|  |
| --- |
| **🔹 FCFS (First Come First Serve)** |
| Sort all processes by arrival time  time = 0  for each process in order:  if time < arrival\_time:  time = arrival\_time  waiting\_time = time - arrival\_time  turnaround\_time = waiting\_time + burst\_time  time += burst\_time |

|  |
| --- |
| **🔹 SJF (Non-Preemptive)** |
| time = 0  completed = 0  while completed < n:  Find process with smallest burst\_time among arrived and not completed  if no such process:  time++  continue  waiting\_time = time - arrival\_time  turnaround\_time = waiting\_time + burst\_time  time += burst\_time  completed++ |

|  |
| --- |
| **🔹 RR (Round Robin)** |
| time = 0  queue = empty  Add all processes to queue in arrival order  while queue not empty:  current = dequeue  exec\_time = min(quantum, remaining\_time[current])  time += exec\_time  remaining\_time[current] -= exec\_time  if remaining\_time[current] > 0:  enqueue(current)  else:  turnaround\_time[current] = time - arrival\_time[current]  waiting\_time[current] = turnaround\_time[current] - burst\_time[current] |

|  |
| --- |
| **🔹 Priority (Non-Preemptive)** |
| time = 0  completed = 0  while completed < n:  Find process with highest priority (lowest number) among arrived and not completed  if no such process:  time++  continue  waiting\_time = time - arrival\_time  turnaround\_time = waiting\_time + burst\_time  time += burst\_time  completed++ |

|  |
| --- |
| **🔹 Priority Preemptive** |
| time = 0  while there are unfinished processes:  Find process with highest priority among arrived and not finished  Execute for 1 unit  remaining\_time--  time++  if remaining\_time == 0:  turnaround\_time = time - arrival\_time  waiting\_time = turnaround\_time - burst\_time |

|  |
| --- |
| **🔹 SRTN (Shortest Remaining Time Next)** |
| time = 0  while there are unfinished processes:  Find process with smallest remaining\_time among arrived  Execute for 1 unit  remaining\_time--  time++  if remaining\_time == 0:  turnaround\_time = time - arrival\_time  waiting\_time = turnaround\_time - burst\_time |

|  |
| --- |
| **🔐 Deadlock** |

|  |
| --- |
| **🔐 Deadlock Detection 🔹 Cycle Detection (RAG)** |
| for each node in graph:  visited = false  if not visited:  if DFS(node, visited, stack) returns true:  deadlock detected  DFS(node, visited, stack):  visited[node] = true  stack.push(node)  for each neighbor of node:  if not visited:  if DFS(neighbor, visited, stack) returns true:  return true  else if neighbor in stack:  return true  stack.pop()  return false |

|  |
| --- |
| **🔐 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 = Work + Allocation[i]  Finish[i] = true  if all Finish[i] == true:  system is safe  else:  system is unsafe |

|  |
| --- |
| **🔹 Producer-Consumer** |
| Producer:  while true:  produce item  wait(empty)  wait(mutex)  add to buffer  signal(mutex)  signal(full)  Consumer:  while true:  wait(full)  wait(mutex)  remove from buffer  signal(mutex)  signal(empty)  consume item |

|  |
| --- |
| **🔹 Reader-Writer (Reader Priority)** |
| Reader:  wait(mutex)  read\_count++  if read\_count == 1:  wait(write)  signal(mutex)  read data  wait(mutex)  read\_count--  if read\_count == 0:  signal(write)  signal(mutex)  Writer:  wait(write)  write data  signal(write) |

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

|  |
| --- |
| **🔹 First Fit** |
| for each memory request:  for each hole in list:  if hole.size >= request.size:  allocate from hole  reduce hole size  break |

|  |
| --- |
| **🔹 Best Fit** |
| for each memory request:  best = NULL  for each hole in list:  if hole.size >= request.size:  if best == NULL or hole.size < best.size:  best = hole  if best != NULL:  allocate from best  reduce best size |

|  |
| --- |
| **🔹 Worst Fit** |
| for each memory request:  worst = NULL  for each hole in list:  if hole.size >= request.size:  if worst == NULL or hole.size > worst.size:  worst = hole  if worst != NULL:  allocate from worst  reduce worst size |

|  |
| --- |
| **📄 Page Replacement** |

|  |
| --- |
| **📄 Page Replacement 🔹 FIFO** |
| Initialize queue for pages  for each page reference:  if page not in memory:  if memory is full:  remove oldest page  add new page to end of queue  page\_fault++ |

|  |
| --- |
| **📄 Page Replacement 🔹 Optimal** |
| for each page reference:  if page not in memory:  if memory is full:  find page that will not be used for longest time  replace that page  add new page  page\_fault++ |

|  |
| --- |
| **📄 Page Replacement 🔹 LRU** |
| for each page reference:  if page not in memory:  if memory is full:  remove least recently used page  add new page  else:  update page to most recently used  page\_fault++ (if not in memory) |