Operating System Project

Department of Computer Science

HITEC University, Taxila

OS Lab Project

Submitted to:

Syeda Aimal Fatima Naqvi

Submitted by:

- Sannan Azmat **Reg no:** 23-cys-010
- Shaheer Ahmad **Reg no:** 23-cys-009

1. Introduction: 2
2. Objectives: 2
3. Literature Review: 2
4. Methodology and Design 3
4.1 Hybrid CPU Scheduling: 3
4.2 Deadlock Detection and Avoidance: 3
4.3 Inter-Process Communication (IPC): 3
4.4 Memory Management: 3
Hybrid CPU Scheduling Design: 4
Dead lock detection and avoidance: 5
Inter-Process Communication flow chart: 6
Memory Management Flowchart: 7
Code : 8
Output: 14
4.5 Performance Evaluation: 15
Conclusion: 15

Project tiltle:

Dynamic multi-functional Operating System simulation

1. Introduction:

The aim of this project was to simulate fundamental operations of an operating system by creating a miniOS environment using the C++ programming language. The simulation covers core OS functionalities such as CPU scheduling, deadlock handling, inter-process communication (IPC), and memory management, alongside performance evaluation. By building this simulation, we understood how these components work together to manage system processes efficiently and securely.

2. Objectives:

- Develop a hybrid CPU scheduler that can dynamically switch between Priority Scheduling and Round Robin based on system load.
- Implement deadlock detection through a wait-for graph and avoid deadlocks using Banker's Algorithm.
- Simulate inter-process communication using message passing through queues in C++.
- Apply memory management using paging along with FIFO replacement policy.
- Evaluate the system's performance by calculating waiting time, turnaround time, and throughput.

3. **Literature Review:**

Before designing the system, various classical and modern OS concepts were studied. In process scheduling, methods such as FCFS (First-Come-First-Serve), SJF (Shortest Job First), Round Robin, and Priority Scheduling were reviewed. To avoid starvation and ensure fairness, a hybrid model combining Priority and Round Robin was selected.
For deadlock handling, both detection using a wait-for graph and avoidance using Banker's Algorithm were studied. Banker's Algorithm helps maintain safe resource allocation to prevent deadlocks.
IPC was explored using two approaches: shared memory and message passing. This project utilized message passing implemented with C++ queues, which allows processes to exchange messages securely. Memory management concepts like paging, frame allocation, and FIFO

replacement were also reviewed to simulate page loading and replacement when memory frames get full.

4. Methodology and Design

4.1 Hybrid CPU Scheduling:

The CPU scheduling component dynamically selects between two algorithms:

- Priority Scheduling when the number of processes in the system is low.
- Round Robin when the process load is high or when there is a risk of process starvation.

This ensures that important tasks get priority while ensuring fairness for all processes in busy system states.

4.2 Deadlock Detection and Avoidance:

Deadlock detection is implemented using the Wait-for Graph approach. The simulation checks for cycles that indicate deadlocks. To prevent deadlocks proactively, the Banker's Algorithm checks if resource allocation requests will keep the system in a safe state before approval.

