Hybrid Microkernel

Akshay Khole - 1100226
Anuja Tupe - 1100227
David Lindskog - 1033581
COEN 283 - Operating Systems
November 10, 2014

Preface

In computer science, a **microkernel** is the near-minimum amount of software that can provide the mechanisms needed to implement an operating system. These mechanisms include low-level address space management, thread management, and interprocess communication (I.P.C). The aim of our project is to overcome the issues related to security, extensibility, reliability and flexibility faced by monolithic kernels. microkernels are one way to achieve this since they divide the work between different servers. Failure of one server would not affect the functioning of the other servers hence, proving microkernels to be a more robust solution. However, microkernels have performance issues due to the indefinite message passing via the kernel space to carry out a task. We will try to overcome this shortcoming by switching the servers dynamically from user-space to kernel space based on the load on the server and the permissions provided to them.

Acknowledgement

We would like to thank Professor Ming-Hwa Wang for his guidance, his constant supervision and for providing us with the necessary information regarding the project. We would like to thank the authors of the IEEE papers that helped us with our research.

Table of Contents

Program Flowchart
Program Source Code with Documentation
Input/Output Listing
Other related material

List of Tables and Figures

1. Abstract

In this project proposal, we present a type of kernel architecture that can be used to overcome the problems of traditional kernels that are widely used in most modern operating systems like Linux, Windows, older Mac OS. This architecture, called microkernel, focuses on modularizing each component of the monolithic kernel into independent 'servers' that run in the user space rather than the kernel space. Running these servers in user space limits the permission level each component has.

2. Introduction

Objective

To simulate the implementation of a microkernel and demonstrate how the microkernel architecture can solve certain problems faced by the traditional monolithic kernel architecture.

What is the problem?

Most of the popular and widely used operating systems like Unix, MS-DOS and older versions of Mac OS follow the monolithic kernel architecture. Monolithic kernels are designed in a way that all components of the kernel reside in one single module. These components permanently run in the 'kernel space'. Not all of these components require the privileges that are given to modules running in the kernel space. Because they run in the kernel space, it is possible that they may execute commands that they aren't supposed to. This poses problems of security, reliability, flexibility and extensibility as we shall discuss further. Because of the monolithic nature of older kernels, if one of the components of the kernel crashes, it causes the entire kernel to crash.

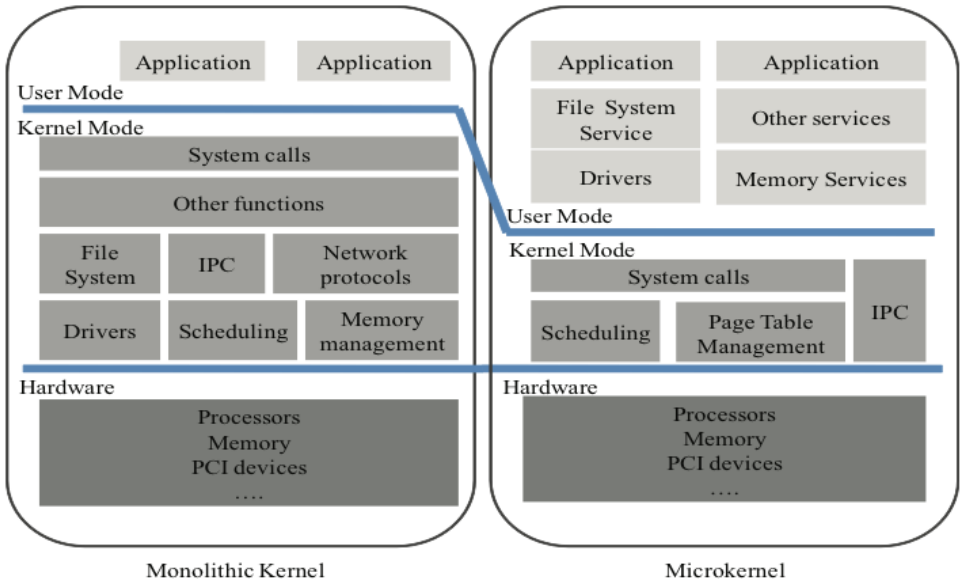


Figure 1. Design Difference between monolithic kernel and microkernel

Why is this project related to this class?

The term 'microkernel' speaks for itself. Every operating system has an integral part without which it cannot function and that part is called the kernel. It is responsible for all the low level operations like communicating with the hardware, managing memory, handling interrupts and managing processes.

Why other approaches are not good?

Layered OS was one of the solutions developed to overcome the problems that were prevalent in operating systems running on the monolithic kernel. This type of OS reduces the security problems in monolithic kernel by layering the OS, however implementing security mechanisms was very pretty difficult due to the constant communication between the adjacent layers. Additionally, code changes in one layer would have numerous effects on the adjacent layers. Hence, this solution was not very helpful.

Why we think our approach is better

Our approach investigates the roots of the problems faced by the traditional monolithic kernels. The fundamental design of the monolithic kernel generates certain problems that can be solved by changing the architecture and making components of the kernel independent of each other.

Statement of the problem

Traditional kernel i.e. monolithic kernel is difficult to extend, maintain and debug due to the nature of its design. We try to solve this problem using microkernel design.

Area or scope of investigation

- We will be simulating the most basic functionalities like memory management, IPC and process management.
- We shall try improving the performance of microkernels by *dynamically* switching servers from the user space to the kernel space, avoiding all the message passing required to carry out a task.
- Future scope - We would like to add the interrupt handling functionality to the project at a later stage.

3. Theoretical basis and literature review

Definition of the problem

Modifying / extending monolithic kernels to suit requirements of modern personal and embedded computers is hard. Microkernel allows us to easily make changes to kernel components to suit requirements of different types of modern computers.

Theoretical background of the problem

For years there have been debates of which architecture should an operating system ideally follow. There is no silver bullet to this problem and one of the solutions is to modify the architecture according to its applications to better perform specific tasks.

Related research to solve this problem

There have been many publications in the area of microkernel architecture to be used on modern computers including smartphones. L4, one of the most popular microkernel is already deployed on billions of devices.

Advantage / disadvantage of those research

Research in this field has opened up the many possibilities of using the kernel for different types of OS running on different types of devices. One clear disadvantage is speed that is compromised to gain the advantages. We try to overcome this disadvantage using our 'hybrid microkernel' approach.

Our solution to solve this problem

The Layered OS approach was not very helpful. Hence, we plan on using microkernels. The philosophy underlying the microkernel is that only absolutely essential core operating system functions should be in the kernel. Other auxiliary components should be in the user space.

Why our solution is better and where your solution different from others

Our solution is better as it makes the operating system more flexible as functionalities can be removed as and when wanted, more extensible as functionalities can be added without affecting the kernel code, more reliable and more secure as all the communication and calls are made via the kernel space based on the permissions assigned to the servers. Since microkernels are modular in the way that since all components are running as independent servers, if a single component crashes, the kernel may still be usable for executing other tasks. Also, a part of our solution tries to improve the performance of the microkernel by switching the servers back and forth between the kernel mode and user mode depending on the current load and security settings.

4. Hypothesis (or goals)

Positive / Negative hypothesis

The microkernel would be more secure, reliable and extensible than monolithic kernel. Microkernels are generally slower than monolithic kernels due to the added IPC calls between kernel components and servers running in user space.

Multiple hypothesis

- The hybrid microkernel aims to increase speed when necessary and when security and reliability can be sacrificed for speed of execution.
- This also gives user the flexibility to switch between modes of execution.

5. Methodology

Generating / Collecting Input Data

Since the activity of our project is distributed amongst various modules, input for the most part will be done through IPC. For overall test cases, we may read commands from a text file, or simply hard code scenarios for simplicity.

Solving the Problem

Language Used

We are designing our project in Java in order to take advantage of it's services such as threads and synchronization. The purpose of the project is to simulate a functioning microkernel. By using java, we will reduce the time we spend building basic services/data structures and increase the time we can spend on bigger ideas.

