 2	$(2-2, 2-1, 2-1) = \mathbf{(0, 1, 1)}$
P_4	0, 0, 2	4, 3, 3	$(4-0, 3-0, 3-2) = \mathbf{(4, 3, 1)}$

Step 1: Compute Initial Available Vector:

$$\text{Total Allocated} = \sum \text{Allocation} = (0+2+3+2+0, 1+0+0+1+0, 0+0+2+1+2) = (7, 2, 5)$$ $$ \text{Available} = \text{Total Resources} - \text{Total Allocated} = (10-7, 5-2, 7-5) = \mathbf{(3, 3, 2)}$$

Step 2: Step-by-Step Safety Trace:

Initialize $\text{Work} = (3, 3, 2)$, $\text{Finish} = [\text{False}, \text{False}, \text{False}, \text{False}, \text{False}]$.

1. **Check P_0**: $\text{Need}_0 (7, 4, 3) \le \text{Work} (3, 3, 2)$ $\implies \text{FALSE}$. Cannot execute yet.
2. **Check P_1**: $\text{Need}_1 (1, 2, 2) \le \text{Work} (3, 3, 2)$ $\implies \mathbf{\text{TRUE}}$.
3. Execute P_1. New $\text{Work} = \text{Work} + \text{Allocation}_1 = (3,3,2) + (2,0,0) = \mathbf{(5, 3, 2)}$.
4. $\text{Finish}[1] = \text{True}$.
5. **Check P_3**: $\text{Need}_3 (0, 1, 1) \le \text{Work} (5, 3, 2)$ $\implies \mathbf{\text{TRUE}}$.
6. Execute P_3. New $\text{Work} = (5,3,2) + (2,1,1) = \mathbf{(7, 4, 3)}$.
7. $\text{Finish}[3] = \text{True}$.
8. **Check P_4**: $\text{Need}_4 (4, 3, 1) \le \text{Work} (7, 4, 3)$ $\implies \mathbf{\text{TRUE}}$.
9. Execute P_4. New $\text{Work} = (7,4,3) + (0,0,2) = \mathbf{(7, 4, 5)}$.
10. $\text{Finish}[4] = \text{True}$.
11. **Check P_0**: $\text{Need}_0 (7, 4, 3) \le \text{Work} (7, 4, 5)$ $\implies \mathbf{\text{TRUE}}$.
12. Execute P_0. New $\text{Work} = (7,4,5) + (0,1,0) = \mathbf{(7, 5, 5)}$.
13. $\text{Finish}[0] = \text{True}$.
14. **Check P_2**: $\text{Need}_2 (6, 0, 0) \le \text{Work} (7, 5, 5)$ $\implies \mathbf{\text{TRUE}}$.
15. Execute P_2. New $\text{Work} = (7,5,5) + (3,0,2) = \mathbf{(10, 5, 7)}$.

16. $\text{Finish}[2] = \text{True}$.

$$\text{System is in a SAFE STATE. Safe Execution Sequence: } \mathbf{\langle P_1, P_3, P_4, P_0, P_2 \rangle}$$

3.4 Worked Numerical Example 2: Resource Request Algorithm

Suppose Process P_1 makes an additional request: $\text{Request}_1 = (1, 0, 2)$. Can this request be granted immediately?

