A project report

Submitted in the partial fulfilment of the requirements for the award of degree of

Bachelor of Technology in Computer Science and Engineering (Data Science with Machine Learning)

Submitted to

LOVELY PROFESSIONAL UNIVERSITY

PHAGWARA, PUNJAB

SUBMITTED BY:

1. Fayaz Basha Shaik (1240063)
2. Sangeeth S (12416188)
3. Yashashree Ganesh Borkar (12409867)

TABLE OF CONTENTS :

1. INTRODUCTION

Multiprogramming operating systems allow multiple programs to run concurrently by sharing the CPU, memory, and I/O resources. However, as system load increases, static resource allocation becomes inefficient and may lead to scenarios like bottlenecks, starvation, memory thrashing, or underutilized CPU time. As workloads become more dynamic—especially in environments that execute CPU-heavy and memory-heavy processes—there arises a need for an adaptable allocation mechanism that responds to real-time system behavior.

Adaptive Resource Allocation is an intelligent strategy that continuously monitors system performance and adjusts resource distribution among active processes. Instead of relying on predetermined, rigid thresholds, adaptive systems react to current CPU utilization, memory pressure, and per-process behavior to optimize throughput and ensure system stability. Techniques like feedback loops, dynamic threshold adjustments, and priority-based selection help maintain system responsiveness under varying loads.

This project implements a practical, real-time Adaptive Resource Allocation module using Python, psutil, PyQt6, and subprocess-based multiprocessing. It demonstrates how a modern system can intelligently pause, resume, or redistribute process execution based on current resource conditions. The accompanying GUI allows users to visualize CPU usage, memory usage, and job states, providing an intuitive understanding of how adaptive systems maintain efficiency in multiprogramming environments.

2. PROJECT OVERVIEW

Multiprogramming systems allow multiple processes to execute concurrently by sharing available system resources such as CPU, memory, and I/O bandwidth. However, static allocation techniques may lead to several performance issues such as resource starvation, bottlenecks, unbalanced execution, and inefficient utilization.

This project implements an **Adaptive Resource Allocation System** that dynamically adjusts resource distribution among active processes based on real-time system monitoring data. The system monitors CPU load, memory pressure, and job behavior using a feedback-control design. According to the dynamic thresholds and process states, resource allocation is adjusted by suspending or resuming processes to maintain optimal system throughput.

The system includes a GUI (PyQt6-based) to visualize job states, CPU usage, memory usage, and real-time resource allocation decisions.

3. MODULE-WISE BREAKDOWN

3.1 Job Generator (job.py)

- Creates CPU-intensive or memory-intensive processes.
- CPU job → performs an infinite computation loop.
- Memory job → continuously allocates memory blocks.
- Each job runs in a separate Python subprocess.

3.2 Job Information Manager (jobinfo.py)

- Maintains metadata about each launched process.
- Uses psutil to extract:
  - ➢ CPU usage
  - ➢ Memory consumption (RSS)
  - ➢ Execution state (RUNNING / PAUSED / TERMINATED)
- Provides:

➢ snapshot() → returns real-time process statistics.

➢ pause() → suspends process.

➢ resume() → resumes process.

3.3 System Monitor (monitor.py)

Responsibilities:

- Launches new jobs (CPU/MEM).
- Maintains a list of active job objects.
- Collects system-wide metrics:
  - ➢ CPU%
  - ➢ Memory%
- Returns job list + system metrics for the allocator.
- Provides find(pid) to locate and control specific jobs.

3.4 Adaptive Allocator (allocator.py)

Implements the core Adaptive Resource Allocation Algorithm:

Conditions:

1. High Memory Load → Pause the job using the most memory

2. Low CPU Load → Resume a paused job with the smallest memory load

Output:

- A list of actions taken (Paused/Resumed) displayed in the UI.

3.5 Graphical User Interface (ui.py)

- Built using **PyQt6**.

- Displays:
  - ➢ Live CPU percentage
  - ➢ Live memory percentage
  - ➢ Table of jobs (PID, Type, CPU%, Memory)

- Buttons for:
  - ➢ Start CPU job
  - ➢ Start memory job

➢ Pause/Resume selected job

- Auto-refresh every 1000 ms.

3.6 Application Entry Point (main.py)

- Initializes PyQt6 app.
- Creates Monitor, Allocator, and UI instances.
- Launches the event loop.
- Ensures multiprocessing compatibility via spawn mode.