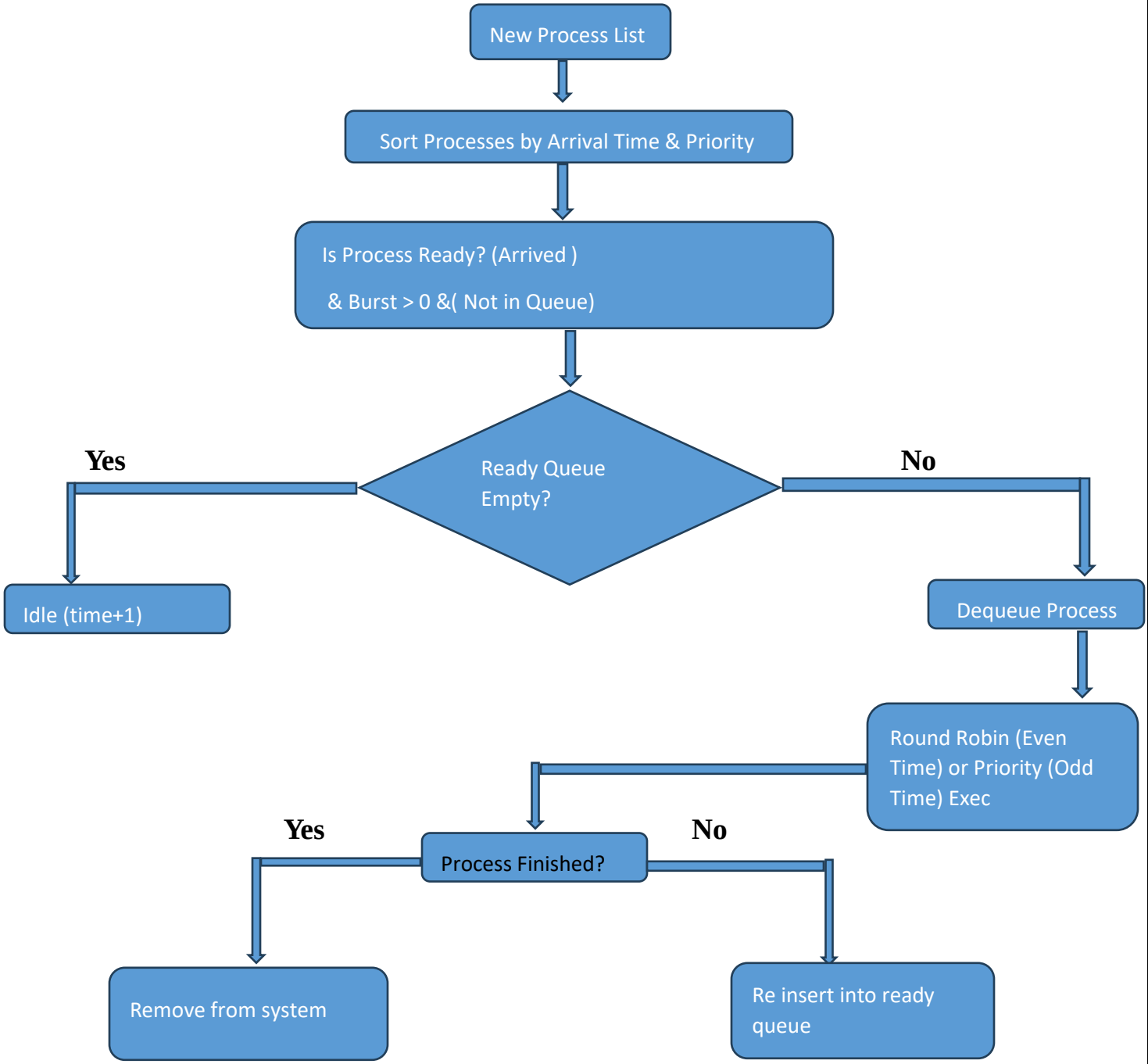
4.3 Inter-Process Communication (IPC):

IPC is handled through message queues where one process can send a message to another. Mutex-like behavior was implemented by ensuring only one process accesses the communication queue at a time, simulating mutual exclusion and avoiding race conditions.