1. **Step 1**: Is $\text{Request}_1 (1,0,2) \le \text{Need}_1 (1,2,2)$? $\implies \mathbf{\text{TRUE}}$.
2. **Step 2**: Is $\text{Request}_1 (1,0,2) \le \text{Available} (3,3,2)$? $\implies \mathbf{\text{TRUE}}$.
3. **Step 3**: Pretend to allocate resources: $$\text{Available} = (3,3,2) - (1,0,2) = \mathbf{(2, 3, 0)}$$ $$\text{Allocation}_1 = (2,0,0) + (1,0,2) = \mathbf{(3, 0, 2)}$$ $$\text{Need}_1 = (1,2,2) - (1,0,2) = \mathbf{(0, 2, 0)}$$
4. **Step 4**: Run Safety Test on Modified State:
5. $\text{Work} = (2, 3, 0)$.
6. P_1: $\text{Need}_1(0,2,0) \le (2,3,0) \implies \text{Run } P_1$. New $\text{Work} = (2,3,0) + (3,0,2) = \mathbf{(5, 3, 2)}$.
7. P_3: $\text{Need}_3(0,1,1) \le (5,3,2) \implies \text{Run } P_3$. New $\text{Work} = (5,3,2) + (2,1,1) = \mathbf{(7, 4, 3)}$.
8. P_4: $\text{Need}_4(4,3,1) \le (7,4,3) \implies \text{Run } P_4$. New $\text{Work} = (7,4,3) + (0,0,2) = \mathbf{(7, 4, 5)}$.
9. P_0: $\text{Need}_0(7,4,3) \le (7,4,5) \implies \text{Run } P_0$. New $\text{Work} = (7,4,5) + (0,1,0) = \mathbf{(7, 5, 5)}$.
10. P_2: $\text{Need}_2(6,0,0) \le (7,5,5) \implies \text{Run } P_2$. New $\text{Work} = (7,5,5) + (3,0,2) = \mathbf{(10, 5, 7)}$.

$$\mathbf{Conclusion: \text{Request can be GRANTED immediately since system remains safe.}}$$

🏛️ Pillar 4: Process Synchronization, Hardware Primitives, Semaphores & Classical C Code Masterclass

4.1 The Critical Section Problem Criteria

Any concurrent synchronization mechanism must satisfy: 1. **Mutual Exclusion**: If process P_i is in its critical section, no other process can execute in its critical section. 2. **Progress**: If no process is in its critical section and some wish to enter, only those processes not executing in their remainder section can participate in deciding who enters next, and this decision cannot be postponed indefinitely. 3. **Bounded Waiting**: A bound must exist on the number of times other processes are allowed to enter their critical section after a process has requested entry before that request is granted.

4.2 Software Solution: Peterson's Algorithm (2 Processes P_0, P_1)

```
#include <stdbool.h>

// Shared Atomic State
bool flag[2] = {false, false}; // flag[i] = true implies P_i wants to enter CS
int turn = 0;                  // Whose turn it is to enter CS

void process_P0() {
    while (1) {
        flag[0] = true;              // Express intent to enter
        turn = 1;                    // Yield turn to P1
        while (flag[1] && turn == 1); // Busy Wait (Spinlock)

        // === CRITICAL SECTION ===
        // Access Shared Data safely
        // ========================

        flag[0] = false;             // Exit Critical Section

        // Remainder Section
    }
}

void process_P1() {
    while (1) {
        flag[1] = true;              // Express intent to enter
        turn = 0;                    // Yield turn to P0
        while (flag[0] && turn == 0); // Busy Wait (Spinlock)

        // === CRITICAL SECTION ===
        // Access Shared Data safely
        // ========================

        flag[1] = false;             // Exit Critical Section

        // Remainder Section
    }
}
```

4.3 Hardware Atomic Synchronization Primitives

1. `TestAndSet` Atomic Instruction

`TestAndSet` atomically reads the target memory location and sets it to `true`.

```
#include <stdbool.h>
```

```
// Atomic Hardware Instruction Emulation
bool TestAndSet(bool *target) {
    bool rv = *target;
    *target = true;
    return rv; // Returns original value
}

// Lock Implementation using TestAndSet
bool lock = false;

void acquire_lock() {
    while (TestAndSet(&lock)); // Spin until lock becomes false
}

void release_lock() {
    lock = false;
}
```

2. `CompareAndSwap` (CAS) Atomic Instruction

```
int CompareAndSwap(int *value, int expected, int new_value) {
    int temp = *value;
    if (*value == expected) {
        *value = new_value;
    }
    return temp; // Returns original value
}

// Lock Implementation using CAS
int lock_val = 0; // 0 = unlocked, 1 = locked

void lock_cas() {
    while (CompareAndSwap(&lock_val, 0, 1) != 0); // Spin until 0 returned
}

void unlock_cas() {
    lock_val = 0;
}
```

4.4 Semaphores Definition & C Pseudocode

A **Semaphore** S is an integer variable accessed only through two standard atomic operations: `wait()` (or `P()`) and `signal()` (or `V()`).

```
typedef struct {
    int value;
    struct process *queue; // Sleeping process queue
```

```
} semaphore;

void wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        // Add process to S->queue;
        // sleep(); // Block calling thread
    }
}

void signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        // Remove process P from S->queue;
        // wakeup(P); // Unblock thread
    }
}
```

4.5 ALL 4 Classical Synchronization Problems & Production C Code

1. Bounded Buffer (Producer-Consumer) Problem

- **Semaphores**: `mutex = 1`, `empty = N`, `full = 0`.

