VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

OPERATING SYSTEM (CO2017)

Assignment report

SIMPLE OPERATING SYSTEM

Advisor: Dr. Lê Thanh Vân
Students: Vũ Nguyễn Lan Vi - 2153094
Thiều Quang Tuấn Anh - 2153171
La Cẩm Huy - 2153379

HO CHI MINH CITY, MAY 2023

Member list & Workload

No.	Full name	Student ID	Contribution
1	Vũ Nguyễn Lan Vi	2153094	33.33%
2	Thiều Quang Tuấn Anh	2153171	33.33%
3	La Cẩm Huy	2153379	33.33%

Contents

Chapter 1

Introduction

Operating systems have become an integral part of modern life, and their impact on human life is difficult to overstate. They are the foundation upon which all computing technology is built, enabling us to interact with computers, smartphones, tablets, and other digital devices in a seamless and intuitive manner. In essence, an operating system is a software program that manages the hardware and software resources of a computer, allowing other applications to run efficiently and effectively.

Figure 1.1: *The integration of operating system in modern life.*

Operating systems play a crucial role in enabling us to accomplish various tasks, such as communicating with others, accessing and storing information, managing finances, and performing work-related functions. They have also revolutionized the way we consume entertainment, making it easier to watch movies, play games, and listen

to music, among other things.

The development of operating systems has had a significant impact on human life, enabling us to achieve levels of productivity and efficiency that would have been unimaginable just a few decades ago. The evolution of operating systems has been critical in shaping the way we live, work, and interact with the world. As technology advances and becomes even more ubiquitous, operating systems will continue to play an essential role in facilitating communication, commerce, entertainment, education, and much more. In this age of rapid technological change, the importance of operating systems in human life is only set to grow, and their continued development will be critical in driving innovation and progress in the years to come.

While modern operating systems have come a long way in terms of performance and functionality, technology

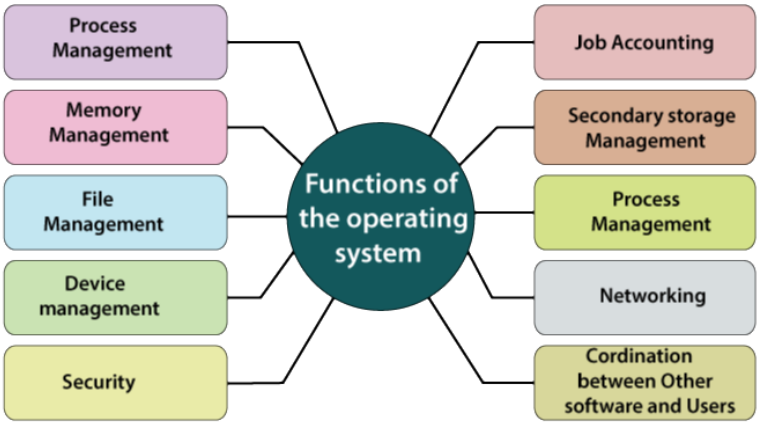

Figure 1.2: *Functions of the operating system.*

is constantly evolving and users' needs are constantly changing. Therefore, it's important for companies to continuously work on improving and updating their operating systems to keep up with these changes. In addition, as more and more people rely on technology in their daily lives, the demand for operating systems that are user-friendly, intuitive, and secure is increasing. Companies that are able to meet these demands are likely to have a competitive advantage in the market.

There are several companies that are improving operating systems nowadays, some of which include:

1. **Microsoft:** Microsoft is the company behind the Windows operating system, which is one of the most widely used operating systems in the world. Microsoft is continuously working to improve the functionality, security, and performance of the Windows operating system through regular updates and new releases.

2. **Apple:** Apple is the company behind the macOS and iOS operating systems, which are used on Mac computers, iPhones, iPads, and other Apple devices. Apple is known for its focus on user experience and is continuously improving its operating systems to provide a seamless and intuitive experience for its users.

3. **Google:** Google is the company behind the Android operating system, which is the most widely used mobile operating system in the world. Google is continuously working to improve the security, performance, and functionality of the Android operating system through regular updates and new releases.

