First Come First Serve (FCFS)

1. Initialize `time = 0`.
2. Sort the processes by arrival time.
3. For each process:
   - If the process arrives before or at the current time, set its start time to the current time.
   - Calculate the completion time as `start time + burst time`.
   - Calculate the turnaround time as `completion time - arrival time`.
   - Calculate the waiting time as `turnaround time - burst time`.
   - Update `time` to the process's completion time.

Shortest Job First (SJF) Non-Preemptive

1. Initialize `time = 0`, `completed = 0`.
2. While there are uncompleted processes:
   - Find the process with the shortest burst time that has arrived and is not yet completed.
   - Set the start time of this process to the current time.
   - Calculate the completion time as `start time + burst time`.
   - Calculate the turnaround time as `completion time - arrival time`.
   - Calculate the waiting time as `turnaround time - burst time`.
   - Update `time` to the process's completion time.
   - Increment `completed`.

Shortest Job First (SJF) Preemptive

1. Initialize `time = 0`, `completed = 0`.
2. While there are uncompleted processes:
   - Find the process with the shortest remaining burst time that has arrived.
   - Execute this process for one unit of time.
   - Update its remaining burst time.
   - If the process completes, calculate its completion, turnaround, and waiting times.
   - Increment `completed` if the process finishes.
   - Increment `time`.

Priority Scheduling Non-Preemptive

1. Initialize `time = 0`, `completed = 0`.
2. While there are uncompleted processes:
   - Find the highest priority process that has arrived and is not yet completed.
   - Set the start time of this process to the current time.
   - Calculate the completion time as `start time + burst time`.
   - Calculate the turnaround time as `completion time - arrival time`.
   - Calculate the waiting time as `turnaround time - burst time`.
   - Update `time` to the process's completion time.
   - Increment `completed`.

Priority Scheduling Preemptive

1. Initialize `time = 0`, `completed = 0`.
2. While there are uncompleted processes:

- Find the highest priority process that has arrived.
- Execute this process for one unit of time.
- Update its remaining burst time.
- If the process completes, calculate its completion, turnaround, and waiting times.
- Increment `completed` if the process finishes.
- Increment `time`.

Round Robin

1. Initialize `time = 0`, `index = 0`.
2. While there are uncompleted processes:
   - Select the next process in the ready queue.
   - Execute this process for the time quantum or until it finishes.
   - Update its remaining burst time.
   - If the process completes, calculate its completion, turnaround, and waiting times.
   - Add processes that have arrived during the execution to the ready queue.
   - Increment `index`.
   - Update `time`.

Multi-Level Queue Scheduling with FCFS

1. Sort processes into system and user queues based on their type.
2. Apply FCFS scheduling to each queue.
3. System queue runs until empty, then user queue runs.

Rate-Monotonic

1. Initialize `time = 0`.
2. Sort tasks by their periods (shortest period first).
3. For each time unit until the end of the hyper-period:
   - Execute the highest priority task that is ready and has remaining execution time.
   - Update the remaining execution time of the task.
   - If a task completes, reset its remaining time for the next period.

Earliest Deadline First

1. Initialize `time = 0`.
2. For each time unit until the end of the hyper-period:
   - Update deadlines and remaining execution times of tasks.
   - Execute the task with the earliest deadline that is ready and has remaining execution time.
   - Update the remaining execution time of the task.
   - If a task completes, reset its remaining time for the next period.

Proportional Scheduling

1. Initialize `time = 0`, `totalTickets = 0`.
2. Assign tickets to each process.
3. Calculate the total number of tickets.
4. While there are uncompleted processes:
   - Generate a random number.

  - Select the process corresponding to the random ticket.
  - Execute the selected process.
  - Update the remaining burst time of the process.
  - If the process completes, calculate its completion, turnaround, and waiting times.

Producer-Consumer Problem

1. Initialize `mutex = 1`, `full = 0`, `empty = BUFFER_SIZE`.
2. While true:
   - Producer:
     - Wait if `mutex == 0` or `empty == 0`.
     - Produce an item.
     - Decrement `empty`.
     - Increment `full`.
     - Signal `mutex`.
   - Consumer:
     - Wait if `mutex == 0` or `full == 0`.
     - Consume an item.
     - Increment `empty`.
     - Decrement `full`.
     - Signal `mutex`.

Dining Philosophers Problem

1. Initialize an array to track the state of each philosopher.
2. While true:
   - A philosopher tries to pick up both forks.
   - If both forks are available, the philosopher eats.
   - After eating, the philosopher puts down both forks.
   - Repeat.

Banker's Algorithm for Deadlock Avoidance

1. Initialize available, maximum, allocation, and need matrices.
2. While true:
   - Check if the system is in a safe state by finding a sequence where each process can finish.
   - If a request is made, check if it can be granted without leaving the system in an unsafe state.
   - If the request can be granted, update the allocation and need matrices.

Deadlock Detection

1. Initialize available, allocation, and request matrices.
2. While true:
   - Mark processes with no requests as finished.
   - Find a process that can be satisfied with the available resources.
   - If found, simulate allocation of resources and mark the process as finished.
   - If no such process is found, the system is in deadlock.

Worst Fit Memory Allocation

1. Initialize memory blocks and process requirements.
2. For each process:
   - Find the largest memory block that can fit the process.
   - Allocate the block to the process.
   - Update the remaining size of the memory block.

Best Fit Memory Allocation

1. Initialize memory blocks and process requirements.
2. For each process:
   - Find the smallest memory block that can fit the process.
   - Allocate the block to the process.
   - Update the remaining size of the memory block.

First Fit Memory Allocation

1. Initialize memory blocks and process requirements.
2. For each process:
   - Find the first memory block that can fit the process.
   - Allocate the block to the process.
   - Update the remaining size of the memory block.

FIFO Page Replacement

1. Initialize an empty queue.
2. For each page reference:
   - If the page is not in the queue:
     - If the queue is full, remove the oldest page.
     - Add the new page to the queue.
   - If the page is in the queue, do nothing.

LRU Page Replacement

1. Initialize an empty list.
2. For each page reference:
   - If the page is not in the list:
     - If the list is full, remove the least recently used page.
     - Add the new page to the front of the list.
   - If the page is in the list, move it to the front.

Optimal Page Replacement

1. Initialize an empty list.
2. For each page reference:
   - If the page is not in the list:
     - If the list is full, remove the page that will not be used for the longest time.
     - Add the new page to the list.
   - If the page is in the list, do nothing.