Algorithm Design

Functionality can be in either kernel space or user space. In a microkernel design, only essential services are kept in kernel space and the rest are in user space. This creates great modularity, security, stability etc. at the cost of speed. As shown in Figure 3 below, to allocate memory in a micro kernel, there are 7 messages that are sent between the kernel and various servers. To reduce the amount of communication between servers and the kernel during high demand circumstances, we want to implement a 'space shifting' algorithm that will allow the kernel to access server functionality directly. Process management is the only module that knows how much process traffic there is. Therefore we will let it decide what servers to put in kernel mode and when. If a process is in kernel mode, the kernel can bypass use of external user functionality and use internal kernel functionality directly (ex. Kernel PMA vs User PMA). By enabling the kernel to skip communication steps, the amount of time needed to get things done is significantly reduced.

Tools Used

We will be using Java to program our project, therefore eclipse will be a tool we use in our development. We also used Python libraries numpy, matplotlib and pyplot to implement some graph generating scripts that illustrate our output more graphically. We also use Git for source code versioning and management.

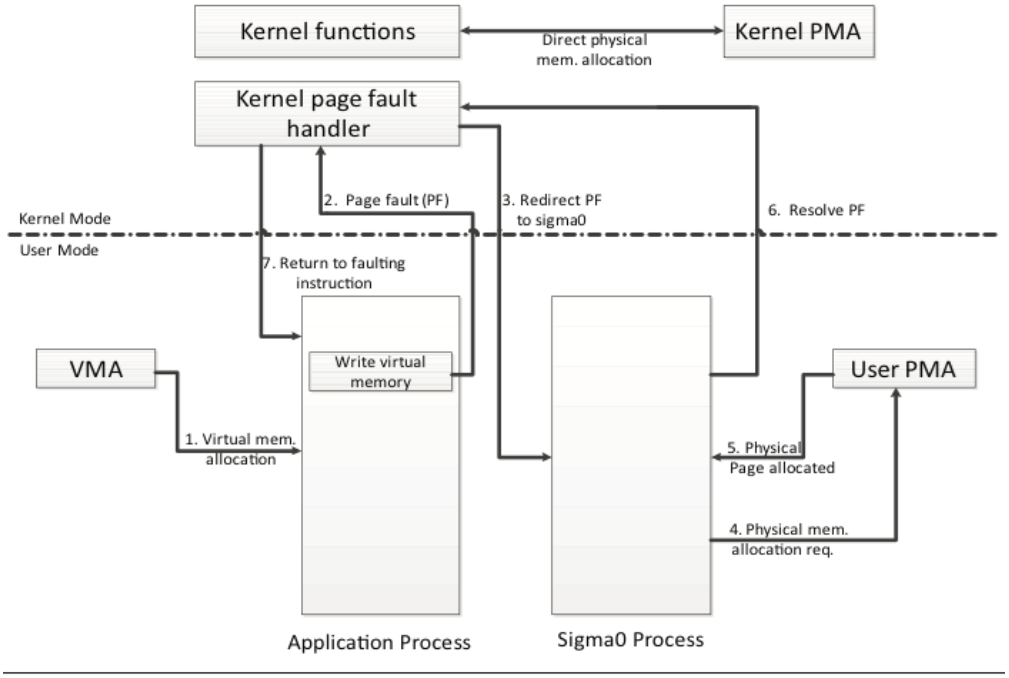


Figure 2. Physical Memory Allocation in Fiasco.OC.

Generating Output

Output is generated as each module communicates and executes. Since we are only simulating a microkernel, we use time to illustrate the transfer of data and messages. To show that each message is being passed or that processing is being done, we have threads wait for random amount of time and display how long tasks are taking. Output is generated in the form of graphs as well. Text files containing process runtime data are created and then read by simple python scripts. These scripts graph each process's start time and duration so it is more visually clear when the kernel shifts servers into kernel space.

Testing Against Hypothesis

Since we are actively displaying how much time tasks are taking, we use this data to compare the speed with and without our space shifting algorithm.

6. Implementation

Code

The project is implemented in Java. The file names are as follows.

1. Main.java
2. User.java
3. Userspace.java
4. Kernel.java
5. SchedulerAdder.java
6. SchedulerPopper.java
7. Process.java
8. ProcessQueue.java
9. Message.java
10. Server.java
11. ServerCrashedException.java
12. ServerPermission.java
13. Sigma.java
14. MemoryManager.java
15. MemoryTree.java
16. Node.java
17. Memory.java
18. Simulation.java
19. FileManager.java
20. Cpu.java
21. Log.java
22. Report.java
23. Graph.java

We have also used python library to display the output of our project in the form of graphs. The file names are as follows.

1. plot_graph_2.py
2. plot_graph_3.py

The python modules create two graphs which display the time taken by every process created by the user. The switch from user mode to kernel mode and back to user mode is visible due to the time variations for every process.

Design Document and Flowchart

Main Algorithm -

1) Main
2) User
3) Create Process
4) Send process to kernel
5) Kernel sends process to scheduler
6) Scheduler schedules process (Explained in the following algoritms)
7) Cpu executes each command of process
8) If in User mode:
    a. If memory command:
        i. Send request message to Sigma0
        ii. Sigma0 sends request to MemoryManager
        iii. MemoryManager reads request type and executes command
9) If in kernel mode
    a. if memory command:
        iv. Retrieve MemoryManager referrence from permissionTable
        v. Directly tell MemoryManager to execute command