4. **Canonical:** Canonical is the company behind the Ubuntu operating system, which is a popular Linux-based operating system used by individuals and businesses around the world. Canonical is continuously working to improve the Ubuntu operating system to make it more user-friendly and accessible.

5. **Red Hat:** Red Hat is a company that provides enterprise-level Linux-based operating systems and software solutions. Red Hat is continuously working to improve its operating systems to meet the needs of businesses and organizations around the world.

These are just a few examples of companies that are continuously working to improve operating systems. There are many other companies and organizations that are also involved in the development and improvement of operating systems, including open-source communities and individual developers.

Figure 1.3: *Most popular operating systems.*

This assignment will teach us how to build a simple operating system, which will help us understand fundamental concepts such as scheduling, synchronization, and memory management. We will primarily be using the C programming language for this project. There are two main reasons we have chosen C: its speed and the fact that the course focuses on Linux.

By completing this assignment, we will gain valuable knowledge and experience in the core concepts that underpin operating systems. We will learn how to manage system resources, schedule processes, and ensure data integrity through synchronization. Additionally, working with C will give us a deeper understanding of how low-level programming languages can interact with the hardware and perform tasks efficiently.

Overall, this assignment is an exciting opportunity for us to gain hands-on experience in building an operating system and deepen our understanding of the principles and techniques involved.

Chapter 2

Scheduler

2.1 Theoretical basis - CPU scheduling

2.1.1 Motivational Concepts

Only one process can run at a time in a system with **a single CPU core** while others will have to wait until the CPU's core becomes available and can be rescheduled. The goal of multi-programming is to have **at least one** process running at all times in order to maximize CPU utilization:

- Without the help of multi-programming, when a process is forced to wait for the completion of an I/O request, the CPU simply sits idle and wastes time.
- With sensible deployment of multi-programming, when one process has to wait for an I/O request, the operating system will take away the CPU and give it to another process; thus making the CPU busy all the time.

Since the operating system can increase the computer productivity by moving the CPU between processes, it is said that the foundation of a multi-programmed operating system is CPU scheduling.

2.1.2 Scheduling algorithms

During theory lectures, 6 basic scheduling algorithms were introduced, including:

1. First Come, First Served
2. Shortest Job First
3. Round Robin
4. Priority
5. Multi-level Queue

6. Multi-level Feedback Queue

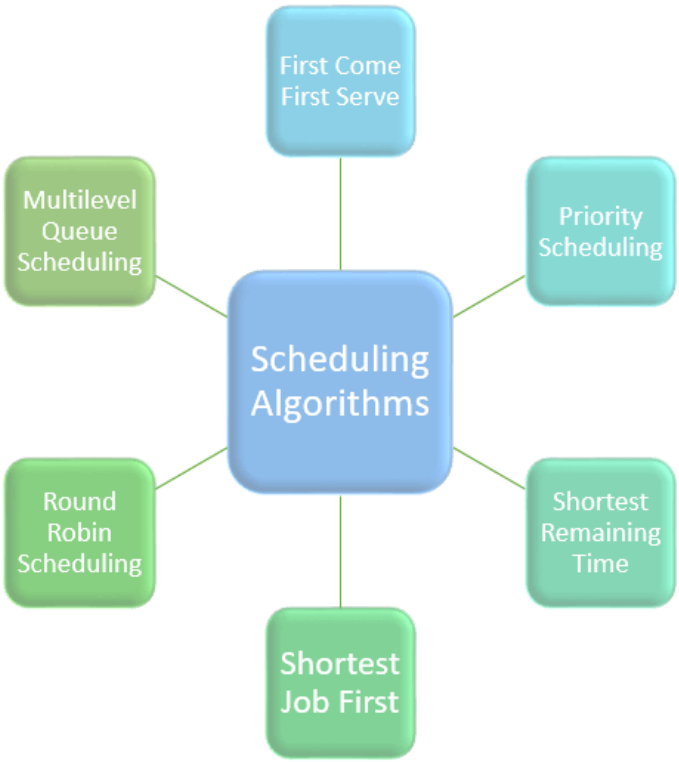

Figure 2.1: *Basic scheduling algorithms.*

In this assignment, we mainly focus on building the simple operating system which has the **Multilevel Queue (MLQ) scheduling**. So, what is Multilevel Queue scheduling?

