**Bubble Sort Algorithm**

1. Define a function bubble\_sort(array, n):

- Input: array[] (array to be sorted), n (size of the array)

2. For i = 0 to n-1:

- Outer loop runs n-1 times

3. For j = 0 to n-i-2:

- Inner loop compares adjacent elements

- If array[j] > array[j+1]:

- Swap array[j] and array[j+1]

4. Print the sorted array.

5. In main:

- Declare an array with values.

- Call bubble\_sort(array, n) with the array and its size.

**Algorithm: Check if a String is a Palindrome**

1. Input: A string `str`.

2. Initialize:

- `len` = length of `str`.

3. Loop:

- For `i` from 0 to `len/2 - 1`:

- Compare `str[i]` with `str[len - i - 1]`.

- If they are not equal:

- Print "Not a palindrome."

- Exit the loop.

4. If the loop completes without finding mismatched characters:

- Print "Palindrome."

5. End.

**Grade Assignment Algorithm - Pseudo Code in C (Shell)**

1. Define a function `assign\_grade(name, score)`:

- Input: name (string), score (integer)

2. If score < 40:

- Assign grade as "Fail"

3. Else if score >= 90 and score <= 100:

- Assign grade as "A"

4. Else if score >= 80 and score < 90:

- Assign grade as "B"

5. Else if score >= 70 and score < 80:

- Assign grade as "C"

6. Else if score >= 60 and score < 70:

- Assign grade as "D"

7. Else:

- Assign grade as "E"

8. Print name and grade.

9. In main:

- Read input (name and score) from file or console.

- Call `assign\_grade(name, score)` for each input.

**Reader-Writer Problem with Semaphores (Pseudo Code in C)**

1. Initialize:

- read\_count = 0 (Number of readers currently accessing the resource).

- mutex: Semaphore for protecting read\_count (initialized to 1).

- write\_lock: Semaphore to control exclusive access for writers (initialized to 1).

- database: Structure holding the clock (hours, minutes, seconds) initialized to {23, 59, 55}.

- iterations = 5 (Number of times each thread will read/write).

2. Reader Function:

- For each iteration:

- Entry Section:

- Acquire mutex to access read\_count.

- Increment read\_count.

- If read\_count == 1, acquire write\_lock (lock writers out).

- Release mutex.

- Critical Section (Reading):

- Print the current time (hours:minutes:seconds).

- Simulate reading with a random sleep.

- Exit Section:

- Acquire mutex to access read\_count.

- Decrement read\_count.

- If read\_count == 0, release write\_lock (allow writers).

- Release mutex.

- Simulate time between read operations with a random sleep.

3. Writer Function:

- For each iteration:

- Entry Section:

- Acquire write\_lock to gain exclusive access to the database.

- Critical Section (Writing):

- Increment seconds in the clock (handle minutes and hours overflow).

- Print the updated time.

- Simulate writing with a random sleep.

- Exit Section:

- Release write\_lock after writing.

- Simulate time between write operations with a random sleep.

4. Main Program:

- Create and start `num\_readers` reader threads and `num\_writers` writer threads.

- Wait for all threads to finish using `join()`.

- Print "All reading and writing operations have been completed."

**Producer-Consumer Problem (Pseudo Code in C)**

1. Initialize:

- buffer[10]: Array representing the buffer with size 10.

- MAX = 10 (maximum size of the buffer).

- lock: Mutex for synchronizing access to the buffer.

2. Producer Function:

- For i = 1 to MAX:

- Wait for some time (simulate work).

- Generate a random item.

- While not inserted:

- Acquire the lock (critical section).

- Check each slot in the buffer for an empty spot.

- If an empty spot is found:

- Insert the item into the buffer.

- Print the producer's action.

- Set inserted = true.

- If no empty spot is found, print "blocked" and wait before trying again.

- Release the lock after inserting.

3. Consumer Function:

- For i = 1 to MAX:

- Wait for some time (simulate work).

- While not consumed:

- Acquire the lock (critical section).

- Check each slot in the buffer for a filled spot.

- If a filled spot is found:

- Consume the item from the buffer.

- Print the consumer's action.

- Set consumed = true.

- If no filled spot is found, print "blocked" and wait before trying again.

- Release the lock after consuming.

4. Main Program:

- Create producer and consumer threads.

- Start producer and consumer threads.

- Wait for all threads to finish.

- Print "All items have been produced and consumed."

**Producer-Consumer Problem with Semaphores (Pseudo Code in C)**

1. Initialize:

- buffer[10]: Array representing the buffer with size 10.

- MAX = 10 (maximum size of the buffer).

- empty = MAX (Semaphore for empty slots, initialized to MAX).

- full = 0 (Semaphore for full slots, initialized to 0).

- lock: Mutex for synchronizing access to the buffer.

