414 Operating Systems

By: Jacob Komi, Alexis, Q

Introduction

The Process and Thread Manager project provides a way for users to run several tasks at the same time in FreeBSD. The purpose is to mimic basic operating system functions such as creating processes, managing threads, synchronizing them, and communicating amongst processes (IPC).

Writing OS-level software in C++ with vi/vim and a Makefile gives you hands-on experience and reinforces what you've learned in class. The project brings together several subsystems, such as process lifecycle management, POSIX threads, mutexes, semaphores, condition variables, pipes, shared memory, and thread pooling, into a single, working design.

Background

Operating systems rely on concurrency to support multitasking, responsiveness, and efficient computation.

Processes provide isolated execution, while threads provide lightweight parallelism within a shared address space. Synchronization and communication mechanisms allow these units to exchange data safely and avoid race conditions.

This project recreates these core OS principles, providing practical insight into:

PCB (Process Control Block) and TCB (Thread Control Block) structures

Lifecycle and state transitions of processes and threads

Preventing race conditions and deadlocks

Communication via pipes, FIFOs, and shared memory

Efficient work distribution using thread pools

Design Overview

The system uses a **modular architecture**, separating responsibilities across dedicated layers to make the design clean, maintainable, and scalable.

Each subsystem—Process Management, Thread Management, Synchronization, IPC, and Thread Pooling—is implemented independently and interacts through flexible APIs.

This layered approach mirrors how actual operating systems are structured, allowing components to be modified or expanded without breaking the entire system.

The architecture encourages incremental development, easier debugging, and clear separation of concerns.

Process Management Subsystem

Implements a **Process Control Block (PCB)** structure that stores essential information like PID, execution state, parent ID, and communication channels.

Uses FreeBSD system calls such as fork(), exec(), and waitpid() to create new processes, launch executables, and handle child termination.

Tracks all state transitions—Ready, Running, Waiting, Terminated—similar to an actual OS scheduler.

Ensures safe process termination, avoiding zombie processes and managing resources carefully.

Supports child processes communicating back to the parent using IPC tools.

Thread Management Subsystem

Uses a **Thread Control Block (TCB)** to represent each user-level thread, containing its state, ID, and execution context.

Relies on POSIX pthreads to create and manage threads efficiently within each process.

Supports operations such as thread creation, joining, detaching, and safe termination, ensuring that threads complete tasks before resources are released.

Tracks thread states to help manage scheduling and avoid unsafe execution flows.

Allows multiple threads to operate simultaneously on shared memory regions, enabling parallel task execution.

Synchronization Primitives

Mutex locks enforce exclusive access to shared resources, ensuring that only one thread can modify a critical section at a time.

Semaphores control access to limited resources by using counters that restrict how many threads can pass through concurrently.

Condition variables allow threads to sleep and wait for specific conditions to occur, enabling more flexible communication between threads.

These primitives prevent race conditions, data corruption, and inconsistent execution outcomes.

They also help avoid deadlocks and ensure smooth coordination between threads.

Inter-Process Communication (IPC)

Unnamed pipes allow parent and child processes to communicate directly during a fork operation.

Named pipes (FIFOs) create communication channels between unrelated processes, enabling more flexible architectures.

Shared memory enables extremely fast communication by giving processes access to a common memory segment.

The IPC layer includes safeguards using mutexes to prevent simultaneous write collisions in shared memory.

These communication methods demonstrate how OS kernels support coordinated execution across multiple processes.

Thread Pooling

A **fixed number of worker threads** are created in advance and remain alive throughout program execution.

Task queue holds work items that need to be run by worker threads. A **mutex** protects the queue and a **condition variable** lets worker threads sleep when there is no work and wake up when a new task gets added. Together this gives a thread safe and efficient task queue

Efficient parallel task execution because multiple worker threads, a queue of tasks and proper synchronization

System Architecture Summary

The architecture combines user-level process management with kernel-level system calls for realistic behavior.