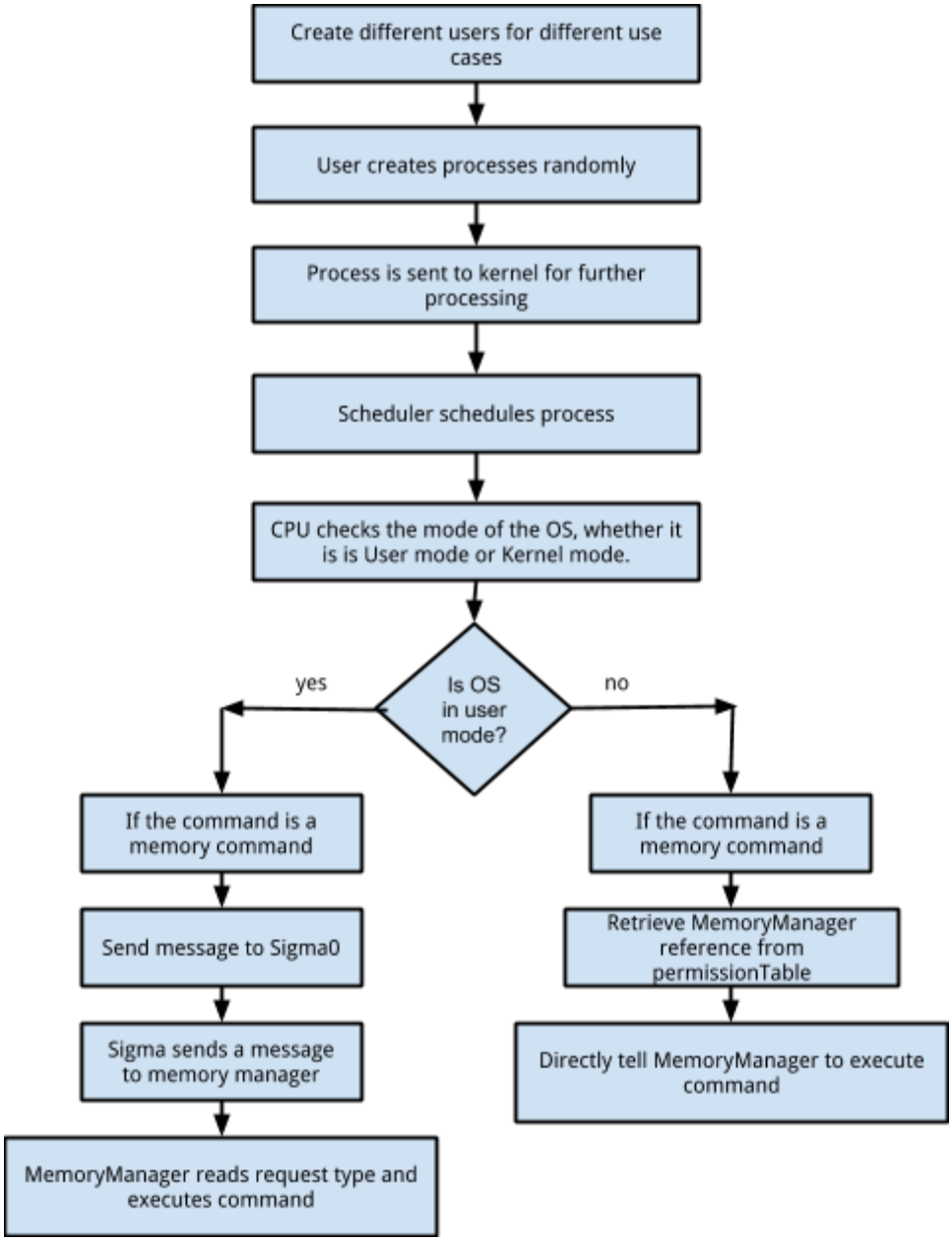


Figure 3. Main Program Flowchart

Scheduler Adder Algorithm -

1) Process is created
2) Process is scheduled using the public schedule method of schedulerAdder
3) Is place there in scheduler queue?
    a) Yes, adds the process to the queue and notifies the scheduler popper waiting for the queue to get processes. Locks the queue so that the scheduler popper doesn't pop a process from the queue when it is trying to add a process.

b) No, prints that the queue has no more space and tells the kernel no space and waits till the queue has space
c) releases the lock.

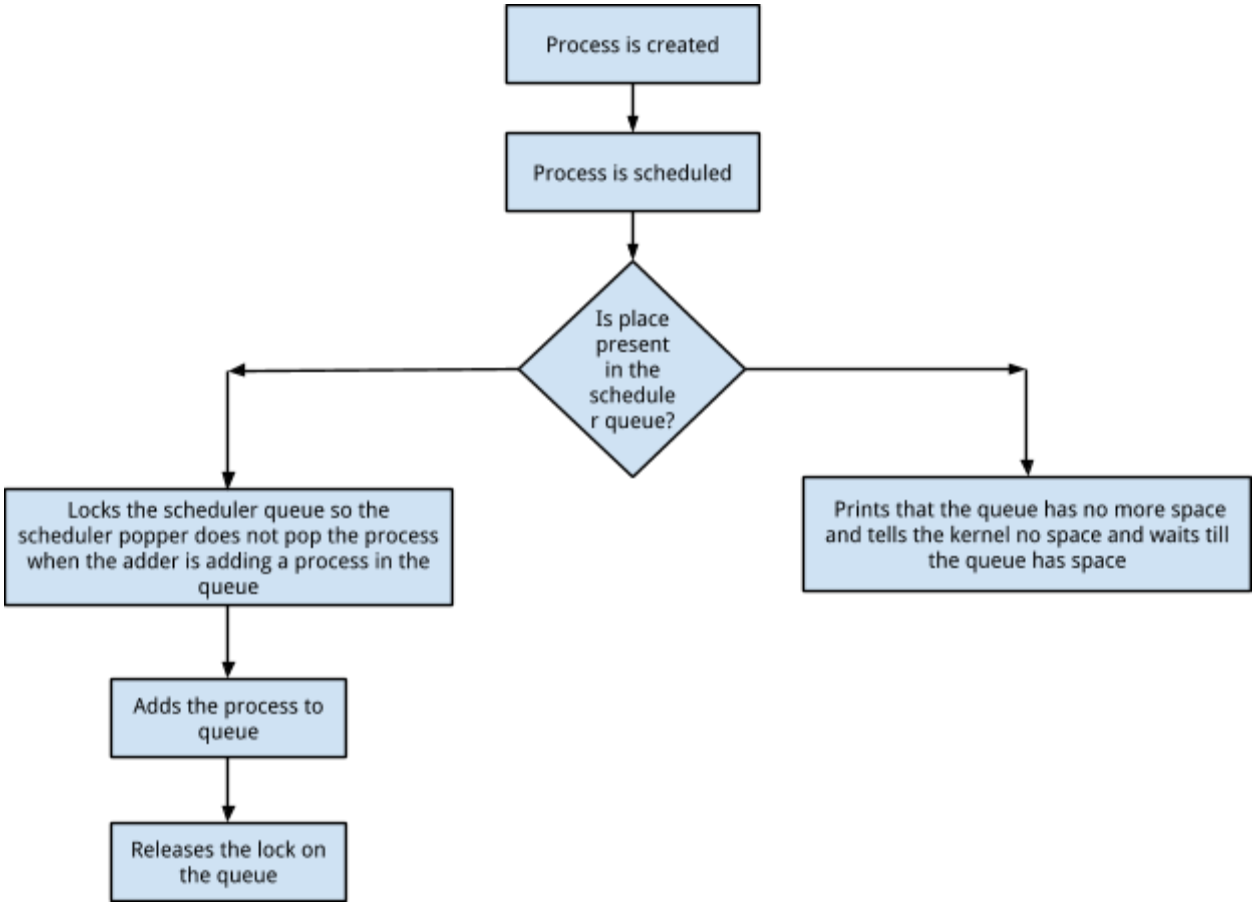


Figure 4. Scheduler Adder Flowchart

Scheduler Popper Algorithm -

1) Decides which OS mode is being used - user mode or kernel mode based on the number of process which are present in the queue for execution
    a) If process queue size is greater than threshold mentioned and os mode is not equal to kernel mode, switch the os mode to be kernel mode.
    b) If process queue size is smaller than the threshold mentioned and the os mode is not equal to user mode, switch the os mode to be user mode.
2) Is scheduler queue empty?
    a) If Yes, scheduler popper waits on the queue till it gets at least one element
    b) If No, pops the process from the top of the queue and notifies the scheduler adder waiting on the queue to get space. Locks the queue so that the scheduler adder doesn't add a process to the queue when it is trying to pop a process.

i) Gives this popped queue for execution to the CPU.