4. Functionalities

- ➔ Dynamic CPU and memory monitoring
- ➔ Real-time adaptive allocation
- ➔ Process suspension and resumption
- ➔ Graphical job and system monitoring
- ➔ Controlled load generation (CPU/MEM jobs)
- ➔ Threshold-based decision-making
- ➔ Feedback-driven resource optimization

5. Technology Used

Programming Language

- Python 3.x

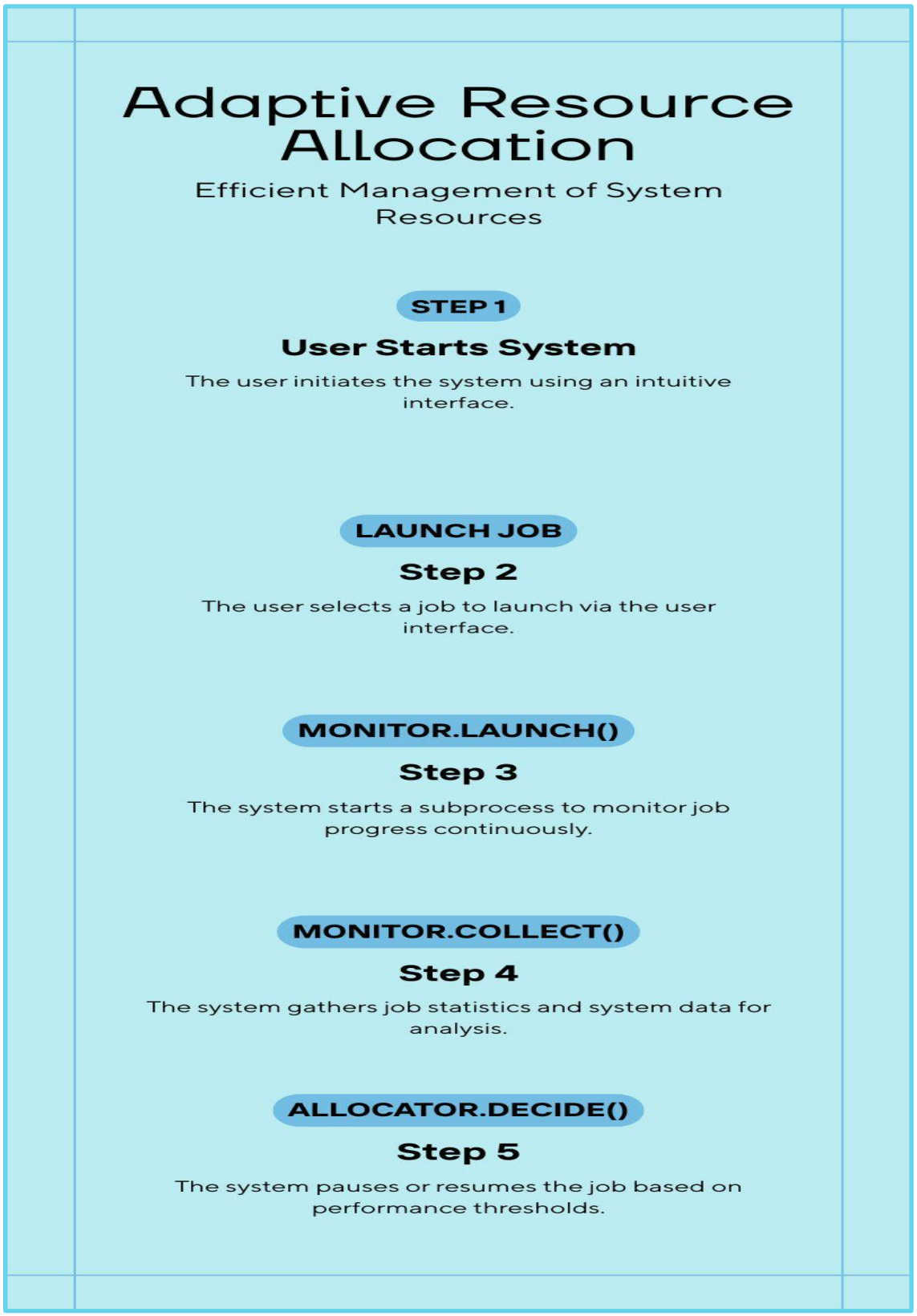
Libraries

- psutil
- PyQt6
- subprocess
- multiprocessing

Tools

- GitHub
- Windows OS environment

6. Flowchart


Adaptive Resource Allocation
Efficient Management of System Resources
STEP 1
User Starts System
The user initiates the system using an intuitive interface.
LAUNCH JOB
Step 2
The user selects a job to launch via the user interface.
MONITOR.LAUNCH()
Step 3
The system starts a subprocess to monitor job progress continuously.
MONITOR.COLLECT()
Step 4
The system gathers job statistics and system data for analysis.
ALLOCATOR.DECIDE()
Step 5
The system pauses or resumes the job based on performance thresholds.

7. Revision Tracking on GitHub

Link: https://github.com/Sangeeth-s01/Adaptive-Resource-Allocation-in-Multiprogramming-Systems

8. Conclusion and Future Scope

Conclusion

This project demonstrates a functioning adaptive resource allocation system capable of monitoring real-time resource usage and adjusting job execution dynamically. By suspending and resuming processes based on CPU and memory thresholds, it improves stability and performance under varying workloads.

Future Scope

- ML-based predictive load balancing
- Priority scheduling models
- I/O monitoring integration
- Cross-platform support

- Real-time performance charts

9. References

- psutil Documentation
- Python Subprocess & Multiprocessing Docs
- PyQt6 Official Docs
- Operating System Concepts — Silberschatz, Galvin & Gagne

10. Appendix A – AI-Generated Breakdown

This project implements an **Adaptive Resource Allocation System** designed for multiprogramming environments where multiple processes compete for CPU and memory resources. The breakdown provides a simplified explanation of how the system works and why each component is necessary.

The system operates using a **feedback-control loop**, where real-time system metrics—such as CPU load and memory usage—are continuously monitored. Based on these observations, the system automatically decides whether to pause or resume specific processes to maintain optimal system performance.