Both PCB and TCB structures store detailed metadata for each running execution unit.

Synchronization primitives ensure thread-safe access to shared memory and other resources.

IPC components allow coordinated communication between separate processes.

The thread pool integrates everything by allowing tasks to run concurrently and efficiently.

Technical Approach

Implemented entirely in **C++**, using POSIX-compliant APIs for threading and synchronization.

FreeBSD's fork(), exec(), shmget(), and pipe() calls were used to implement process handling and IPC.

A structured Makefile automates building, debugging, cleaning, and as well testing

The folder structure separates code into logical modules such as src/, include/, test/, doc/, and scripts/.

Error handling is robust, checking system call failures and avoiding resource leaks.

Implementation Plan

Milestone 1: Set up FreeBSD environment, GitHub integration, and Makefile.

Milestone 2: Build the complete Process Manager, including PCBs and state tracking

Milestone 3: Develop Thread Manager and TCBs using pthreads.

Milestone 4: Implement mutexes, semaphores, and condition variables

Milestone 5: Build IPC subsystem with pipes and shared memory.

Milestone 6: Create thread pool and integrate task distribution.

Milestone 7: Perform full-system integration and conduction of unit/system tests.

Milestone 8: Prepare documentation, README, and final presentation.

Testing Strategy

Unit tests verify that each subsystem behaves correctly in isolation (e.g., thread creation, pipe communication, semaphore operations).

System tests validate interactions between subsystems, such as threads accessing shared memory with mutexes.

Stress tests evaluate how the system behaves under high thread loads or multiple IPC operations.

Performance tests measure execution time, task throughput, and synchronization overhead.

Fault tests simulate deadlocks, race conditions, and resource exhaustion to ensure resilience.

Performance Considerations

Mutexes and semaphores were used efficiently to minimize lock contention and avoid blocking threads unnecessarily.

Shared memory was selected for high-throughput communication since it outperforms pipes in bandwidth-heavy scenarios.

Deadlock prevention techniques, such as lock ordering and immediate release policies, were practiced throughout implementation.

The thread pool dramatically reduces CPU overhead by preventing repeated thread creation/destruction cycles.

Load distribution across worker threads ensures optimal CPU utilization on multi-core systems.

Demo Instructions

Build the project using:
make build — compiles with optimized flags

./manager_demo –threads 4
Demonstrate shared memory IPC using:
./manager_demo –ipc shm

Run all tests using:
make test – executes unit and system tests

Run base demo:

./manager_demo showcases process and thread behavior.

https://drive.google.com/file/d/1Gt5nFJCkD7V12pOJaQiBAz5G0s3OMjcN/view?ts=69356cc1

```
#ifndef THREAD_MANAGER_H
#define THREAD_MANAGER_H

#include <pthread.h>
#include <string>
#include <vector>

enum class ThreadState {
    RUNNING,
    COMPLETED
};

struct ThreadInfo {
    pthread_t tid;
    std::string name;
    ThreadState state;
};

using ThreadFunc = void* (*)(void*);

class ThreadManager {
public:
    ThreadManager();

    // Create a new thread running `func(arg)`
    pthread_t createThread(ThreadFunc func, void* arg, const std::string& name = "");

    // Join a specific thread
    bool joinThread(pthread_t tid);

    // Join all threads
    void joinAll();

    // Print a table of all threads
    void printThreadTable() const;

private:
    std::vector<ThreadInfo> threads;
};
```

Conclusion

This project successfully recreates a concurrency framework at the user level that works like how current operating systems handle threads, processes, synchronization, and communication between processes. The system shows that it understands OS ideas both theoretically and practically by combining process management, thread lifecycle tracking, IPC mechanisms, and a fully functional thread pool. The performance-focused method and modular design show off strong system programming skills and reinforce basic operating system ideas. Thanks for taking a look at my work.