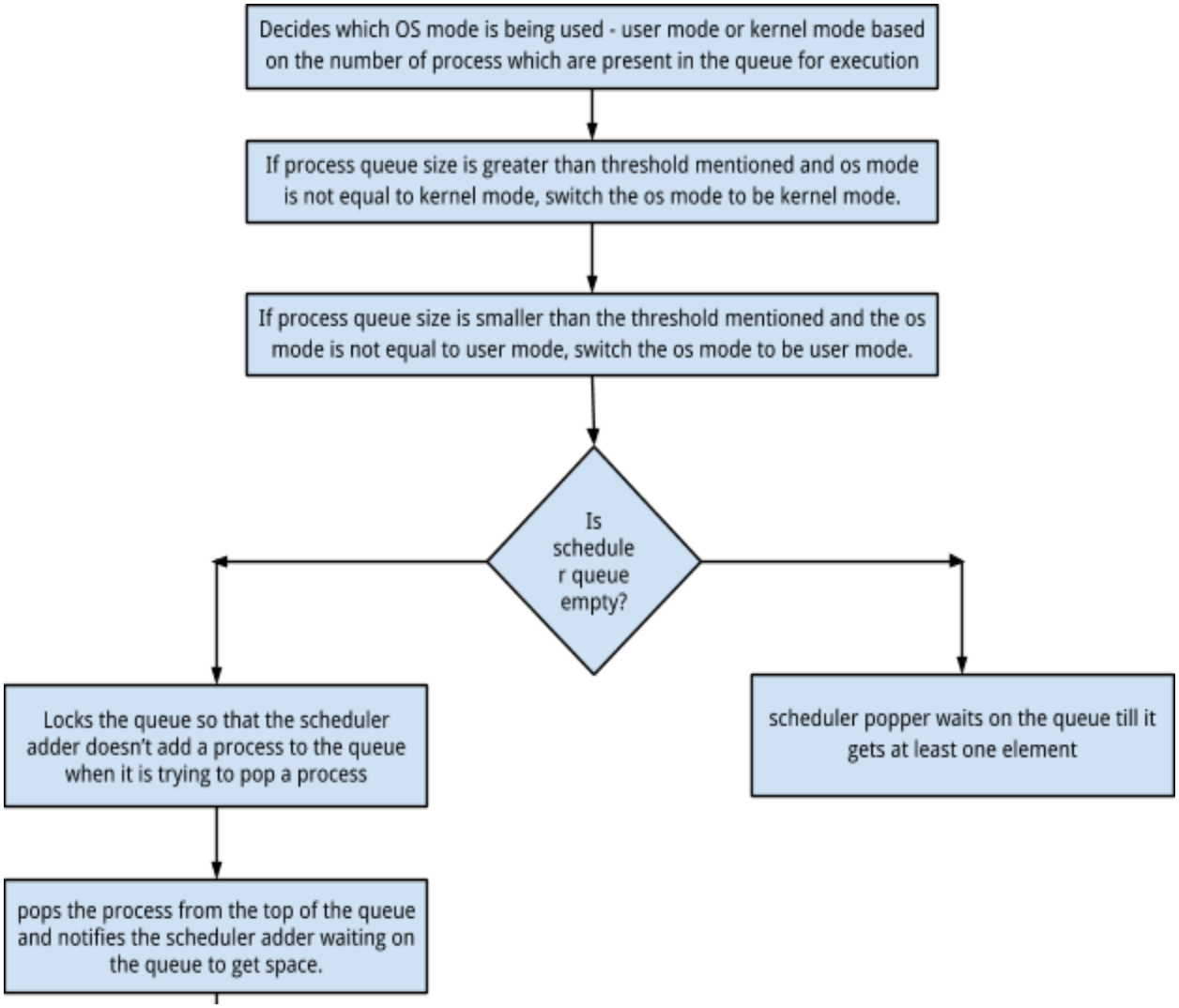


Figure 5. Scheduler Popper Flowchart

Memory Allocation Algorithm -

Memory Allocation in our project is implemented as a Buddy System. We have a slab size for the system. Any requests which are bigger than the slab size are allocated in the buddy system else they are allocated in the slab system which is a pre split tree. The slab size is decided by the request sizes which are very frequent.

1) Process comes in with a memory requirement and PID
2) Process checks the type of command
3) If command is Alloc, it checks if that PID is already holding some memory
    a) If Yes, Displays a message "Memory is already allocated for PID"
    b) If No, Memory Manager checks if the the request for memory is less than or equal to the slab size
        i) If the request is smaller or equal to slab size and there is place in the slab tree, the request is fulfilled.
        ii) If the request is bigger than the slab size and there is place in the buddy system, the request is fulfilled.
4) If command is Free, it checks if the PID is holding memory in the system
    a) If yes, it frees the memory held by that PID
    b) If No, it displays a warning message "PID not found. Could not free memory for PID"
5) If command is Realloc, it checks if the PID is holding memory in the system.
    a) If yes, it follows step 4) followed by step 3)
    b) If No, it displays a warning message "PID not found. Could not realloc memory for PID"

7. Data Analysis and Discussion

Output Generation

For simplicity purposes, we have presented three test cases. All the outputs are displayed on the console. We have also maintained a detailed report.txt file which records all the details of the kernel, scheduler, CPU, processes and servers. Additionally, we have also displayed the performance variation for the processes when the OS mode switch comes into action. This is displayed in the form of graphs which are implemented using Python libraries - matlibplot.

Simulation of basic microkernel - This test case is to display the basic working or microkernel and the improvements introduced by us. The microkernel switches the OS mode to kernel mode depending on the traffic in the scheduler. If the number of processes go beyond the threshold of the servers, the OS mode is changed to kernel mode which is faster than using the user mode as the communication part between the servers is discarded.

Simulation of security in microkernel - This test case displays the strong security in the microkernel. We make a server call the command which it does not have a permission for and the kernel denies the execution of the command as the server does not have the rights to run that command.

Simulation of reliability - This test case displays the reliability of microkernels. We make a server crash. Any calls to the crashed server are not completed. However, the rest of the microkernel continues to work properly. We crash the memory manager and the file manager runs perfectly even when the memory manager has crashed.

Output Analysis

From the output following observations can be made -

1. When any server is in user mode, the number of steps for executing are increased since a lot of messages need to be passed between kernel and the server and server and its internal components before any work is actually done. This adds overhead.
2. When any server is in kernel mode, kernel is in full control and it can directly invoke servers' components to which it has references. This eliminates the overhead of message passing and makes code run faster.
3. Because the kernel also needs to have some understanding for directly operating servers when they are in kernel mode, additional code needs to be saved within the kernel. This might increase the size of the kernel.
4. After generating output, we realize that it may happen that the queue gets filled much faster than processes get executed so it is possible that even though a server has a threshold value of, say 5, the 2nd process itself may start executing in the kernel mode. This is because even though only the 2nd process is currently running, the process queue may have number of processes well over the threshold value (5).

Compare output against hypothesis

Our hypothesis states that the time required for the execution of the process reduces when a particular module switches from the user mode to the kernel mode. Thus, increasing the performance of the OS by eliminating the overhead caused by the intermediate messaging passing steps via the kernel. Our hypothesis stands true. We have depicted this using a graph. We run 20 processes one after the other. The starting process stays in the user mode and takes a considerable amount of time to execute. However, as more and more processes start coming in the scheduler, a mode switch takes place. When the module is in kernel mode, the processes run faster than what they would run in user mode. Additionally, for the last few processes, the processes will run in user mode as there won't be any more processes coming in.

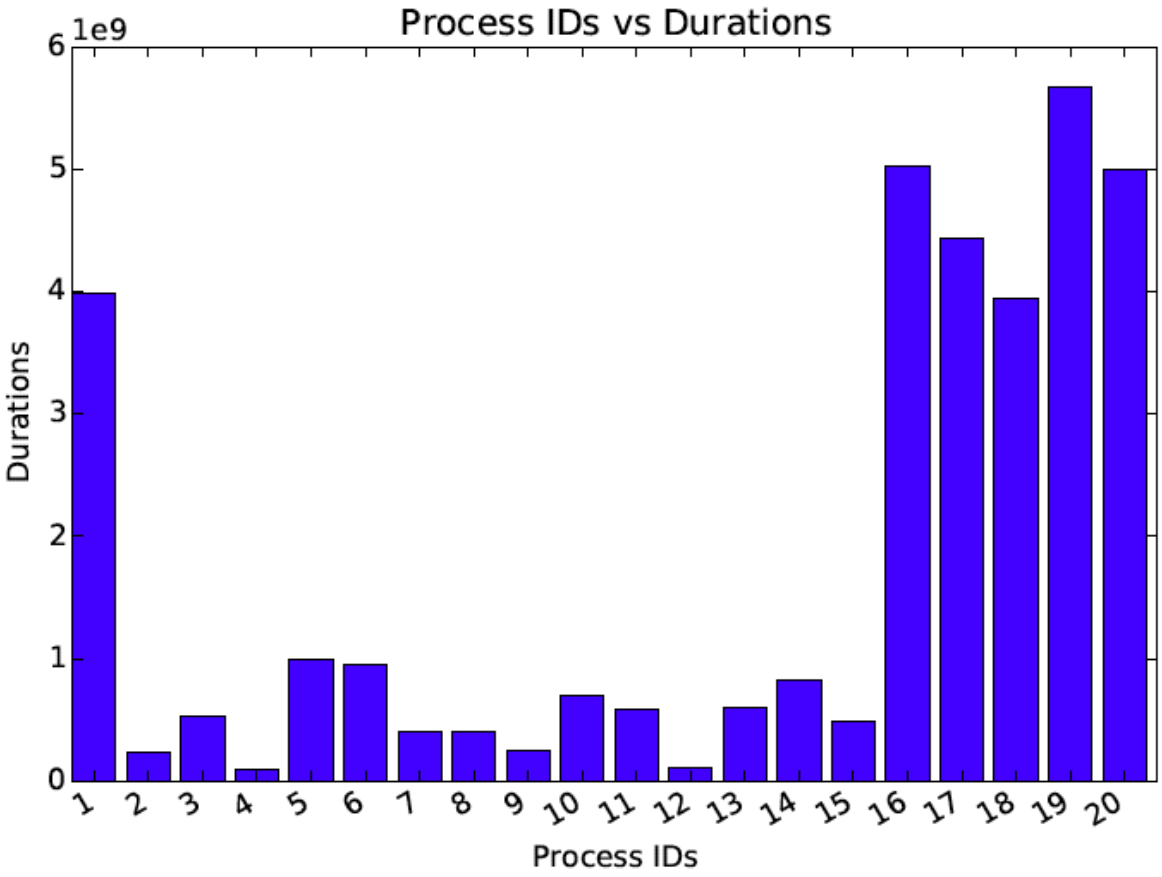


Figure 6. Graph of process id versus their execution time

Figure 6 gives us the following findings:

Average time taken for executing in user mode for PIDs 1, 16, 17, 18, 19 and 20:

(4 + 5 + 4.5 + 4 + 5.7 + 5) / 6 = 4.7

If a server executes in kernel mode all the time, it takes about 4.7 time units for execution.