MLQ scheduling is one of many CPU scheduling algorithms. Processes are divided into multiple queues based on their priority, with each queue having a different priority level. Higher-priority processes are placed in queues with higher priority levels, while lower-priority processes are placed in queues with lower priority levels.

The processes in each queue are assigned a priority level based on their characteristics, such as the type of process, its priority level, and the amount of resources it requires. The highest priority queue contains the most important processes, such as real-time processes that require immediate attention. The lowest priority queue contains the least important processes, such as background processes that can run in the background without affecting system performance.

Each queue has its own scheduling algorithm, such as round-robin, priority-based, or shortest-job-first, which is used to schedule the processes in that queue. The algorithm used for each queue can be customized based on the needs of that particular queue. In this assignment, **round-robin** is used to schedule each single queue.

2.1.3 Advantages and disadvantages of using MLQ

Advantages

- **Better prioritization:** The multilevel queue algorithm allows processes to be prioritized based on their characteristics, such as the type of process, its priority level, and the amount of resources it requires. This ensures that high-priority processes receive the necessary resources and attention, while low-priority processes are still executed without affecting the performance of higher-priority processes.
- **Improved system performance:** The multilevel queue algorithm can improve system performance by ensuring that high-priority processes are executed quickly and efficiently, without waiting in long queues. This can lead to faster response times, better resource utilization, and higher throughput.
- **Increased fairness:** The multilevel queue algorithm can provide fairness by ensuring that low-priority processes are not starved for resources. Each queue is assigned a certain amount of CPU time, ensuring that all processes are given a fair share of system resources.
- **Better management of different types of processes:** The multilevel queue algorithm can manage different types of processes more effectively than other scheduling algorithms. For example, interactive processes that require immediate attention can be assigned to a high-priority queue, while long-running batch processes can be assigned to a lower-priority queue.

Disadvantages

- **Complexity:** The multilevel queue algorithm can be complex to manage, especially when there are many different types of processes with varying priorities and resource requirements. This complexity can lead to longer queue wait times and a higher likelihood of system errors or failures.
- **Overhead:** The multilevel queue algorithm can introduce additional overhead in the system, such as the need to manage multiple queues and scheduling algorithms. This overhead can lead to slower system performance and reduced throughput.
- **Difficulty in determining queue parameters:** Determining the optimal parameters for each queue, such as the priority levels, time slices, and scheduling algorithms, can be challenging. If the queue parameters are not set correctly, it can lead to inefficient resource utilization or starvation of low-priority processes.
- **Inflexibility:** The multilevel queue algorithm may not be as flexible as other scheduling algorithms in certain situations. For example, it may not be able to handle sudden changes in workload or priorities as well as other algorithms.

2.1.4 Feedback and non-feedback mechanism Multi-level Queue

The Linux kernel uses the Priority Feedback Queue scheduling algorithm, which shares similarities with the Multi-level Queue and Multi-level Feedback Queue algorithms that have been studied. Before delving into the

coding aspect, it is important to understand this scheduling algorithm and its benefits over the previously studied ones.

Multi-level queue (main CPU scheduling algorithm in this assignment)

- In a multi-level queue scheduling algorithm, all processes are categorized into a specific queue based on their initial relative priority, and cannot change queues once they have been classified. For example, in the figure provided, real-time processes have the highest priority, meaning that processes in this queue are given CPU time first. Only when the real-time processes queue is empty will the CPU be allocated to processes in the system processes queue.
- Each queue is assigned a fixed priority level, and the processes are classified into the queues based on their priority. The highest-priority queue contains the most critical processes, while the lower-priority queues contain less critical processes.
- Furthermore, if a real-time process pre-empts an interactive process while it is running, and this happens frequently, processes in the lower priority queue are at risk of experiencing starvation.

