1. Project Description

The purpose of this project is to simulate MapReduce on a single host in order to study operating-system concepts like parallelism, IPC and synchronization. Two workloads are implemented with both multithreading and multiprocessing.

Part 1 — Parallel Sorting

Map: Split an array into chunks; each worker sorts its chunk (QuickSort).
Reduce: Merge all sorted chunks into one globally sorted array.

Part 2 — Max-Value Aggregation (Single-Int Shared Buffer)

Map: Each worker scans its chunk to find a local maximum.
Reduce: Workers compete to update a single shared integer (the global max) using synchronization to avoid races.

Why multithreading, multiprocessing, and synchronization?

Multithreading: Low overhead, shared address space; ideal when workers share data structures. Requires synchronization to protect critical sections (e.g., global max update).

Multiprocessing: Process isolation is closer to real MapReduce workers; requires IPC (shared memory, pipes, or semaphores) to pass results and coordinate. More overhead (fork/IPC) but clearer failure isolation.

Synchronization: Necessary whenever multiple workers touch shared state (the reducer's data structures, or the single-int global max). Without it, lost updates and inconsistent results occur.

2. Instructions

Build:

```
make clean && run
```

Run:

```
./bin/thread_sort    32 4
./bin/process_sort  32 4
./bin/thread_max    32 4
./bin/process_max  32 4
```

Larger Examples:

```
./bin/thread_sort   131072 8
./bin/process_sort  131072 8
./bin/thread_max    131072 8
./bin/process_max   131072 8
```

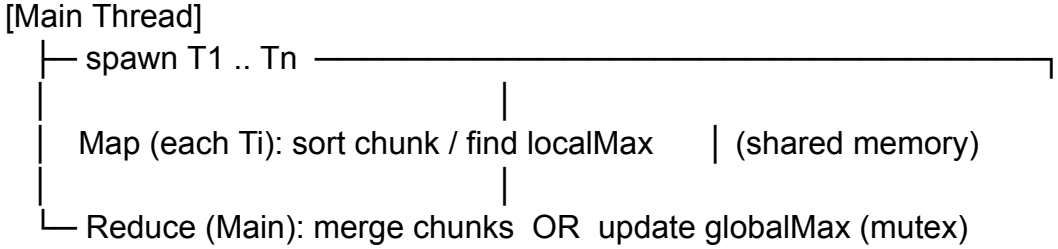
Running Helper Scripts:

```
bash scripts/run_all_small.sh
bash scripts/run_all_large.sh
```

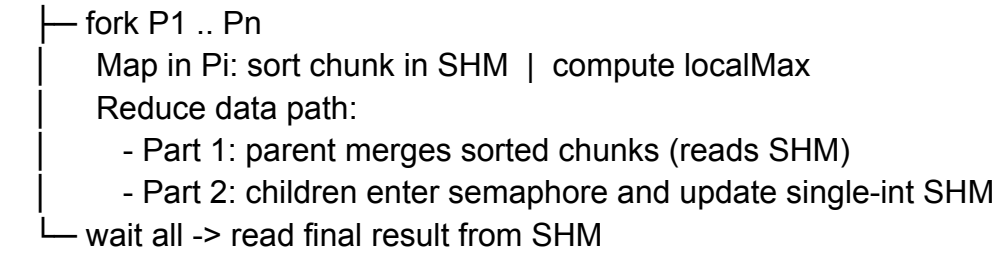
3. Structure of the Code:

```
mapreduce-os-project/
├── src/
│   ├── thread_sort.c      # Part 1 — threads
│   ├── process_sort.c     # Part 1 — processes + shm
│   ├── thread_max.c       # Part 2 — threads + mutex
│   └── process_max.c      # Part 2 — processes + shm + semaphore
├── scripts/               # run & summarize helpers
├── bin/                   # built binaries
└── results/               # logs / screenshots
```

3.1 Architecture Diagrams (Threads)



3.1.1 Architecture Diagrams (Processes)



Map: Independent chunk processing (sort or local max).

Shuffle/Gather (implicit): Parent collects chunk outputs (already in SHM for processes / in-memory arrays for threads).

Reduce: Single reducer combines: K-way merge (Part 1) or max over locals (Part 2).

4. Description of the Implementation Required by Project Requirements

4.1 Tools/Libraries

Language: C (POSIX)

Threads: pthread

Processes/IPC: fork, shm_open/mmap, sem_open (POSIX named semaphore)

Timing/memory: gettimeofday, /proc/self/statm

4.2 Process Management

Threads: pthread_create per worker; pthread_join at barrier.