Average time taken for executing in kernel mode for PIDs 2 - 15:
(0.3 + 0.5 + 0.2 + 1 + 0.9 + 0.4 + 0.4 + 0.3 + 0.7 + 0.6 + 0.2 + 0.6 + 0.8 + 0.5) / 14 = 0.52

While these values are highly ideal and represent simulation conditions, we expect actual findings to be similar.

8. Conclusions and Recommendations

Summary and Conclusions

The project demonstrates the merits of using a microkernel. We have explored the security, reliability and the switch mode aspects of the microkernel. The demonstrations given by us indicate that using a microkernel is a good choice when you have tight security constraints and at the same time you need a good performance from the OS. The switch between the modes i.e. the switch from user mode and kernel mode back and forth show how the microkernel may solve the performance issues faced my monolithic kernels.

Recommendations for future studies

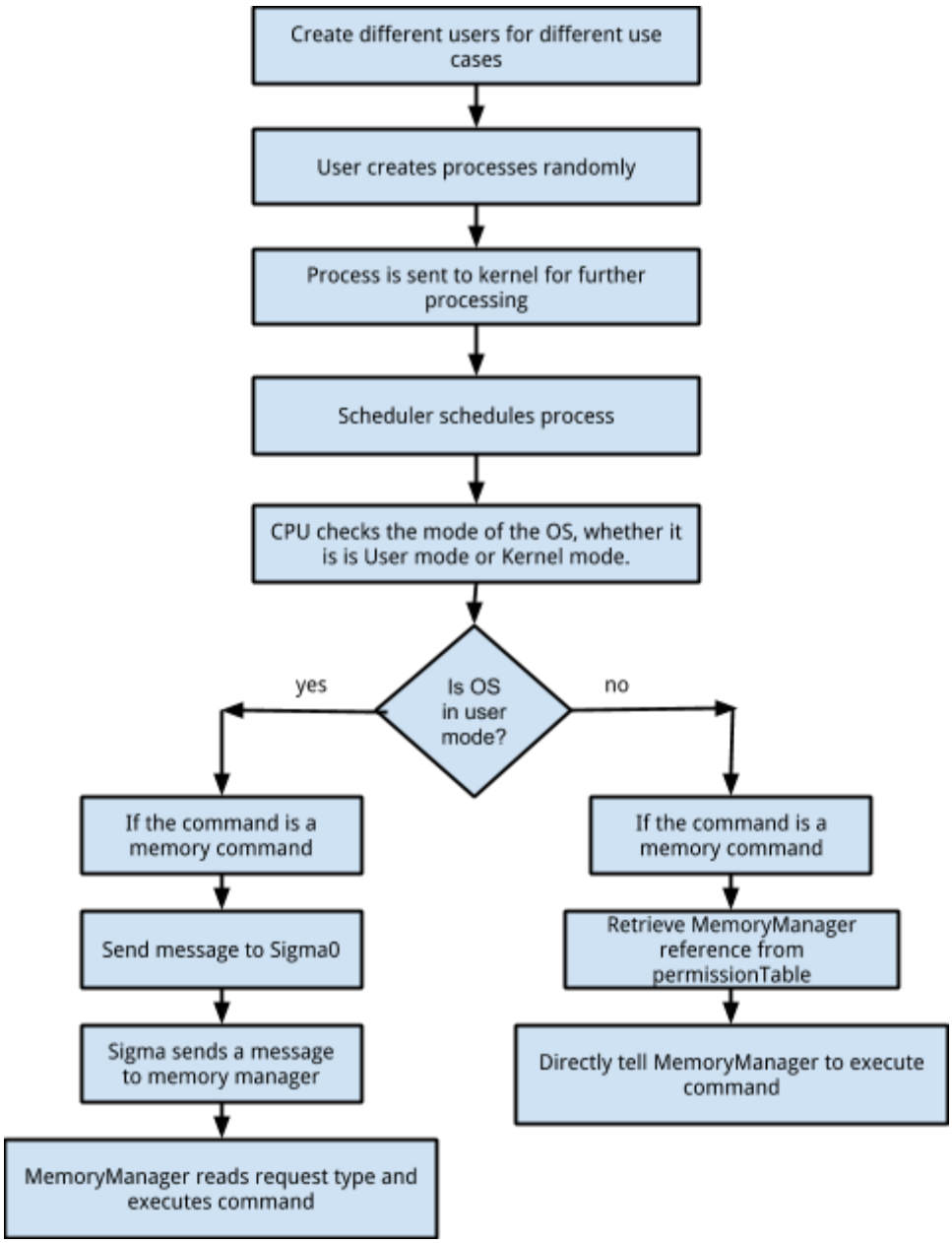
- HelenOS http://www.helenos.org/ - Our furture studies include studying the HelenOS. HelenOS will become a complete and usable modern operating system, offering room for experimenting and research. HelenOS uses its own microkernel written from scratch and supports SMP, multitasking and multithreading on both 32-bit and 64-bit, little-endian and big-endian processor architectures. HelenOS is being developed mostly by faculty members, and former and contemporary students of Faculty of Mathematics and Physics at Charles University in Prague. Nonetheless, the project is open for everyone, so we also have developers with different backgrounds from various places around the world. The source code is open and available under the BSD license.
- Future project goals include implementation of device drivers.

9. Bibliography

- A Practical Look at Micro-Kernels and Virtual Machine Monitors - François Armand, Michel Gien, Member, IEEE
- Research on Microkernel Technology - Wang Chengjun, Department of Computer Science and Technology, WeiFang University
- A Scalable Physical Memory Allocation Scheme For L4 Microkernel - Chen Tian, Daniel Waddington, Jilong Kuang
- Design of Embedded OS Micro-kernel Experiment Series on ARM - Bo Qu
- Ray - a secure micro kernel architecture - Nurnberger, S. ; CASED-Center for Adv. Security Res., Darmstadt, Germany ; Feller, T. ; Huss, S.A.

10. Appendices

Program Flowchart



Program Source Code with Documentation

Scheduler Adder -

```
package hybrid_mircokernel;

import java.util.ArrayList;
import java.util.Iterator;

public class SchedulerAdder extends Thread {
	private ArrayList<Process> process_table;
	public int QUEUE_SIZE;
	public static final int RANGE = 1000;
	public static final int MIN_NUMBER = 0;

	/*
	 * Constructor
	 */
	public SchedulerAdder(ArrayList<Process> process_table, int size) {
		this.process_table = process_table;
		this.QUEUE_SIZE = size;
	}


	/*
	 * Adds the incoming process to the process table i.e the scheduler queue
	 */
	public boolean schedule(Process p) {
		Log.i("PROCESS TABLE SIZE: ", String.valueOf(process_table.size()));
		if(process_table.size() == QUEUE_SIZE) {
			synchronized(process_table) {
				try {
					Log.i("QUEUE IS FULL..");
					process_table.wait();
				} catch(InterruptedException e) {
					Log.e("SchedulerAdder.schedule()", e);
				}
			}
			return false;
		}
```

```
		synchronized(process_table) {
			p.setBurstTime((long)(Math.random() * RANGE) + MIN_NUMBER);
			process_table.add(p);
			p.appendToLog("Process added to queue");
			process_table.notify();
		}
		return true;
	}
}
```