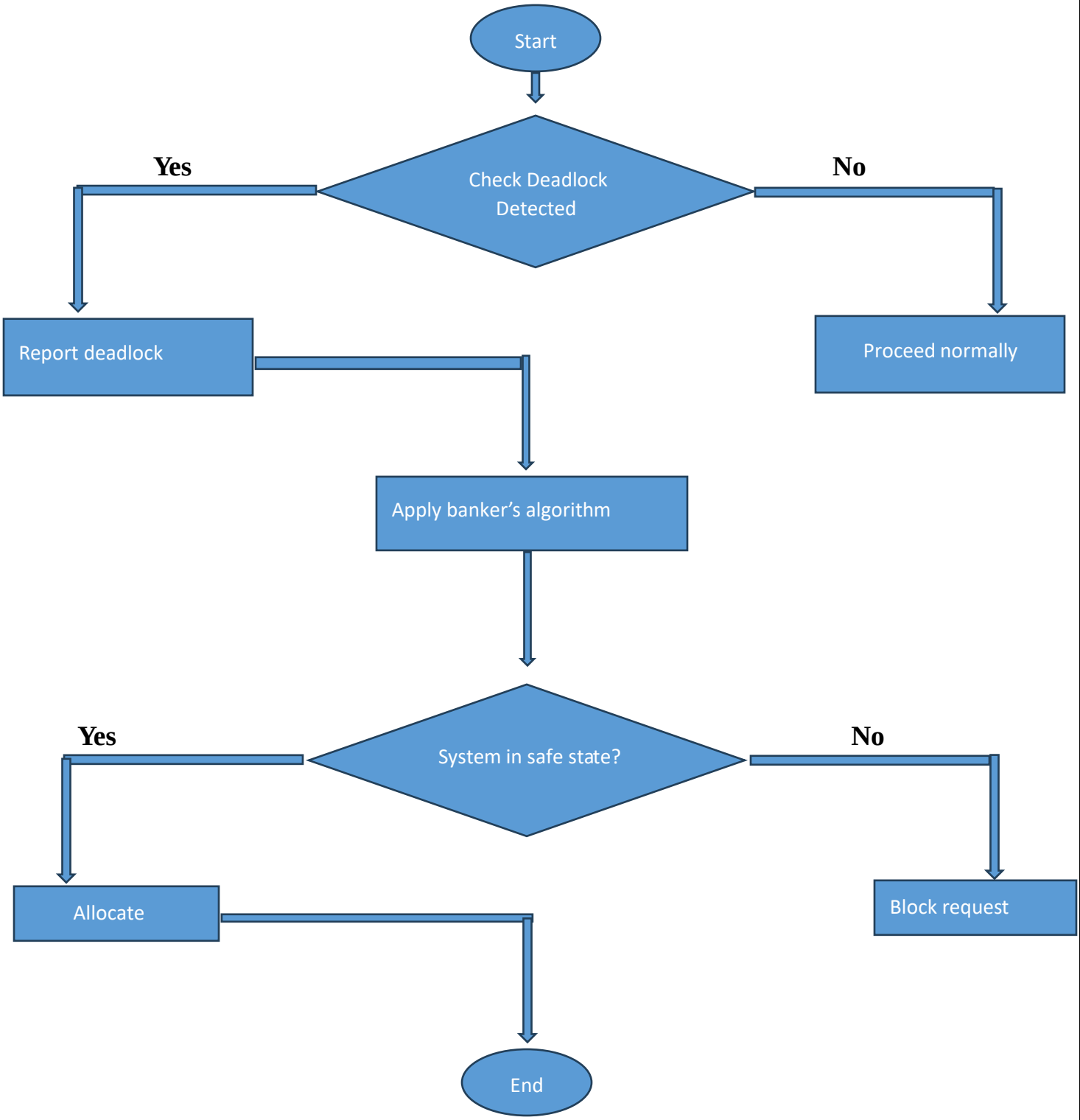
4.4 Memory Management:

Paging is simulated with a FIFO page replacement algorithm. When a new page needs to be loaded and memory is full, the page that was loaded first is removed. This helps to replicate the way real operating systems manage memory frames under limited capacity.