Processes: fork per worker; waitpid for completion.

4.3 IPC Mechanisms

Part 1 (processes): Input array resides in a shared memory segment; each child sorts its assigned slice in place; parent merges slices.

Part 2 (processes): A shared memory single int stores the global max; children compute localMax and update via a semaphore-protected critical section.

4.4 Threading Model

Manual thread creation (no thread pool) since worker count is small and static (1,2,4,8).

4.5 Synchronization

Threads (Part 2): pthread_mutex_t protects the globalMax update:

lock -> read globalMax -> compare -> maybe write -> unlock

Processes (Part 2): POSIX named semaphore wraps the same critical section for the SHM integer.

Sorting reduce (Part 1): Merging is done by the reducer (single thread/process), so no lock required there.

4.6 Performance Measurement

Time: wall-clock (gettimeofday) measured from before map to after reduce.

Memory: RSS from /proc/self/statm (approx).

Scenarios: Workers = 1,2,4,8; Sizes = 32 (correctness) and 131,072 (performance). Each case repeated 5×; median reported.

5. Performance Evaluation

Result of Code:

=== thread_sort results ===
Array size: 32
Workers (threads): 4
Time: 0.204 ms
Approx RSS: 1536 KB
Final sorted array:
[53523743, 71876166, 175389892, 229233696, 291474504, 442951012, 469976352, 537146758, 590056433, 647800535, 682599717, 708592740, 783815874, 907283241, 975807809, 1113801033, 1183169448, 1192349579, 1212580698, 1232315727, 1253520176, 1350496504, 1366999021, 1483128881, 1564541169, 1596161259, 1633529762, 1643643143, 1709672141, 1781548307, 1854614940, 2108313867]
=== process_sort results ===
Array size: 32
Workers (processes): 4
Time: 0.469 ms
Approx Parent RSS: 1536 KB
Final sorted array:
[53523743, 71876166, 175389892, 229233696, 291474504, 442951012, 469976352, 537146758, 590056433, 647800535, 682599717, 708592740, 783815874, 907283241, 975807809, 1113801033, 1183169448, 1192349579, 1212580698, 1232315727, 1253520176, 1350496504, 1366999021, 1483128881, 1564541169, 1596161259, 1633529762, 1643643143, 1709672141, 1781548307, 1854614940, 2108313867]
=== thread_max results ===
Array size: 32
Workers (threads): 4
Global max: 2108313867
Time: 0.211 ms
Approx RSS: 1536 KB
Input array:
[71876166, 708592740, 1483128881, 907283241, 442951012, 537146758, 1366999021, 1854614940, 647800535, 53523743, 783815874, 1643643143, 682599717, 291474504, 229233696, 1633529762, 175389892, 1183169448, 1212580698, 1596161259, 2108313867, 469976352, 975807809, 1113801033, 1232315727, 1192349579, 1564541169, 1350496504, 1709672141, 1253520176, 590056433, 1781548307]

=== process_max results ===
Array size: 32
Workers (processes): 4
Global max: 2108313867
Time: 0.551 ms
Approx Parent RSS: 1664 KB
Input array:
[71876166, 708592740, 1483128881, 907283241, 442951012, 537146758, 1366999021, 1854614940, 647800535, 53523743, 783815874, 1643643143, 682599717, 291474504, 229233696, 1633529762, 175389892, 1183169448, 1212580698, 1596161259, 2108313867, 469976352, 975807809, 1113801033, 1232315727, 1192349579, 1564541169, 1350496504, 1709672141, 1253520176, 590056433, 1781548307]

6. Conclusion

6.1 Key Findings

- The patterns within MapReduce map cleanly to single-host implementations with threads/processes.
- Scaling helps on more large inputs, but speedup saturates due to merge costs (as shown in Part 1) and lock contention (as shown in Part 2).
- Threads outperform for these in-memory workloads; processes better mimic distributed isolation at some overhead.

6.2 Challenges

There were a multitude of challenges when completing this program. The main challenges in this project were ensuring correct synchronization for the single-integer shared buffer, designing a reducer that could merge all mapper outputs cleanly without introducing extra locking overhead, and obtaining consistent timing results on a shared machine. To address the last issue, each experiment was run multiple times and the median values were reported, which helped smooth out noise from background processes and scheduling differences.

6.3 Limitations & Future Work

- Single reducer; real MapReduce shards keys across many reducers.
- Single node; no network or failure handling.

- Only max aggregation; future: sums, histograms, word count, and k-way parallel merge or tournament tree to speed reduce.
- Explore thread pools and work-stealing for better load balance on uneven partitions.