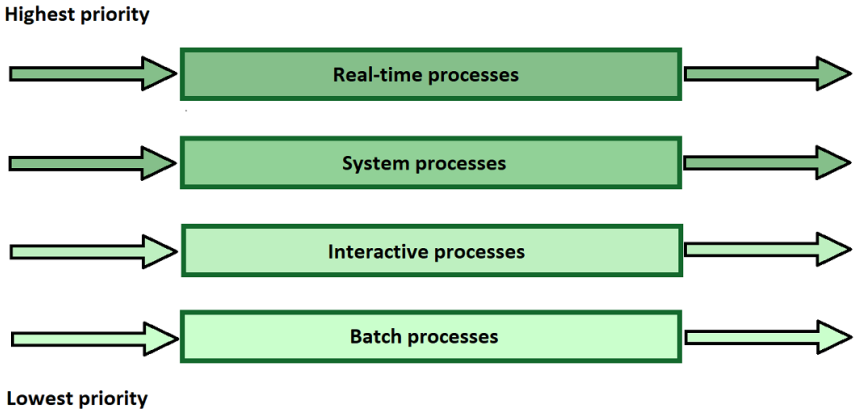

Figure 2.2: *Multi-level queue.*

Multi-level Feedback Queue

- The Multi-level Feedback Queue algorithm resolves the issue of starvation present in the Multi-level Queue algorithm by utilizing an aging mechanism. This mechanism enables processes to be moved to a lower level queue if they are unable to complete their execution within the allocated time slice at the current level.
- For instance, in the provided illustration, a process is initially given 8ms of CPU time, after which it is transferred to the second level queue, where it is allotted 16ms of execution time. If the process still fails

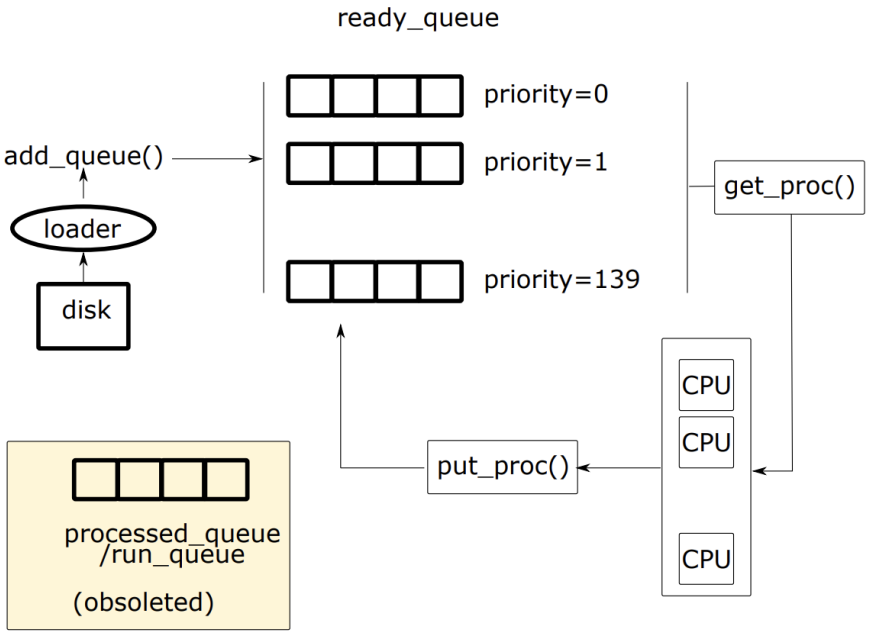

Figure 2.3: *The operation of scheduler in the assignment (using non-feedback MLQ).*

to complete its task, it is moved to the last level queue, which utilizes the First-Come-First-Serve (FCFS) algorithm to allow the process to execute until it finishes.

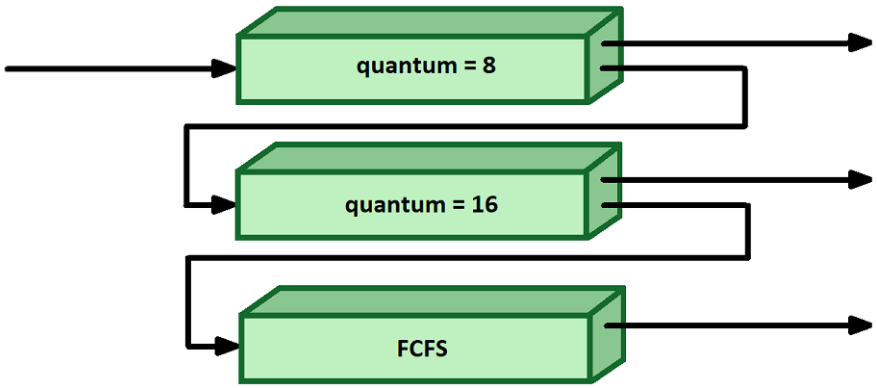

Figure 2.4: *Multi-level Feedback queue.*

Question 1

What is the advantage of using priority queue in comparison with other scheduling algorithms you have learned?