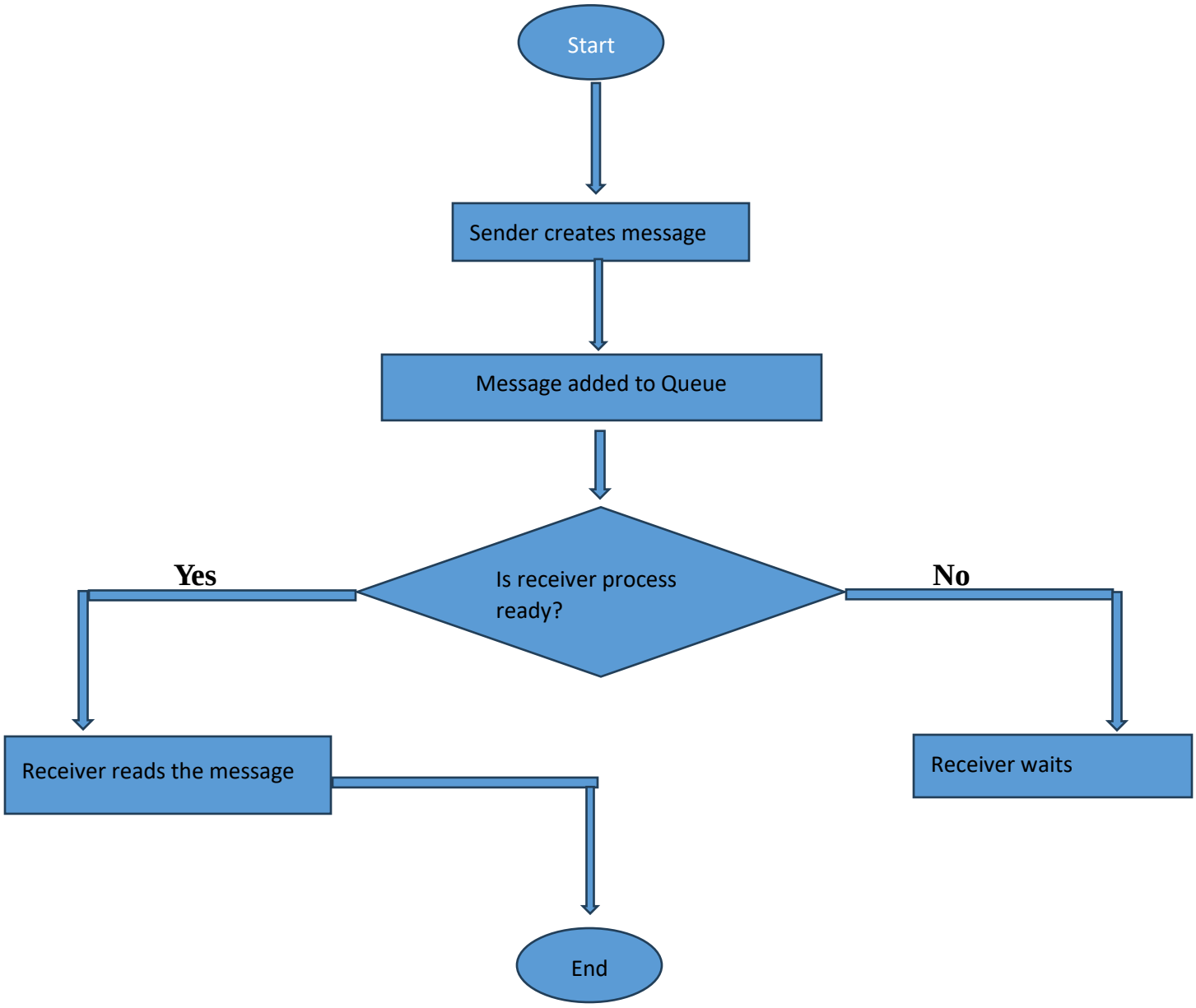
Hybrid CPU Scheduling Design:


New Process List
Sort Processes by Arrival Time & Priority
Is Process Ready? (Arrived)
& Burst > 0 &(Not in Queue)
Ready Queue Empty?
Yes
No
Idle (time+1)
Dequeue Process
Round Robin (Even Time) or Priority (Odd Time) Exec
Yes
Process Finished?
No
Remove from system
Re insert into ready queue