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int in = 0, out = 0;

sem_t empty;
sem_t full;
pthread_mutex_t mutex;

void* producer(void* arg) {
    int item = 1;
    while (1) {
        sem_wait(&empty);           // Wait for empty slot
        pthread_mutex_lock(&mutex); // Lock buffer

        buffer[in] = item;
        printf("Producer produced: %d at index %d\n", item, in);
        in = (in + 1) % BUFFER_SIZE;

        pthread_mutex_unlock(&mutex); // Unlock buffer
```

```
        sem_post(&full);              // Increment full count
        item++;
    }
}

void* consumer(void* arg) {
    while (1) {
        sem_wait(&full);            // Wait for full slot
        pthread_mutex_lock(&mutex); // Lock buffer

        int item = buffer[out];
        printf("Consumer consumed: %d from index %d\n", item, out);
        out = (out + 1) % BUFFER_SIZE;

        pthread_mutex_unlock(&mutex); // Unlock buffer
        sem_post(&empty);             // Increment empty count
    }
}
```

---

2. Readers-Writers Problem (First Readers-Writers: Reader Preference)

- **Semaphores**: `rw_mutex = 1`, `mutex = 1`. Shared variable: `read_count = 0`.

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

int read_count = 0;
pthread_mutex_t mutex; // Protects read_count
sem_t rw_mutex;        // Controls writer access

void* reader(void* arg) {
    while (1) {
        pthread_mutex_lock(&mutex);
        read_count++;
        if (read_count == 1) {
            sem_wait(&rw_mutex); // First reader locks writers out
        }
        pthread_mutex_unlock(&mutex);

        // === READING SECTION ===
        printf("Reader is reading shared data...\n");

        pthread_mutex_lock(&mutex);
        read_count--;
        if (read_count == 0) {
            sem_post(&rw_mutex); // Last reader allows writers in
        }
```

```
        pthread_mutex_unlock(&mutex);
    }
}

void* writer(void* arg) {
    while (1) {
        sem_wait(&rw_mutex); // Lock out readers and other writers

        // === WRITING SECTION ===
        printf("Writer is modifying shared data...\n");

        sem_post(&rw_mutex); // Release lock
    }
}
```

---

3. Dining Philosophers Problem

- **Setup**: 5 Philosophers sitting at a round table with 5 chopsticks (`sem_t chopstick[5]`).
- **Deadlock Avoidance**: Odd philosophers pick up left chopstick first; even philosophers pick up right chopstick first.

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <unistd.h>

sem_t chopstick[5];

void* philosopher(void* num) {
    int id = *(int*)num;
    int left = id;
    int right = (id + 1) % 5;

    while (1) {
        printf("Philosopher %d is thinking...\n", id);

        if (id % 2 == 0) {
            sem_wait(&chopstick[right]); // Pick up Right first
            sem_wait(&chopstick[left]);  // Pick up Left second
        } else {
            sem_wait(&chopstick[left]);  // Pick up Left first
            sem_wait(&chopstick[right]); // Pick up Right second
        }

        // === EATING SECTION ===
        printf("Philosopher %d is EATING using chopsticks %d and %d\n", id, left, right);