Scheduler Popper -

```
package hybrid_mircokernel;

import java.util.ArrayList;
import java.util.Iterator;

public class SchedulerPopper extends Thread {
	private ArrayList<Process> process_table;
	public int QUEUE_SIZE;
	boolean running = false;

	/*
	 * Constructor
	 */
	public SchedulerPopper(ArrayList<Process> process_table, int size) {
		this.process_table = process_table;
		this.QUEUE_SIZE = size;
	}

	/*
	 * Toggles the OS mode between kernel mode and User mode based on the traffic in the queue
	 */
	private void setOSMode() {
		Iterator<ServerPermission> iterator = Kernel.permissionTable.values().iterator();
		while (iterator.hasNext()) {
			ServerPermission temp = iterator.next();
			if(process_table.size() >= temp.getThreshold() && temp.getMode() !=
ServerPermission.MODE_KERNEL) {
				temp.setMode(ServerPermission.MODE_KERNEL);
				Log.i("Changed to Kernel Mode");
```

```
                        } else if(process_table.size() < temp.getThreshold() && temp.getMode() !=
ServerPermission.MODE_USER){
                                temp.setMode(ServerPermission.MODE_USER);
                                Log.i("Changed to User Mode");
                        }
                        System.out.println("NAME: " + temp.getName() + " MODE: " +
temp.getMode());
                }
        }

        /*
         * Pops the next process from the head of the queue for execution
         */
        private void popProcess() {
                Process activeProcess;
                setOSMode();
                while(process_table.isEmpty()) {
                        synchronized(process_table) {
                                try {
                                        Log.i("QUEUE IS EMPTY..");
                                        process_table.wait();
                                } catch(InterruptedException e) {
                                        Log.e(e);
                                }
                        }
                }
                synchronized (process_table) {
        process_table.notifyAll();
        activeProcess = process_table.get(0);
        process_table.remove(0);
        activeProcess.appendToLog("Process popped from queue and given to CPU");
    }
                if (activeProcess != null) {
                        Cpu.execute(activeProcess);
                }
        }

        public void run() {
                running = true;
    while (running) {
        popProcess();
    }
  }
```

```
	public void shutdown() {
		running = false;
	}
}
```

CPU -

```
package hybrid_mircokernel;

import java.util.Random;

public class Cpu extends Thread {

	/* Constructor */

	private Cpu() {}

	/*
	 * Method to execute the processes sent by the scheduler.
	 * Executes the requested commands by the processes.
	 */

	public static void execute(Process p) {

		if (Simulation.type == Simulation.SECURITY) {
			simulateSecurity(p);
		}
		else if (Simulation.type == Simulation.CRASH) {
			simulateReliability(p);
		} else {


			long burstTime = p.getBurstTime();
			p.appendToLog("Starting to execute process with burst time: "
					+ burstTime);
			p.setProcessExecutionStartTime();

			StringBuffer report = new StringBuffer("Process: " + p.getPid() + "\n");

			try {
				sleep(burstTime);
```

```
                    if (Kernel.permissionTable.get("memoryManager").getMode() ==
ServerPermission.MODE_USER) {

                        report.append("Mode: USER\n");
                        p.appendToLog("==========>>> Running " + p.getPid() + "
in User mode.");

                        // We check if this server has the permission to execute
                        // ALLOC
                        UserSpace.getSigma0().receive(
                                new Message(p, Message.CODE_ALLOC));

                        for (int i = 0; i < 5; i++) {
                            executeRandomCommand(p,
ServerPermission.MODE_USER);
                        }

                        UserSpace.getSigma0().receive(
                                new Message(p, Message.CODE_FREE));

                    } else if (Kernel.permissionTable.get("memoryManager")
                            .getMode() == ServerPermission.MODE_KERNEL) {

                        report.append("Mode: KERNEL\n");
                        p.appendToLog("==========>>> Running " + p.getPid() + "
in Kernel mode.");

                        MemoryManager memoryManager = (MemoryManager)
Kernel.permissionTable
                                .get("memoryManager").getReference();

                        memoryManager.receive(new Message(p,
Message.CODE_ALLOC));

                        for (int i = 0; i < 5; i++) {
                            executeRandomCommand(p,
ServerPermission.MODE_KERNEL);
                        }

                        memoryManager.receive(new Message(p,
Message.CODE_FREE));
```

```
                        }
                } catch (InterruptedException e) {
                        Log.e("Cpu.execute()", e);
                }
                p.appendToLog("Finished executing process.");
                p.setProcessExecutionEndTime();
                Graph.printValuesForGraph(p);

                report.append("Start: " + p.getExecutionStartTime() + "\n");
                report.append("End: " + p.getExecutionEndTime() + "\n");
                report.append("Duration: " + p.processExecutionDuration() + "\n");

report.append("\n=========================================================\n\n");
                Report.write(report.toString());

                p.appendToLog("Process executed for "
                                + p.processExecutionDuration() + " nsec");


p.appendToLog("\n=====================================================\n");
                }
        }

        /* Method to randomly select the memory commands to be executed by the processes */

        private static void executeRandomCommand(Process p, int mode) {
                int randomNo = (new Random()).nextInt(100);
                if (randomNo % 2 == 0) {
                        p.setNewMemoryRequirements();
                        if (mode == ServerPermission.MODE_USER) {
                                UserSpace.getSigma0().receive(
                                                new Message(p, Message.CODE_REALLOC));
                        } else {
                                ((MemoryManager) Kernel.permissionTable.get("memoryManager")
                                                .getReference()).receive(new Message(p,
                                                Message.CODE_REALLOC));
                        }
                }
        }

        /* Method to randomly select the file commands to be executed by the processes */

        private static void executeRandomFileCommand(Process p, FileManager fm) {
```

```
		int randomNo = (new Random()).nextInt(100);
		if (randomNo % 2 == 0) {
			p.setNewMemoryRequirements();
			if (Kernel.permissionTable.get("memoryManager").getMode() ==
ServerPermission.MODE_USER) {
				UserSpace.getSigma0().receive(
						new Message(p, Message.CODE_REALLOC));
			} else {
				((MemoryManager) Kernel.permissionTable.get("memoryManager")
						.getReference()).receive(new Message(p,
						Message.CODE_REALLOC));
			}
		}
		else {
			if (Kernel.permissionTable.get("fileManager").getMode() ==
ServerPermission.MODE_USER) {
				fm.receive(new Message(p, Message.CODE_CREATE_FILE));
			} else {
				((FileManager) Kernel.permissionTable.get("fileManager")
						.getReference()).receive(new Message(p,
Message.CODE_CREATE_FILE));
			}
		}
	}

	/* Method to simulate the security aspect of microkernel */

	private static void simulateSecurity(Process p) {
		FileManager fileManager = new FileManager();

		int[] fileCommands = { Message.CODE_CREATE_FILE, Message.CODE_FREE,
Message.CODE_DELETE_FILE };

		for (int i = 0; i < fileCommands.length; i++) {
			if (Kernel.permissionTable.get("fileManager").hasPermission(
				fileCommands[i])) {
				fileManager.receive(new Message(p, fileCommands[i]));
			} else {
				Log.e("***EXCEPTION: File manager server tried to execute a FREE
MEMORY command ("
						+ fileCommands[i]
						+ ")"  + "\nbut was not executed as it does not have the
necessary permissions");
```

```
                }
            }
        }

        /* Method to simulate the reliability aspect of microkernel */

        private static void simulateReliability(Process p) {
            FileManager fileManager = new FileManager();

            for (int i = 0; i < 20; i++) {
                executeRandomFileCommand(p, fileManager);
            }

        }

        /* Exception handler for server crashes */

        public static void catchException(ServerCrashedException e) {
            if (e.getCode() == ServerCrashedException.CODE_CRASH) {
                Log.e("File manager has crashed during process " + e.getProcess().getPid());
            } else if (e.getCode() == ServerCrashedException.CODE_TIMEOUT) {
                Log.e("File manager is not responding.");
            }
        }
}
```