The **Monitor module** collects data about running jobs, including their CPU and memory usage. The **Allocator module** then evaluates this information using threshold-based rules. When memory usage exceeds a high threshold, the system pauses the most memory-intensive running job to relieve pressure. When CPU usage drops below a low threshold, it resumes a paused job with the smallest memory footprint, improving system utilization.

The **Job and JobInfo modules** represent individual processes and provide mechanisms to query or control their execution state. The **GUI module** displays all system metrics and processes in real time, giving users clear visibility and interactive control.

Overall, this breakdown highlights how each component works together to form a dynamic, self-adjusting resource manager that improves system stability, prevents bottlenecks, and adapts smoothly to changing workload conditions.

11. Appendix B – Problem Statement

Develop a system that dynamically adjusts resource allocation among multiple programs to optimize CPU and memory utilization. The solution must continuously monitor system performance and redistribute resources in real time to avoid bottlenecks.

12. Appendix C – Full Solution Code

allocator.py

```
class Allocator:
    def __init__(self, cpu_low=40, mem_high=80):
        self.cpu_low = cpu_low
        self.mem_high = mem_high

    def decide(self, monitor, jobs, sysdata):
        actions = []

        if sysdata["mem"] > self.mem_high:
            run = [j for j in jobs if j["state"] == "RUNNING"]
            if run:
                worst = max(run, key=lambda j: j["mem"])
                job = monitor.find(worst["pid"])
                job.pause()
                actions.append(f"Paused PID {job.pid}")
```

```
        if sysdata["cpu"] < self.cpu_low:
            paused = [j for j in jobs if j["state"] == "PAUSED"]
            if paused:
                best = min(paused, key=lambda j: j["mem"])
                job = monitor.find(best["pid"])
                job.resume()
                actions.append(f"Resumed PID {job.pid}")

        return actions
```

job.py

```
import sys
import time

job_type = sys.argv[1]

if job_type == "CPU":
    x = 0
    while True:
        x = x * 1.01 + 1

elif job_type == "MEM":
    data = []
    while True:
        data.extend([0] * 50000)
        time.sleep(0.05)
```

