◇ 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)
```