Answer

In comparison with:

- **FCFS**
  - The FCFS algorithm is easy to implement and is suitable for systems with low process arrival rates and short execution times. However, it can lead to long waiting times for high-priority tasks if they arrive after low-priority tasks. Multilevel priority queue scheduling allows critical tasks to be executed before less important tasks. This can be particularly important in real-time systems, where certain tasks must be completed within a specific timeframe.
  - Using a priority queue can offer greater flexibility compared to FCFS due to the ability to assign priorities to each queue. By dividing tasks into multiple ready queues with varying levels of priority, developers have more options to create a system that operates in the most optimal manner.
  - In summary, multilevel priority queue scheduling provides several advantages over FCFS, including better prioritization of critical tasks, improved resource utilization, reduced waiting times, and greater flexibility.
- **SJF**
  - While the SJF algorithm improves upon FCFS in terms of average waiting time, it still cannot be compared to a priority queue due to similar reasons in the FCFS algorithm.
  - Overall, a multilevel priority queue offers more flexibility, prioritization, fairness, and optimization compared to SJF, which makes it a more powerful and versatile scheduling algorithm.
- **Round-robin**
  - Round-robin schedules tasks by giving each task a fixed amount of time to run before moving on to the next task. This approach is useful when there are a large number of tasks that need to be executed and the system wants to ensure that each task gets a fair share of CPU time.
  - One of the main benefits of using a priority queue over round-robin is that it enables the system to execute higher-priority tasks quickly. This feature is especially useful in scenarios where there are crucial tasks that require urgent attention. By prioritizing these tasks, a priority queue can prevent delays and guarantee that they are finished before less critical ones. Conversely, round-robin may not offer comparable task prioritization, potentially resulting in delays when it comes to completing critical tasks.
- **Priority**

- The challenge of starvation is faced by both pre-emptive and non-preemptive versions with priority scheduling. However, the Priority Queue mentioned earlier can help avoid such scenarios.
- Moreover, the multiple queue mechanism helps reduce time consumption as it eliminates the need to search through the entire list of the processes to find the highest priority process.

- **Multi-level Feedback Queue**
  - One of the key advantages of a priority queue is its simple and efficient method for assigning priorities to processes. With a priority queue, the process with the highest priority is always executed next, which means there is no need to spend time or resources on shifting processes between different queues based on their changing priorities. In contrast, a multi-level feedback queue requires processes to move between different queues as their priority changes over time. This can make it difficult to predict which process will be executed next, leading to longer response times and lower overall system performance.
  - Another advantage of a priority queue is its ease of implementation and maintenance compared to a multi-level feedback queue. Priority queues can be created using a variety of data structures, such as arrays, linked lists, and binary heaps. Updating the priority of a process is as simple as changing its position in the queue. This makes it easier to manage and maintain the queue over time. In contrast, a multi-level feedback queue requires complex data structures and algorithms to manage the movement of processes between queues. This can be more challenging to implement and maintain, especially as the system grows more complex.

2.2 Algorithm and code

2.2.1 Declaration and initialization

Now, our task is to implement four functions:

```
1    // Enqueue function in queue.c
2    void enqueue(struct queue_t * q, struct pcb_t * proc)
3
4    // Dequeue function in queue.c
5    struct pcb_t * dequeue(struct queue_t * q)
6
7    // Get process function in sched.c
8    struct pcb_t * get_mlq_proc(void)
9
10   // Put process function in sched.c
11   void put_mlq_proc(struct pcb_t * proc)
```

2.2.2 Enqueue

- **Programming code**

```
void enqueue(struct queue_t * q, struct pcb_t * proc) {
    /* TODO: put a new process to queue [q] */
    if(q->size < MAX_QUEUE_SIZE)
    {
            q->proc[(q->cursor + q->size) % MAX_QUEUE_SIZE] = proc;
            q->size++;
    }
    else
    {
            printf("FULL QUEUE");
    }
}
```

