|  |
| --- |
| 🔹 FCFS (First Come First Serve) |
| Sort processes by arrival time  time = 0  For each process:      waiting\_time = time - arrival\_time      turnaround\_time = waiting\_time + burst\_time      time += burst\_time |

|  |
| --- |
| 🔹 SJF (Non-Preemptive) |
| Initialize time = 0  While not all processes are completed:      Select process with shortest burst\_time among arrived and not completed      waiting\_time = time - arrival\_time      turnaround\_time = waiting\_time + burst\_time      time += burst\_time      Mark process as completed |

|  |
| --- |
| 🔹 RR (Round Robin) |
| Initialize queue with arrived processes  While queue is not empty:  Dequeue process  Execute for min(time\_quantum, remaining\_time)  time += executed\_time  If process not finished:  Enqueue again  Else:  turnaround\_time = time - arrival\_time  waiting\_time = turnaround\_time - burst\_time |

|  |
| --- |
| 🔹 Priority (Non-Preemptive) |
| Initialize time = 0  While not all processes are completed:      Select process with highest priority among arrived and not completed      waiting\_time = time - arrival\_time      turnaround\_time = waiting\_time + burst\_time      time += burst\_time      Mark process as completed |

|  |
| --- |
| 🔹 Priority Preemptive |
| Initialize time = 0  While not all processes are completed:      Select process with highest priority among arrived and not finished      Execute for 1 unit      Update remaining\_time and time      If process finishes:          waiting\_time = time - arrival\_time - burst\_time          turnaround\_time = waiting\_time + burst\_time |

|  |
| --- |
| 🔹 SRTN (Shortest Remaining Time Next) |
| Initialize time = 0  While not all processes are completed:      Select process with shortest remaining\_time among arrived      Execute for 1 unit      Update remaining\_time and time      If process finishes:          waiting\_time = time - arrival\_time - burst\_time          turnaround\_time = waiting\_time + burst\_time |

|  |
| --- |
| 🔐 Deadlock Detection 🔹 Cycle Detection (RAG) |
| For each node:      If node not visited:          Call DFS          If cycle found:              Deadlock detected  If no cycle:      No deadlock |

|  |
| --- |
| 🔐 Deadlock Detection 🔹 Banker's Algorithm |
| Work = Available  Finish[i] = false for all i  While exists i such that Finish[i] == false and Need[i] <= Work:      Work += Allocation[i]      Finish[i] = true  If all Finish[i] == true:      Safe state  Else:      Unsafe state |

|  |
| --- |
| 🔹 Producer-Consumer |
| Repeat:      Wait(empty)      Wait(mutex)      Add item to buffer      Signal(mutex)      Signal(full) |

|  |
| --- |
| 🔹 Reader-Writer (Reader Priority) |
| Reader:      Wait(mutex)      reader\_count++      If reader\_count == 1: Wait(write)      Signal(mutex)      Read      Wait(mutex)      reader\_count--      If reader\_count == 0: Signal(write)      Signal(mutex)  Writer:      Wait(write)      Write      Signal(write) |

|  |
| --- |
| **🧠 Memory Allocation** |

|  |
| --- |
| 🔹 First Fit |
| For each request:      Find first hole where size >= request      Allocate and reduce hole size |

|  |
| --- |
| 🔹 Best Fit |
| For each request:      Find hole with minimum leftover space after allocation      Allocate and reduce hole size |

|  |
| --- |
| 🔹 Worst Fit |
| For each request:      Find hole with maximum size      Allocate and reduce hole size |

|  |
| --- |
| 📄 Page Replacement 🔹 FIFO |
| If page not in memory:      If memory full:          Remove oldest page      Add new page |