ServerPermission -

```
package hybrid_mircokernel;

public class ServerPermission {
        public static final int MODE_KERNEL = 0;
        public static final int MODE_USER = 1;
        private static int processCount;
        private boolean[] permissions;

        private int mode;
        private String name;
        private Object reference;
        private int threshold;
```

```
/* Constructor */

public ServerPermission(String name, Object reference, int threshold) {
	mode = ServerPermission.MODE_USER;
	this.name = name;
	this.reference = reference;
	this.threshold = threshold;
}


/* getters/setters */

public int getMode() {
	return mode;
}

public void setMode(int mode) {
	this.mode = mode;
}

public String getName() {
	return name;
}

public void setName(String name) {
	this.name = name;
}

protected Object getReference() {
	return reference;
}

public void setReference(Object reference) {
	this.reference = reference;
}

public void setThreshold(int threshold) {
	this.threshold = threshold;
}
```

```
	public int getThreshold() {
		return threshold;
	}

	public void upCount() {
		processCount++;
	}

	public void downCount() {
		processCount--;
	}

	protected void setPermissions(boolean[] permissions) {
		this.permissions = permissions;
	}

	public boolean hasPermission(int code) {
		return permissions[code];
	}
}
```

Kernel -

```
package hybrid_mircokernel;

import java.util.ArrayList;
import java.util.HashMap;

public class Kernel extends Thread {
	private boolean running;
	private SchedulerAdder schedulerAdder;
	private SchedulerPopper schedulerPopper;
	private static final int QUEUE_SIZE = 10;
	public static HashMap<String, ServerPermission> permissionTable;
	private ArrayList<Process> process_table = new ArrayList<Process>();
	private Object state;

	/* constructors */
```

```
public Kernel() {}

/* instance methods */

/* Boots the kernel */

public void boot() {
	schedulerAdder = new SchedulerAdder(process_table, QUEUE_SIZE);
	schedulerPopper = new SchedulerPopper(process_table, QUEUE_SIZE);
	schedulerPopper.start();
	permissionTable = new HashMap<String, ServerPermission>();
	state = new Object();
	UserSpace.create();
	addToPermissionTable(UserSpace.getServers());
	running = true;
	try {
		this.start();
	} catch (Exception e) {
		Log.e(e);
	}
}

/* Shuts down the kernel */

public void shutdown() {
	UserSpace.shutDownServers();
	schedulerPopper.shutdown();
	try {
		schedulerPopper.join();
	} catch (InterruptedException e) {
		e.printStackTrace();
	}
	Report.close();
	running = false;
	wakeup();
}

public void run() {
	while (running) {
		synchronized (state) {
			try {
				state.wait();
			} catch (InterruptedException e) {
```

```
                                Log.e("Kernel.run()", e);
                        }
                }
        }
}

private void wakeup() {
        synchronized (state) {
                state.notify();
        }
}

public boolean receive(Message message) {
        wakeup();
        if (message == null) {
                Log.e("Message was null.");
                return false;
        }
        boolean status = schedulerAdder.schedule(message.getProcess());
        return status;
}


/* Adds a server permission to the server permission table */

protected void addToPermissionTable(String name, Object reference, int threshold) {
        ServerPermission s = new ServerPermission(name, reference, threshold);
        Kernel.permissionTable.put(name, s);
}

/* Adds a server permission to the server permission table */

private void addToPermissionTable(ServerPermission[] servers) {
        int i = 0;
        ServerPermission s = servers[i];
        boolean[] permissions = {false, false, false, false, false, false};
        while (s != null) {
                if (servers[i].getName().equals(FileManager.NAME)) {
                        permissions[Message.CODE_CREATE_FILE] = true;
                        permissions[Message.CODE_DELETE_FILE] = true;
                        permissions[Message.CODE_FREE] = false;
                        permissions[Message.CODE_ALLOC] = false;
                        permissions[Message.CODE_REALLOC] = false;
```

```
                }
                else if (servers[i].getName().equals(MemoryManager.NAME)) {
                        permissions[Message.CODE_ALLOC] = true;
                        permissions[Message.CODE_REALLOC] = true;
                        permissions[Message.CODE_FREE] = true;
                        permissions[Message.CODE_CREATE_FILE] = false;
                        permissions[Message.CODE_DELETE_FILE] = false;
                }
                else if (servers[i].getName().equals(Sigma.NAME)) {
                        // none?
                }
                s.setPermissions(permissions);
                Kernel.permissionTable.put(s.getName(), s);
                i++;
                s = servers[i];
            }
        }

        /* Removes a server permission to the server permission table */

        protected ServerPermission removeFromPermissionTable(String name) {
            return permissionTable.remove(name);
        }
}
```

