# Report: Shell and Construction Site Management System

## Introduction

This report outlines the design and implementation of two distinct systems: a simulated construction site management system and a custom shell system. Both systems demonstrate key operating system concepts, including process creation, process control, and inter-process communication (IPC). The report also discusses the project’s goals, the approach to solving the problems, and insights gained during development. Diagrams illustrating the utilization of dup() and dup2() system calls are included, along with a summary of work distribution among team members.

## Goals

### Construction Site Management System

The primary goal of the construction site management system is to simulate the management of resources, workers, and tasks within a construction environment. The system aims to:  
- Demonstrate synchronization and resource allocation.  
- Utilize OS-level mechanisms like semaphores, mutexes, or shared memory for coordination.

### Custom Shell System

The shell system aims to emulate a command-line interface with functionality to:  
- Execute standalone and parameterized commands.  
- Support pipelined execution for advanced command operations.  
- Implement IPC using pipes and demonstrate the functionality of dup() and dup2().

## Approach and Implementation

### Construction Site Management System

The construction site management system was designed to:  
1. Simulate Resource Management:  
 - Resources like tools and materials were managed using shared data structures.  
 - Mutexes and semaphores ensured safe access to shared resources.  
2. Task Scheduling:  
 - Worker tasks were scheduled using a producer-consumer model.  
 - Task completion and worker availability were synchronized using condition variables.  
3. Error Handling and Debugging:  
 - Proper error handling was implemented for shared memory allocation and semaphore operations.

### Custom Shell System

The shell system was implemented to:  
1. Execute Commands:  
 - Commands were parsed into tokens and executed using the execvp() system call.  
2. Handle Pipelining:  
 - Commands were split at the | symbol, and pipes were created to connect processes.  
 - File descriptors were redirected using dup2() to manage data flow.  
3. Error Handling:  
 - Invalid commands and system call failures were reported with descriptive error messages.  
4. Resource Management:  
 - Allocated memory for command arguments was freed after execution.  
 - Open file descriptors were properly closed to prevent resource leaks.

## Utilization of dup() and dup2()

### Explanation of dup()

dup() duplicates an existing file descriptor and returns the lowest available file descriptor.  
Example:  
  
int fd = dup(1); // Duplicate stdout  
write(fd, "Hello, World!\n", 14); // Outputs to the terminal

### 

### 

### Explanation of dup2()

dup2() duplicates an existing file descriptor to a specific file descriptor number.  
Example:  
  
int file\_fd = open("output.txt", O\_WRONLY | O\_CREAT, 0644);  
dup2(file\_fd, 1); // Redirect stdout to the file  
printf("This goes to the file\n");  
close(file\_fd);

### Diagram

A notebook with writing on it

Description automatically generated

## Work Distribution

Group Member Contributions:  
1. Faraz Ibrar:  
 - Implemented the construction site management system.  
 - Designed shared memory and synchronization mechanisms.  
 - Documented system-level concepts for resource management.  
2. Amir Hamza:  
 - Implemented the custom shell system.  
 - Developed tokenization and pipelining features.  
 - Handled error management and system call integration.  
3. Mishkat Fatima:  
 - Created flowcharts and diagrams illustrating dup() and dup2().  
 - Conducted testing and debugging for both systems.  
 - Consolidated the documentation and prepared the final report.

## Lessons Learned

- Gained hands-on experience with process creation, control, and IPC.  
- Understood the practical application of file descriptor manipulation.  
- Learned to break down complex problems and distribute tasks efficiently.  
- Improved skills in debugging and handling edge cases.  
- Developed robust mechanisms to gracefully handle system call failures and invalid inputs.

## Improvements and Future Work

1. Construction Site Management System:  
- Integrate a GUI to visualize resource allocation and worker tasks.  
- Implement advanced scheduling algorithms for task assignment.  
2. Custom Shell System:  
- Add support for background processes using &.  
- Implement input and output redirection (>, <).  
- Enhance with job control features (e.g., fg, bg, jobs).

## Conclusion

This project provided a comprehensive understanding of operating system principles and their real-world applications. By implementing both the construction site management system and the custom shell, we developed skills in process synchronization, IPC, and error management. The systems are modular and can be extended with additional features for greater functionality and user interactivity.

# Simulated Construction Site Management System

## Introduction

Construction site management involves the efficient use of resources, synchronized task execution, and worker coordination. This report outlines the design and implementation of a simulated construction site management system using core operating system concepts. The simulation incorporates resource management, thread synchronization, memory management, and dynamic task assignment to replicate real-world challenges and solutions.

## Goals

1. Efficiently manage and allocate limited resources.

2. Coordinate tasks among workers with proper synchronization.

3. Implement priority scheduling to handle critical tasks effectively.

4. Simulate realistic worker behavior, including fatigue and breaks.

5. Handle errors and external factors such as adverse weather conditions.

6. Integrate dynamic resource generation and task assignment.

## System Design

### Key Components

1. Resource Management:  
- Simulated resources: Bricks, cement, tools.  
- Replenishment mechanisms based on usage patterns and external factors.  
  