2. Producer Function:

- For i = 1 to MAX:

- Generate a random item.

- While not inserted:

- Wait (empty.acquire()) to ensure there is an empty slot in the buffer.

- Acquire the lock (critical section).

- Loop through the buffer to find an empty slot.

- If an empty slot is found:

- Insert the item into the slot.

- Print the producer's action.

- Set inserted = true.

- Release the lock after inserting.

- Signal (full.release()) that an item has been produced.

- If no slot found, print "blocked" and retry.

3. Consumer Function:

- For i = 1 to MAX:

- While not consumed:

- Wait (full.acquire()) to ensure there is a full slot in the buffer.

- Acquire the lock (critical section).

- Loop through the buffer to find a filled slot.

- If a filled slot is found:

- Consume the item from the slot.

- Print the consumer's action.

- Set consumed = true.

- Release the lock after consuming.

- Signal (empty.release()) that an item has been consumed.

- If no filled slot found, print "blocked" and retry.

4. Main Program:

- Create and start producer and consumer threads.

- Wait for all threads to finish.

- Print "All items have been produced and consumed."

**FCFS (First-Come, First-Served) Scheduling Algorithm (Pseudo Code in C)**

1. Initialize:

- process[]: Array to hold processes.

- current\_time = 0.

- gantt[]: Array to store the Gantt chart information.

2. Sort processes[] by arrival time.

3. For each process in processes[]:

- If current\_time < process.arrival\_time:

- Set current\_time = process.arrival\_time (wait until the process arrives).

- Calculate start\_time = current\_time.

- Set process.completion\_time = start\_time + process.burst\_time.

- Set process.turnaround\_time = process.completion\_time - process.arrival\_time.

- Set process.waiting\_time = process.turnaround\_time - process.burst\_time.

- Add (process.pid, start\_time, process.completion\_time) to gantt[].

- Update current\_time = current\_time + process.burst\_time.

4. Print process details (Completion Time, Turnaround Time, Waiting Time).

5. Display the Gantt chart based on gantt[].

6. End.

**Algorithm: Priority Preemptive Scheduling**

1. \*\*Define Structure\*\*:

- Structure `Process` with:

- `pid`, `arrival\_time`, `burst\_time`, `priority`, `remaining\_time`, `completion\_time`, `turnaround\_time`, `waiting\_time`.

2. \*\*Initialize\*\*:

- Set `current\_time = 0` and `completed = 0`.

3. \*\*While Not All Processes Are Completed\*\*:

- Get list of processes that have arrived and have remaining time.

- If available processes:

- Select the process with the highest priority (lowest priority number).

- Run the selected process for 1 time unit.

- Update `remaining\_time` and check if the process is completed.

- Calculate `completion\_time`, `turnaround\_time`, and `waiting\_time` for completed processes.

- If no processes are available, increment `current\_time`.

4. \*\*Draw Gantt Chart\*\*.

5. \*\*End\*\*.

**Algorithm: Round Robin Scheduling**

1. \*\*Define Structure\*\*:

- Structure `Process` with:

- `pid`, `arrival\_time`, `burst\_time`, `remaining\_time`, `completion\_time`, `turnaround\_time`, `waiting\_time`.

2. \*\*Initialize\*\*:

- Create an empty queue `queue`.

- Set `current\_time = 0`.

3. \*\*While There Are Processes or Queue is Not Empty\*\*:

- Add processes to `queue` that have arrived (`arrival\_time <= current\_time`).

- If `queue` is not empty:

- Get the first process from the queue.

- Calculate `start\_time` and check if process can finish within the time quantum:

- If yes, update `remaining\_time`, `completion\_time`, `turnaround\_time`, and `waiting\_time`.

- If no, reduce `remaining\_time` by `time\_quantum` and reinsert the process back into `queue`.

- If `queue` is empty, increment `current\_time`.

4. \*\*Draw Gantt Chart\*\*.

5. \*\*End\*\*.

**Algorithm: Shortest Job First (SJF) Non-Preemptive Scheduling**

1. \*\*Define Structure\*\*:

- Structure `Process` with:

- `pid`, `arrival\_time`, `burst\_time`, `completion\_time`, `turnaround\_time`, `waiting\_time`.

2. \*\*Sort Processes\*\*:

- Sort processes by `arrival\_time`, then by `burst\_time` (ascending).

3. \*\*Initialize\*\*:

- Set `current\_time = 0`.

- Create an empty list for completed processes `completed\_processes`.

4. \*\*While Processes Are Not Empty\*\*:

- Find processes that have arrived by `current\_time`.

- If available processes exist:

- Select the process with the shortest `burst\_time`.

- Update `current\_time` by adding the `burst\_time` of the selected process.