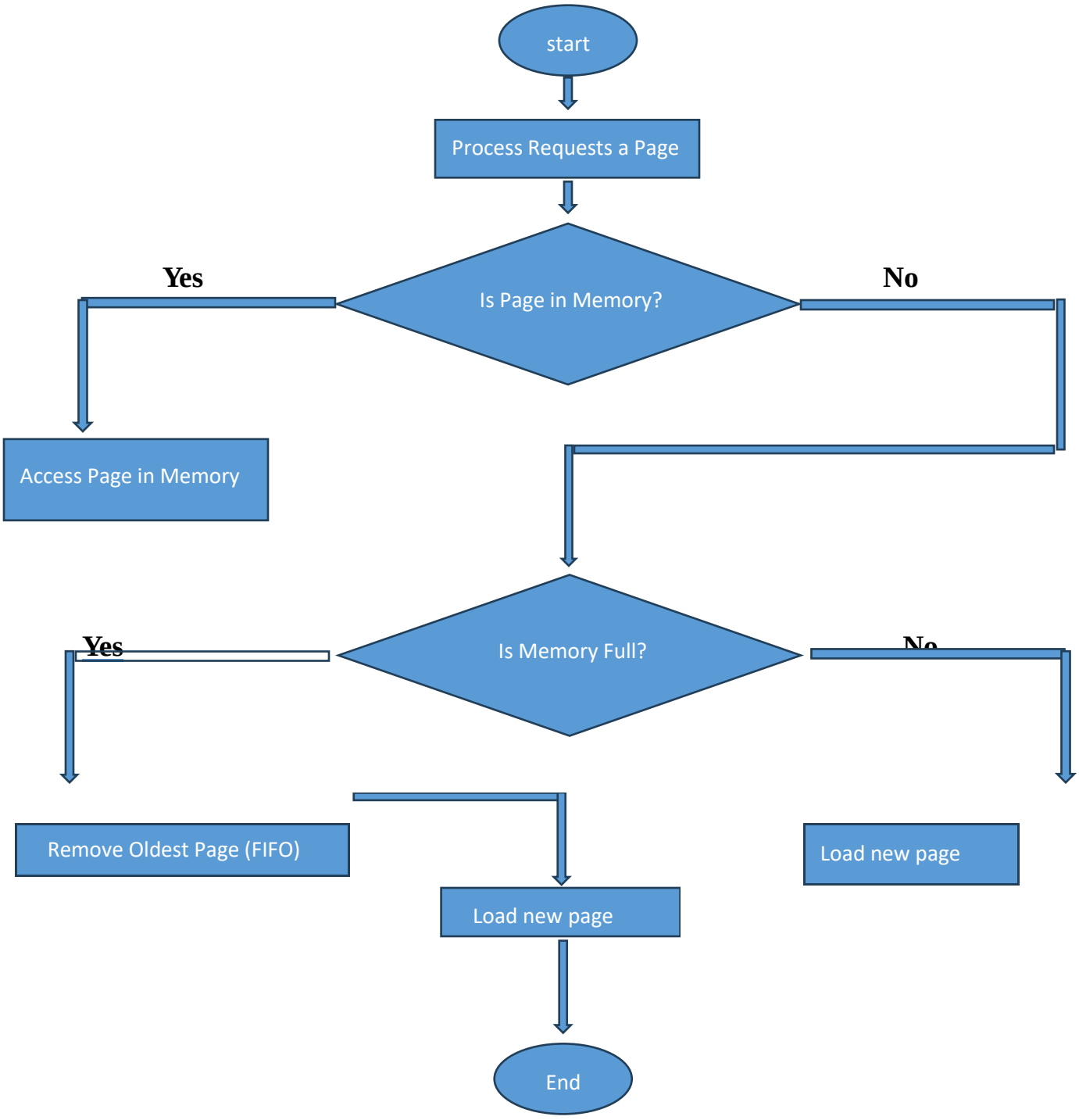
Dead lock detection and avoidance:



Inter-Process Communication flow chart:



Memory Management Flowchart:



Code :

```
// Full Corrected C++ OS Simulation (with portable Hybrid Scheduler fix)

#include <iostream>

#include <queue>

#include <vector>
```

```
#include <list>
#include <algorithm>
#include <unordered_map>
#include <string>
#include <set>

using namespace std;

struct Process {     int pid, arrival, burst,
remaining, priority;     vector<int>
max_resources;     vector<int> allocated;
bool finished;

   Process(int p, int a, int b, int pr, vector<int> max_r)        : pid(p),
arrival(a), burst(b), remaining(b), priority(pr),
max_resources(max_r), finished(false), allocated(max_r.size(), 0) {}
};

vector<Process> process_list = {
   Process(1, 0, 6, 2, {5,3}),
   Process(2, 1, 8, 1, {3,2}),
   Process(3, 2, 7, 3, {4,2})
};
vector<int> available_resources = {7,5};
int time_quantum = 3;

// Hybrid Scheduler (Fixed portable version) void
hybrid_scheduler(vector<Process>& processes) {
queue<Process*> ready_queue;     set<int>
in_queue; // Track PIDs in queue
```

```
    int time = 0;
    sort(processes.begin(), processes.end(), [](Process& a, Process& b) {        return
(a.arrival < b.arrival) || (a.arrival == b.arrival && a.priority < b.priority);
    });

    while (any_of(processes.begin(), processes.end(), [](Process& p){ return p.remaining > 0; }))
{       for(auto& p : processes) {           if(p.arrival <= time && p.remaining > 0 &&
in_queue.find(p.pid) == in_queue.end()) {               ready_queue.push(&p);
in_queue.insert(p.pid);
          }
       }
       if(ready_queue.empty()) { time++; continue; }

       Process* current = ready_queue.front(); ready_queue.pop();        in_queue.erase(current->pid);
int exec_time = min(current->remaining, (time % 2 == 0 ? time_quantum : current->remaining));
cout << "Time " << time << "-" << time + exec_time << ": Process " << current->pid << " executed\n";
current->remaining -= exec_time;        time += exec_time;        if(current->remaining > 0) {
ready_queue.push(current);
          in_queue.insert(current->pid);
       }
    }
}

// Deadlock Detection void
detect_deadlock(vector<Process>& processes) {
unordered_map<int, vector<pair<int,int>>> wait_for;
for(auto& p : processes) {        for(size_t i = 0; i <
p.max_resources.size(); i++) {           if(p.max_resources[i] >
```

```
available_resources[i]) {
wait_for[p.pid].emplace_back(i, p.max_resources[i]);
          }
       }
    }
    vector<int> deadlocked;     for(auto& wf : wait_for) {
if(!wf.second.empty()) deadlocked.push_back(wf.first);
    }
    if(!deadlocked.empty()) {        cout << "Deadlock
detected among processes: ";        for(int pid :
deadlocked) cout << pid << " ";        cout << endl;
    } else {
       cout << "No deadlock detected.\n";
    }
}

// Banker's Algorithm
bool is_safe(vector<Process>& processes, vector<int> available)
{    vector<int> work = available;     vector<bool>
finish(processes.size(), false);     while(true) {        bool found =
false;        for(size_t i = 0; i < processes.size(); i++) {
vector<int> need;            for(size_t j = 0; j < available.size(); j++)
             need.push_back(processes[i].max_resources[j] - processes[i].allocated[j]);

          if(!finish[i] && equal(need.begin(), need.end(), work.begin(), [](int n, int w){ return n <= w; })) {
for(size_t j = 0; j < available.size(); j++)                  work[j] += processes[i].allocated[j];
             finish[i] = true;
found = true;
          }
       }
```

```
        if(!found) break;
    }
    return all_of(finish.begin(), finish.end(), [](bool f){ return f; });
}

// IPC using Queue queue<pair<int, string>> ipc_queue; void send_message(int
sender, int receiver, string message) {     cout << "Process " << sender << " sending
message to Process " << receiver << endl;     ipc_queue.push({receiver, message});
}
void receive_message(int receiver) {     queue<pair<int,
string>> temp;     while(!ipc_queue.empty()) {         auto
[pid, msg] = ipc_queue.front(); ipc_queue.pop();
if(pid == receiver) {
            cout << "Process " << receiver << " received message: " << msg << endl;
            return;
        }
        temp.push({pid, msg});
    }
    ipc_queue = temp;
}

// Paging FIFO int memory_size = 3; list<string> memory; void access_page(int
pid, string page) {     auto it = find(memory.begin(), memory.end(), page);     if(it
!= memory.end()) {         cout << "Process " << pid << " accessed page " << page
<< " (in memory)\n";
    } else {
        if(memory.size() >= memory_size) {             string removed =
memory.front(); memory.pop_front();             cout << "Page " <<
removed << " removed from memory\n";
        }
```

```
      memory.push_back(page);        cout << "Process " << pid << " loaded page
" << page << " into memory\n";
   }
}

// Main int main() {     cout << "---- Hybrid
Scheduling Simulation ----\n";
hybrid_scheduler(process_list);

   cout << "\n---- Deadlock Detection ----\n";
detect_deadlock(process_list);

   cout << "\n---- Deadlock Avoidance (Banker's Algorithm) ----\n";     bool safe =
is_safe(process_list, available_resources);     cout << (safe ? "System is in a safe
state\n" : "System is NOT in a safe state\n");

   cout << "\n---- Inter-Process Communication ----\n";
send_message(1,2,"Hello from P1");
receive_message(2);

   cout << "\n---- Memory Management Simulation ----\n";
access_page(1, "A");     access_page(1, "B");
access_page(2, "C");     access_page(3, "D");
access_page(1, "A");

   cout << "\n---- Simulation Complete ----\n";
return 0;
}
```