Input/Output Listing

```
Choose from the following test cases:
(1) Simulate Microkernel Security
(2) Simulate reliability
(3) Hybrid use case
(0) Quit

 1
INFO:: QUEUE IS EMPTY..


INFO:: PROCESS TABLE SIZE: :: 0
INFO:: Creating a new file for process 1
ERROR:: ***EXCEPTION: File manager server tried to execute a FREE MEMORY command (3)
```

```
but was not executed as it does not have the necessary permissions
INFO:: Deleting a file requested by process 1
NAME: fileManager MODE: 1
NAME: memoryManager MODE: 1
NAME: sigma0 MODE: 1
INFO:: QUEUE IS EMPTY..



Choose from the following test cases:
(1) Simulate Microkernel Security
(2) Simulate reliability
(3) Hybrid use case
(0) Quit

 2
INFO:: QUEUE IS EMPTY..


INFO:: PROCESS TABLE SIZE: :: 0
INFO:: Memory reallocated for process 1
ERROR:: File manager has crashed!
ERROR:: File manager has crashed during process 1
ERROR:: File manager is not responding.
ERROR:: File manager is not responding.
INFO:: Memory reallocated for process 1
ERROR:: File manager is not responding.
INFO:: Memory reallocated for process 1
INFO:: Memory reallocated for process 1
INFO:: Memory reallocated for process 1
INFO:: Memory reallocated for process 1
ERROR:: File manager is not responding.
ERROR:: File manager is not responding.

Choose from the following test cases:
(1) Simulate Microkernel Security
(2) Simulate reliability
(3) Hybrid use case
(0) Quit

 3
INFO:: QUEUE IS EMPTY..
```

```
INFO:: PROCESS TABLE SIZE: :: 0

INFO:: PROCESS TABLE SIZE: :: 0

INFO:: PROCESS TABLE SIZE: :: 1

INFO:: Memory allocated for process 1
INFO:: PROCESS TABLE SIZE: :: 2

INFO:: PROCESS TABLE SIZE: :: 3

INFO:: Memory reallocated for process 1
INFO:: PROCESS TABLE SIZE: :: 4

INFO:: PROCESS TABLE SIZE: :: 5

INFO:: Memory reallocated for process 1
INFO:: PROCESS TABLE SIZE: :: 6

INFO:: PROCESS TABLE SIZE: :: 7

INFO:: Memory with PID '1' free'd.

INFO:: Changed to Kernel Mode
NAME: fileManager MODE: 0
INFO:: Changed to Kernel Mode
NAME: memoryManager MODE: 0
INFO:: Changed to Kernel Mode
NAME: sigma0 MODE: 0
INFO:: PROCESS TABLE SIZE: :: 7

INFO:: Memory allocated for process 2
```

```
INFO:: Memory reallocated for process 2
INFO:: Memory reallocated for process 2
INFO:: Memory reallocated for process 2
INFO:: Memory reallocated for process 2
INFO:: Memory with PID '2' free'd.

NAME: fileManager MODE: 0
NAME: memoryManager MODE: 0
NAME: sigma0 MODE: 0
INFO:: PROCESS TABLE SIZE: :: 7


INFO:: Memory allocated for process 3
INFO:: Memory reallocated for process 3
INFO:: Memory with PID '3' free'd.

NAME: fileManager MODE: 0
NAME: memoryManager MODE: 0
NAME: sigma0 MODE: 0
INFO:: Memory allocated for process 4
INFO:: Memory reallocated for process 4
INFO:: Memory with PID '4' free'd.

NAME: fileManager MODE: 0
NAME: memoryManager MODE: 0
NAME: sigma0 MODE: 0
INFO:: PROCESS TABLE SIZE: :: 6


INFO:: Memory allocated for process 5
INFO:: Memory reallocated for process 5
INFO:: Memory reallocated for process 5
INFO:: Memory with PID '5' free'd.

NAME: fileManager MODE: 0
NAME: memoryManager MODE: 0
NAME: sigma0 MODE: 0
INFO:: PROCESS TABLE SIZE: :: 6


INFO:: PROCESS TABLE SIZE: :: 7
```

```
INFO:: PROCESS TABLE SIZE: :: 8


INFO:: Memory allocated for process 6
INFO:: Memory reallocated for process 6
INFO:: Memory reallocated for process 6
INFO:: Memory reallocated for process 6
INFO:: Memory reallocated for process 6
INFO:: Memory with PID '6' free'd.

NAME: fileManager MODE: 0
NAME: memoryManager MODE: 0
NAME: sigma0 MODE: 0
INFO:: Memory allocated for process 7
INFO:: Memory reallocated for process 7
INFO:: Memory reallocated for process 7
INFO:: Memory with PID '7' free'd.

NAME: fileManager MODE: 0
NAME: memoryManager MODE: 0
NAME: sigma0 MODE: 0
INFO:: PROCESS TABLE SIZE: :: 7


INFO:: Memory allocated for process 8
INFO:: Memory reallocated for process 8
INFO:: Memory reallocated for process 8
INFO:: Memory with PID '8' free'd.

NAME: fileManager MODE: 0
NAME: memoryManager MODE: 0
NAME: sigma0 MODE: 0
INFO:: Memory allocated for process 9
INFO:: Memory reallocated for process 9
INFO:: Memory reallocated for process 9
INFO:: Memory with PID '9' free'd.

NAME: fileManager MODE: 0
NAME: memoryManager MODE: 0
NAME: sigma0 MODE: 0
INFO:: Memory allocated for process 10
INFO:: Memory reallocated for process 10
INFO:: Memory reallocated for process 10
```

```
INFO:: Memory reallocated for process 10
INFO:: Memory with PID '10' free'd.

NAME: fileManager MODE: 0
NAME: memoryManager MODE: 0
NAME: sigma0 MODE: 0
INFO:: PROCESS TABLE SIZE: :: 5


INFO:: PROCESS TABLE SIZE: :: 6


INFO:: Memory allocated for process 11
INFO:: Memory reallocated for process 11
INFO:: Memory reallocated for process 11
INFO:: Memory reallocated for process 11
INFO:: Memory with PID '11' free'd.

NAME: fileManager MODE: 0
NAME: memoryManager MODE: 0
NAME: sigma0 MODE: 0
INFO:: PROCESS TABLE SIZE: :: 6


INFO:: Memory allocated for process 12
INFO:: Memory reallocated for process 12
INFO:: Memory reallocated for process 12
INFO:: Memory reallocated for process 12
INFO:: Memory with PID '12' free'd.

NAME: fileManager MODE: 0
NAME: memoryManager MODE: 0
NAME: sigma0 MODE: 0
INFO:: Memory allocated for process 13
INFO:: Memory reallocated for process 13
INFO:: Memory with PID '13' free'd.

NAME: fileManager MODE: 0
NAME: memoryManager MODE: 0
NAME: sigma0 MODE: 0
INFO:: PROCESS TABLE SIZE: :: 5
INFO:: Memory allocated for process 14
INFO:: Memory reallocated for process 14
```

```
INFO:: Memory reallocated for process 14
INFO:: Memory with PID '14' free'd.

NAME: fileManager MODE: 0
NAME: memoryManager MODE: 0
NAME: sigma0 MODE: 0
INFO:: Memory allocated for process 15
INFO:: Memory reallocated for process 15
INFO:: Memory reallocated for process 15
INFO:: Memory reallocated for process 15
NAME: fileManager MODE: 0
NAME: memoryManager MODE: 0
NAME: sigma0 MODE: 0
INFO:: Memory allocated for process 16
INFO:: Memory reallocated for process 16
INFO:: Memory reallocated for process 16
INFO:: Memory reallocated for process 16
INFO:: Memory reallocated for process 16
INFO:: Memory reallocated for process 16
INFO:: Memory with PID '16' free'd.

INFO:: Changed to User Mode
NAME: fileManager MODE: 1
INFO:: Changed to User Mode
NAME: memoryManager MODE: 1
INFO:: Changed to User Mode
NAME: sigma0 MODE: 1
INFO:: Memory allocated for process 17
INFO:: Memory reallocated for process 17
INFO:: Memory reallocated for process 17
INFO:: Memory reallocated for process 17
INFO:: Memory with PID '17' free'd.

NAME: fileManager MODE: 1
NAME: memoryManager MODE: 1
NAME: sigma0 MODE: 1
INFO:: Memory allocated for process 18
INFO:: Memory reallocated for process 18
INFO:: Memory reallocated for process 18
INFO:: Memory reallocated for process 18
INFO:: Memory with PID '18' free'd.

NAME: fileManager MODE: 1
```

```
NAME: memoryManager MODE: 1
NAME: sigma0 MODE: 1
INFO:: Memory allocated for process 19
INFO:: Memory reallocated for process 19
INFO:: Memory reallocated for process 19
INFO:: Memory with PID '19' free'd.

NAME: fileManager MODE: 1
NAME: memoryManager MODE: 1
NAME: sigma0 MODE: 1
INFO:: Memory allocated for process 20
INFO:: Memory reallocated for process 20
INFO:: Memory reallocated for process 20
NAME: fileManager MODE: 1
NAME: memoryManager MODE: 1
NAME: sigma0 MODE: 1
INFO:: QUEUE IS EMPTY..


REPORT:

Process: 1
Mode: USER
Start: 108956947565975
End: 108961189705041
Duration: 4242139066

=========================================================

Process: 2
Mode: KERNEL
Start: 108961258141953
End: 108961864459608
Duration: 606317655

=========================================================

Process: 3
Mode: KERNEL
Start: 108961864723513
End: 108962667291049
Duration: 802567536
```

==========

Process: 4
Mode: KERNEL
Start: 108962667487512
End: 108962856228442
Duration: 188740930

==========

Process: 5
Mode: KERNEL
Start: 108962856465481
End: 108963098319620
Duration: 241854139

==========

Process: 6
Mode: KERNEL
Start: 108963098582477
End: 108963916640750
Duration: 818058273

==========

Process: 7
Mode: KERNEL
Start: 108963916864063
End: 108964625452530
Duration: 708588467

==========

Process: 8
Mode: KERNEL
Start: 108964625727167
End: 108965408312697
Duration: 782585530

==========

Process: 9

Mode: KERNEL
Start: 108965408489761
End: 108965648508061
Duration: 240018300

==========

Process: 10
Mode: KERNEL
Start: 108965648859072
End: 108965688428910
Duration: 39569838

==========

Process: 11
Mode: KERNEL
Start: 108965688643080
End: 108966580560522
Duration: 891917442

==========

Process: 12
Mode: KERNEL
Start: 108966580729162
End: 108967223508025
Duration: 642778863

==========

Process: 13
Mode: KERNEL
Start: 108967223736608
End: 108967679180434
Duration: 455443826

==========

Process: 14
Mode: KERNEL
Start: 108967679350347
End: 108968263406866

Duration: 584056519

==========

Process: 15
Mode: KERNEL
Start: 108968263624732
End: 108968625040930
Duration: 361416198

==========

Process: 16
Mode: KERNEL
Start: 108968625290858
End: 108968686031732
Duration: 60740874

==========

Process: 17
Mode: USER
Start: 108968686386421
End: 108973919294752
Duration: 5232908331

==========

Process: 18
Mode: USER
Start: 108973919548011
End: 108979164301151
Duration: 5244753140

==========

Process: 19
Mode: USER
Start: 108979164682227
End: 108983710903237
Duration: 4546221010

==========

Process: 20
Mode: USER
Start: 108983728433485
End: 108988496038160
Duration: 4767604675

==========

Other related material

Matplotlib -

Matplotlib is a python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. matplotlib can be used in python scripts, the python and ipython shell (ala MATLAB®* or Mathematica®†), web application servers, and six graphical user interface toolkits.