        sem_post(&chopstick[left]);
```

```
        sem_post(&chopstick[right]);
    }
}
```

---

4. Sleeping Barber Problem

- **Setup**: Barber shop with 1 Barber chair and N waiting chairs.
- **Semaphores**: `customers = 0`, `barbers = 0`, `mutex = 1`. Variable: `waiting = 0`.

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

#define CHAIRS 4

sem_t customers; // Number of waiting customers
sem_t barbers;   // Number of idle barbers
pthread_mutex_t mutex;
int waiting = 0;

void* barber(void* arg) {
    while (1) {
        sem_wait(&customers);       // Sleep if no customers waiting
        pthread_mutex_lock(&mutex); // Lock waiting chair count
        waiting--;                  // Decr waiting count
        sem_post(&barbers);         // Barber ready to cut hair
        pthread_mutex_unlock(&mutex);

        // === CUTTING HAIR ===
        printf("Barber is cutting hair...\n");
    }
}

void* customer(void* arg) {
    pthread_mutex_lock(&mutex);
    if (waiting < CHAIRS) {
        waiting++;
        sem_post(&customers);       // Wake up barber if sleeping
        pthread_mutex_unlock(&mutex);

        sem_wait(&barbers);         // Wait for barber to become ready
        printf("Customer is getting haircut...\n");
    } else {
        // No empty chairs available
        pthread_mutex_unlock(&mutex);
        printf("Customer left: No empty chairs available.\n");
    }
}
```

🏛️ Pillar 5: Complete Disk Scheduling Master Benchmark (All 6 Algorithms)

Benchmark Test Setup:

- **Disk Tracks**: $0 - 199$ (Total 200 tracks).
- **Initial Head Position**: **53**
- **Head Movement Direction**: Moving towards **Higher Tracks (Right)**.
- **Request Queue**: `98, 183, 37, 122, 14, 124, 65, 67`

---

1. FCFS (First-Come, First-Served)

- **Path**: $53 \to 98 \to 183 \to 37 \to 122 \to 14 \to 124 \to 65 \to 67$
- **Step-by-Step Distance Calculation**: $$|98 - 53| + |183 - 98| + |37 - 183| + |122 - 37| + |14 - 122| + |124 - 14| + |65 - 124| + |67 - 65|$$ $$= 45 + 85 + 146 + 85 + 108 + 110 + 59 + 2 = \mathbf{640 \text{ tracks}}$$

---

2. SSTF (Shortest Seek Time First)

- **Path**: $53 \to 65 \to 67 \to 37 \to 14 \to 98 \to 122 \to 124 \to 183$
- **Step-by-Step Distance Calculation**: $$|65 - 53| + |67 - 65| + |37 - 67| + |14 - 37| + |98 - 14| + |122 - 98| + |124 - 122| + |183 - 124|$$ $$= 12 + 2 + 30 + 23 + 84 + 24 + 2 + 59 = \mathbf{236 \text{ tracks}}$$

---

3. SCAN (Elevator Algorithm)

- **Behavior**: Moves towards track 199 servicing requests, reaches end of disk boundary (199), then reverses direction.
- **Path**: $53 \to 65 \to 67 \to 98 \to 122 \to 124 \to 183 \to \mathbf{199 \text{ (disk boundary)}} \to 37 \to 14$
- **Step-by-Step Distance Calculation**: $$(199 - 53) + (199 - 14) = 146 + 185 = \mathbf{331 \text{ tracks}}$$

---

4. C-SCAN (Circular SCAN)

- **Behavior**: Moves towards track 199 servicing requests, reaches track 199, jumps instantly to track 0 without servicing, then moves right servicing remaining requests.
- **Path**: $53 \to 65 \to 67 \to 98 \to 122 \to 124 \to 183 \to \mathbf{199} \to \mathbf{0 \text{ (circular jump)}} \to 14 \to 37$
- **Step-by-Step Distance Calculation**: $$(199 - 53) + (199 - 0) + (37 - 0) = 146 + 199 + 37 = \mathbf{382 \text{ tracks}}$$

---

5. LOOK

- **Behavior**: Moves towards higher requests up to the maximum request (**183**), then reverses direction without hitting the disk boundary (199).
- **Path**: $53 \to 65 \to 67 \to 98 \to 122 \to 124 \to 183 \to 37 \to 14$
- **Step-by-Step Distance Calculation**: $$(183 - 53) + (183 - 14) = 130 + 169 = \mathbf{299 \text{ tracks}}$$

---

6. C-LOOK (Circular LOOK)

- **Behavior**: Moves towards higher requests up to maximum request (**183**), jumps directly to minimum request (**14**), then moves right.
- **Path**: $53 \to 65 \to 67 \to 98 \to 122 \to 124 \to 183 \to \mathbf{14 \text{ (circular jump)}} \to 37$
- **Step-by-Step Distance Calculation**: $$(183 - 53) + (183 - 14) + (37 - 14) = 130 + 169 + 23 = \mathbf{322 \text{ tracks}}$$