jobinfo.py

```
import psutil

class Job:
    def __init__(self, pid, job_type):
        self.pid = pid
        self.proc = psutil.Process(pid)
        self.type = job_type
        self.state = "RUNNING"

    def snapshot(self):
        try:
            return {
                "pid": self.pid,
                "type": self.type,
                "cpu": self.proc.cpu_percent(),
                "mem": self.proc.memory_info().rss / (1024 * 1024),
                "state": self.state
            }
        except psutil.NoSuchProcess:
            self.state = "TERM"
            return None

    def pause(self):
        self.proc.suspend()
        self.state = "PAUSED"

    def resume(self):
        self.proc.resume()
```

```
        self.state = "RUNNING"
```

monitor.py

```
import subprocess
import sys
import time
import psutil
from jobinfo import Job

class Monitor:
    def __init__(self):
        self.jobs = []

    def launch(self, job_type):
        p = subprocess.Popen([sys.executable, "job.py", job_type])
        time.sleep(0.3)

        if psutil.pid_exists(p.pid):
            job = Job(p.pid, job_type)
            self.jobs.append(job)
            print(f"Started {job_type} job PID={p.pid}")
        else:
            print("Process failed!")

    def collect(self):
        data = []
        alive = []
```

```
        for job in self.jobs:
            if not psutil.pid_exists(job.pid):
                continue
            data.append(job.snapshot())
            alive.append(job)

        self.jobs = alive

        sys_cpu = psutil.cpu_percent()
        mem = psutil.virtual_memory()
        return data, {"cpu": sys_cpu, "mem": mem.percent}

    def find(self, pid):
        for j in self.jobs:
            if j.pid == pid:
                return j
        return None
```

ui.py

```
from PyQt6.QtWidgets import *
from PyQt6.QtCore import QTimer

class UI(QWidget):
    def __init__(self, monitor, allocator):
        super().__init__()
        self.monitor = monitor
        self.alloc = allocator
```

```
        self.setWindowTitle("Adaptive Resource Allocator (Windows)")
        layout = QVBoxLayout(self)

        self.lbl_sys = QLabel("CPU: --  MEM: --")
        layout.addWidget(self.lbl_sys)

        self.table = QTableWidget(0, 4)
        self.table.setHorizontalHeaderLabels(["PID", "Type", "CPU%",
"Mem(MB)"])
        layout.addWidget(self.table)

        btns = QHBoxLayout()
        self.btn_cpu = QPushButton("Start CPU Job")
        self.btn_mem = QPushButton("Start Memory Job")
        self.btn_toggle = QPushButton("Pause/Resume")
        btns.addWidget(self.btn_cpu)
        btns.addWidget(self.btn_mem)
        btns.addWidget(self.btn_toggle)
        layout.addLayout(btns)

        self.btn_cpu.clicked.connect(lambda: self.monitor.launch("CPU"))
        self.btn_mem.clicked.connect(lambda: self.monitor.launch("MEM"))
        self.btn_toggle.clicked.connect(self.toggle)

        self.timer = QTimer(self)
        self.timer.timeout.connect(self.refresh)
        self.timer.start(1000)

    def refresh(self):
```

```
        jobs, sysd = self.monitor.collect()
        self.lbl_sys.setText(f"CPU: {sysd['cpu']}%   MEM: {sysd['mem']}%")

        acts = self.alloc.decide(self.monitor, jobs, sysd)
        if acts:
            print(" | ".join(acts))

        self.table.setRowCount(len(jobs))
        for i, j in enumerate(jobs):
            self.table.setItem(i, 0, QTableWidgetItem(str(j["pid"])))
            self.table.setItem(i, 1, QTableWidgetItem(j["type"]))
            self.table.setItem(i, 2, QTableWidgetItem(f"{j['cpu']:.1f}"))
            self.table.setItem(i, 3, QTableWidgetItem(f"{j['mem']:.1f}"))

    def toggle(self):
        row = self.table.currentRow()
        if row < 0:
            return

        pid = int(self.table.item(row, 0).text())
        job = self.monitor.find(pid)

        if job.state == "RUNNING":
            job.pause()
        else:
            job.resume()
```

main.py

```
import sys
import multiprocessing as mp
from PyQt6.QtWidgets import QApplication
from monitor import Monitor
from allocator import Allocator
from ui import UI

if __name__ == "__main__":
    mp.set_start_method("spawn")

    app = QApplication(sys.argv)
    monitor = Monitor()
    allocator = Allocator()
    gui = UI(monitor, allocator)
    gui.show()
    sys.exit(app.exec())
```