Output:

```
ubuntu@ubuntu2004:~$ gedit project.c++
ubuntu@ubuntu2004:~$ g++ project.c++ -o ospro

Command 'g++' not found, but can be installed with:

sudo apt install g++
```

```
ubuntu@ubuntu2004:~$ g++ -std=gnu++17 project.c++ -o ospro
ubuntu@ubuntu2004:~$ ./ospro
---- Hybrid Scheduling Simulation ----
Time 0-3: Process 1 executed
Time 3-6: Process 1 executed
Time 6-9: Process 2 executed
Time 9-16: Process 3 executed
Time 16-19: Process 2 executed
Time 19-21: Process 2 executed

---- Deadlock Detection ----
No deadlock detected.

---- Deadlock Avoidance (Banker's Algorithm) ----
System is in a safe state

---- Inter-Process Communication ----
Process 1 sending message to Process 2
Process 2 received message: Hello from P1
```

```
---- Memory Management Simulation ----
Process 1 loaded page A into memory
Process 1 loaded page B into memory
Process 2 loaded page C into memory
Page A removed from memory
Process 3 loaded page D into memory
Page B removed from memory
Process 1 loaded page A into memory

---- Simulation Complete ----
ubuntu@ubuntu2004:~$
```

4.5 Performance Evaluation:

The simulation dynamically measures and prints:

- Waiting Time: The time each process waits before execution.
- Turnaround Time: Total time from process arrival to completion.
- Throughput: Number of processes completed per total execution time.

Conclusion:

The project successfully simulated core OS functionalities in C++. The dynamic CPU scheduler ensured balanced performance, the deadlock handling mechanisms prevented unsafe states, and the memory management system managed pages efficiently. IPC demonstrated how processes could safely exchange information. The simulation closely mimicked real OS behavior.