📊 Master Disk Scheduling Benchmark Summary Table

Algorithm	Total Head Movement (Tracks)	Reaches Disk Boundary (199)?	Starvation Risk	Primary Advantage
SSTF	**236 (Minimum)**	No	High (Wild requests)	Lowest average seek time
LOOK	**299**	No	Low	High efficiency without boundary overhead
C-LOOK	**322**	No	None	Uniform response time across all tracks
SCAN	**331**	Yes (199)	Low	Elevator algorithm, predictable bound
C-SCAN	**382**	Yes (199 & 0)	None	Eliminates edge request bias
FCFS	**640 (Worst)**	No	None	Completely fair, zero scheduling overhead

---

🏛️ Pillar 6: 🎯 CoCubes Technical MCQ Master Patterns & High-Yield OS Insights

6.1 CoCubes CPU Scheduling & Gantt Chart Question Patterns

CoCubes technical modules heavily feature conceptual and quick-calculation MCQs on CPU scheduling:

1. Time Quantum (TQ) Edge Cases in Round Robin:

- **As $TQ \to \infty$**: Round Robin degrades into **FCFS (First-Come, First-Served)**.
- **As $TQ \to 0$**: Round Robin approaches **Processor Sharing** (pure time sharing), but context-switch overhead approaches 100%, drastically lowering CPU throughput.
- **Optimal TQ Rule of Thumb**: Set TQ such that 80% of CPU bursts finish within a single time quantum.

2. Convoy Effect:

- Occurs in **FCFS** when a long CPU-bound process holds the processor, forcing multiple short I/O-bound processes to wait in the Ready Queue.

3. Shortest Remaining Time First (SRTF) Preemption Triggers:

- A running process P_{run} is preempted **only** when a newly arrived process P_{new} has a burst time $BT(P_{\text{new}}) < \text{Remaining } BT(P_{\text{run}})$.
- If $BT(P_{\text{new}}) = \text{Remaining } BT(P_{\text{run}})$, preemption does **not** occur (tie goes to the running process).

6.2 CoCubes Page Replacement & Memory Management Patterns

1. Belady's Anomaly Identification:

- **Definition**: Page fault rate increases despite increasing the number of physical frame allocations.
- **Vulnerable Algorithms**: **FIFO**, **Second Chance**, **Simple Counting Algorithms**.
- **Immune Algorithms (Stack Algorithms)**: **LRU**, **Optimal (OPT)**, **LFU/MFU with stack properties**.
- **Classic CoCubes MCQ Distractor**: *"Which of the following page replacement algorithms CANNOT exhibit Belady's Anomaly?"* $\implies$ **LRU / Optimal**.

2. Thrashing & Working Set Model:

- **Thrashing** occurs when the total size of processes' working sets exceeds total physical memory capacity ($\sum WSS_i > D$).
- **Symptom**: CPU utilization drops to near zero because processes are constantly handling page faults.
- **Solution**: Decrease the degree of multiprogramming (suspend low-priority processes).

6.3 CoCubes Semaphores & Process Synchronization Patterns

1. Semaphore Operations & Atomic Guarantees:

- **`wait(S)` / `P(S)` Operation**: `c S.value--; if (S.value < 0) { // add this process to S.queue; // block(); }`
- **`signal(S)` / `V(S)` Operation**: `c S.value++; if (S.value <= 0) { // remove a process P from S.queue; // wakeup(P); }`
- **Counting Semaphore Value Interpretation**:

- If $S > 0$: S represents the number of available resource instances.
- If $S < 0$: $|S|$ represents the exact number of processes currently blocked in the semaphore waiting queue.

2. Mutex vs Counting Semaphore:

- **Mutex (Binary Semaphore)**: Initialized to `1`. Used strictly for **Mutual Exclusion** (ownership enforced: only the thread that locked the mutex can unlock it).
- **Counting Semaphore**: Initialized to `N`. Used for **Resource Counting** (any thread can execute `signal()`).