2. Thread Management:  
- Tasks such as bricklaying, cement mixing, and scaffolding are implemented as threads.  
- Threads are created or terminated dynamically based on resource availability.  
  
3. Synchronization:  
- Mutex and semaphores ensure synchronized access to shared resources.  
- Data integrity is maintained by allowing only one thread to access critical resources at a time.  
  
4. Memory Management:  
- Shared memory stores information about resources, worker statuses, and task progress.  
- Data structures include priority queues for task scheduling and worker assignment.  
  
5. Priority Scheduling:  
- Tasks are categorized into high, medium, and low priority levels.  
- A dynamic priority scheduling algorithm selects and assigns tasks based on criticality.  
  
6. Dynamic Resource and Task Management:  
- Resources are generated or degraded over time.  
- Tasks are dynamically assigned to workers based on skills and priorities.  
  
7. Error Handling and I/O Simulation:  
- Handles scenarios such as resource shortages, task conflicts, and delays due to weather conditions.  
- Simulates external factors like adverse weather to adjust task priorities.

### Flowcharts and Diagrams

System Flowchart:  
 --> [Initialize Resources and Workers]  
 --> [Create Threads for Tasks]  
 --> [Check Resource Availability]  
 --> [Yes: Assign Task] --> [Synchronize Resource Access]  
 --> [No: Wait for Resource Replenishment]  
 --> [Dynamic Priority Adjustment Based on Conditions]  
 --> [Update Worker Status and Task Progress]  
 --> [Error Handling (if required)]  
 --> [End Simulation]

Synchronization Process (Using Mutex):  
[Worker Requests Resource] --> [Check Resource Availability]  
 --> [Acquire Mutex Lock]  
 --> [Access Resource]  
 --> [Release Mutex Lock]  
 --> [Worker Proceeds with Task]

## Implementation

### Resource Management

Resources are represented as shared variables. The system replenishes or degrades resources based on defined rules.

// Example of resource replenishment  
pthread\_mutex\_lock(&resource\_mutex);  
if (bricks < MAX\_BRICKS) {  
 bricks++;  
}  
pthread\_mutex\_unlock(&resource\_mutex);

### Thread and Process Management

Tasks such as laying bricks or mixing cement are threads. Threads are synchronized to prevent resource conflicts.

pthread\_create(&worker\_thread, NULL, lay\_bricks, NULL);  
pthread\_join(worker\_thread, NULL);

### Synchronization

Semaphores and mutex locks prevent race conditions.

sem\_wait(&resource\_semaphore);  
// Access shared resource  
sem\_post(&resource\_semaphore);

### Priority Scheduling

Tasks are stored in priority queues, and the scheduling algorithm selects tasks dynamically based on conditions.

high\_priority\_queue = Queue()  
medium\_priority\_queue = Queue()  
low\_priority\_queue = Queue()  
  
if not high\_priority\_queue.empty():  
 execute\_task(high\_priority\_queue.get())  
elif not medium\_priority\_queue.empty():  
 execute\_task(medium\_priority\_queue.get())  
else:  
 execute\_task(low\_priority\_queue.get())

### Dynamic Resource and Task Assignment

Tasks are dynamically assigned based on worker skills and priority.

for worker in available\_workers:  
 if worker.skill == required\_skill and worker.is\_available:  
 assign\_task(worker, task)

### Error Handling

Unexpected scenarios are handled gracefully.

if (resources < required\_resources) {  
 printf("Error: Insufficient resources\n");  
 // Replenish resources or wait  
}

## Learning Outcomes

1. Synchronization Techniques:  
- Mastered the use of mutex and semaphores to manage shared resources.  
  
2. Thread Management:  
- Learned to create, manage, and synchronize threads.  
  
3. Dynamic Priority Scheduling:  
- Developed a dynamic system to handle task prioritization in real time.  
  
4. Error Handling:  
- Implemented robust error handling to manage unexpected scenarios.  
  
5. Resource Optimization:  
- Gained insights into efficient resource allocation and utilization.

**Team Members and Responsibilities**

**Amir Hamza: Resource Management and Synchronization**

- Designed the system for simulating and managing resources.

- Implemented synchronization using mutex and semaphores.

**Mishkat Fatima: Thread Management and Priority Scheduling**

- Developed the thread-based task system.

- Implemented dynamic priority scheduling algorithms.

**Faraz Ibrar: Memory Management and Dynamic Task Assignment**

- Optimized memory usage and implemented shared memory structures.

- Designed the task assignment logic considering worker skills and priorities.

## Conclusion

This project demonstrates how operating system concepts such as thread management, synchronization, and priority scheduling can be applied to simulate a construction site management system. The implementation highlights the importance of efficient resource management and dynamic task assignment. Future enhancements could further improve the simulation’s realism and scalability, providing deeper insights into construction site management challenges.