- Compute `completion\_time`, `turnaround\_time`, and `waiting\_time` for the process.

- Add process to `completed\_processes` and record its execution in `gantt` chart.

- Else, increment `current\_time` (when no process is available).

5. \*\*Draw Gantt Chart\*\*.

6. \*\*End\*\*.

**Algorithm: Shortest Job First (SJF) Preemptive Scheduling**

1. \*\*Define Structure\*\*:

- Structure `Process` with:

- `pid`, `arrival\_time`, `burst\_time`, `remaining\_time`, `completion\_time`, `turnaround\_time`, `waiting\_time`.

2. \*\*Initialize\*\*:

- Set `current\_time = 0`.

- Set `completed = 0` (to track completed processes).

- Create an empty list `gantt` to store process execution details.

3. \*\*While Not All Processes Are Completed\*\*:

- Get available processes that have arrived and have remaining time > 0.

- If available processes exist:

- Select the process with the shortest `remaining\_time`.

- Increment `current\_time` by 1.

- Decrease the `remaining\_time` of the selected process by 1.

- Add process details to the `gantt` chart.

- If process is completed (`remaining\_time == 0`):

- Calculate `completion\_time`, `turnaround\_time`, and `waiting\_time`.

- Mark the process as completed and print its details.

4. \*\*If No Process Is Available\*\*:

- Increment `current\_time`.

5. \*\*Draw Gantt Chart\*\* to visualize process execution.

6. \*\*End\*\*.

**Algorithm: Banker's Algorithm for Safety Check**

1. \*\*Define Structures\*\*:

- `available[]`: Array representing available resources.

- `max\_demand[][]`: Matrix representing the maximum demand of resources for each process.

- `allocation[][]`: Matrix representing resources allocated to each process.

- `need[][]`: Matrix representing remaining needs of resources for each process.

- `finish[]`: Boolean array indicating if each process is finished.

- `safe\_sequence[]`: Array to store the safe sequence of processes.

2. \*\*Initialize\*\*:

- Copy `available[]` to `work[]` (resources available for allocation).

- Set `finish[]` to `false` (no process is finished initially).

3. \*\*While Safe Sequence is Incomplete\*\*:

- Set `allocated\_in\_this\_round = false` (flag to track if any process is allocated).

- For each process `i`:

- If process `i` is not finished and its `need[i][j] <= work[j]` for all resources `j`:

- Mark process `i` as finished (`finish[i] = true`).

- Add allocated resources from `allocation[i]` to `work[]`.

- Add process `i` to `safe\_sequence[]`.

- Set `allocated\_in\_this\_round = true`.

4. \*\*If No Process is Allocated in This Round\*\*:

- Return `false` (unsafe state).

5. \*\*End of Loop\*\*:

- If all processes are finished, return `true` (safe state) and print `safe\_sequence[]`.

- Otherwise, the system is unsafe.

6. \*\*End\*\*.

**Page Replacement Algorithms (FIFO, LRU, Optimal)**

**1. FIFO (First-In-First-Out) Algorithm**

1. Initialize:

- memory[]: Array to hold pages in memory.

- page\_faults = 0.

- capacity: Maximum number of pages that memory can hold.

2. For each page in pages[]:

- If the page is not in memory:

- If memory has space:

- Add the page to memory.

- Else:

- Remove the first page (FIFO).

- Add the new page.

- Increment page\_faults by 1.

- Print memory state after each step (optional).

3. Return page\_faults.

**2. LRU (Least Recently Used) Algorithm**

1. Initialize:

- memory[]: Array to hold pages in memory.

- page\_faults = 0.

- page\_index[]: Array to store last accessed index for each page.

- capacity: Maximum number of pages memory can hold.

2. For each page in pages[] with index `i`:

- If the page is not in memory:

- If memory has space:

- Add the page to memory.

- Else:

- Find the least recently used (LRU) page by checking the last accessed index using page\_index[].

- Remove the LRU page.

- Add the new page.

- Increment page\_faults by 1.

- Update the last accessed index of the current page to `i`.

- Print memory state after each step (optional).

3. Return page\_faults.

**3. Optimal Algorithm**

1. Initialize:

- memory[]: Array to hold pages in memory.

- page\_faults = 0.

- capacity: Maximum number of pages memory can hold.

2. For each page in pages[] with index `i`:

- If the page is not in memory:

- If memory has space:

- Add the page to memory.

- Else:

- Calculate future indices for each page in memory[] (i.e., the next time each page will be used).

- Replace the page in memory[] that has the farthest next use.

- Increment page\_faults by 1.

- Print memory state after each step (optional).

3. Return page\_faults.

**Note:**

* FIFO replaces the oldest page in memory.
* LRU replaces the least recently used page.
* Optimal replaces the page that won't be used for the longest time in the future.