- **Explanation**
  - First of all, in order to guarantee the queue capacity is not exceeded when adding a process, we first check if the current size of the corresponding queue is smaller than the maximum capacity, `MAX_QUEUE_SIZE`.
  - If the current size is not smaller than the capacity, it becomes impossible to add new processes to the queue. In such a case, we output the message "FULL QUEUE".
  - Otherwise, if the current size is smaller than the capacity, we locate the process at the “end” position of the queue. Since we are using a circular array, the “end” index for the process is calculated as `(q->cursor + q->size) % MAX_QUEUE_SIZE` . We append the process at this index and then increase the size of the queue.

2.2.3 Dequeue

- **Programming code**

```
struct pcb_t * dequeue(struct queue_t * q) {
    /* TODO: return a pcb whose prioprity is the highest
     * in the queue [q] and remember to remove it from q
     * */
    if(q->size <= 0) return NULL;
    else
    {
            // cursor = (cursor + size) % MAX_SIZE_SLOT;
            struct pcb_t* proc = q->proc[q->cursor];
            q->cursor = (q->cursor + 1) % MAX_QUEUE_SIZE;
            q->size--;
            q->timeslot--;
            return proc;
    }
}
```

- **Explanation**

- Firstly, we need to examine whether the queue is empty or not. If it is empty, return NULL as there are no processes to retrieve.
- If the queue is not empty, retrieve the process located at the cursor position. Then, update the cursor by assigning it a new value `(q->cursor + 1) % MAX_QUEUE_SIZE`. Finally, decrease the size and time slot of the current queue before returning the obtained process.

2.2.4 Get a process from MLQ

- **Programming code**

```
struct pcb_t * get_mlq_proc(void)
{
      struct pcb_t * proc = NULL;
      /*TODO: get a process from PRIORITY [ready_queue].
       * Remember to use lock to protect the queue.
       * */
      pthread_mutex_lock(&queue_lock);
      int last_check = 0;
      for (; last_check < MAX_PRIO; last_check++) {
        if (mlq_ready_queue[last_check].timeslot > 0) {
          break;
        }
      }
      // Reset the timeslot for every queue
      if (last_check == MAX_PRIO) {
        for (int i = 0; i < MAX_PRIO; i++) {
          mlq_ready_queue[i].timeslot = MAX_PRIO - i;
        }
      }
      for (int i = 0; i < MAX_PRIO; i++)
      {
        if(mlq_ready_queue[i].timeslot > 0 && mlq_ready_queue[i].size > 0)
        {
          proc = dequeue(&mlq_ready_queue[i]);
          pthread_mutex_unlock(&queue_lock);
          return proc;
        }
      }
      pthread_mutex_unlock(&queue_lock);

      return proc;
}
```

- **Explanation**

- First of all, we need to protect our `ready_queue` from race conditions by utilizing a mutex lock (`queue_lock`) to handle simultaneous adding or removal of multiple processes of the current queue at the same time.

– Next, iterate through all 140 `mlq_ready_queue` to check if all queues have run out of time slots or not. In this case, we have to loop over and reset the time slot for every queue.

– Finally, check the queues one by one in ascending order of priority. If a non-empty queue with a positive time slot is found, call dequeue in the selected queue, release the mutex lock, and return the corresponding process. If all queues have been inspected and no suitable queue is found, the returned process will be NULL.

2.2.5 Put a process to MLQ

- **Programming code**

```
1     void put_mlq_proc(struct pcb_t * proc) {
2           pthread_mutex_lock(&queue_lock);
3           enqueue(&mlq_ready_queue[proc->prio], proc);
4           pthread_mutex_unlock(&queue_lock);
5         }
```

- **Explanation**

– To ensure proper synchronization, call mutex lock to acquire the queue lock. Then, invoke the enqueue function in the queue that has the same priority compared to the required process. Finally, release the queue lock to complete the operation.

2.3 Results and Conclusion

2.3.1 Gantt chart and detailed explanation for ouputs

- **Gantt chart**

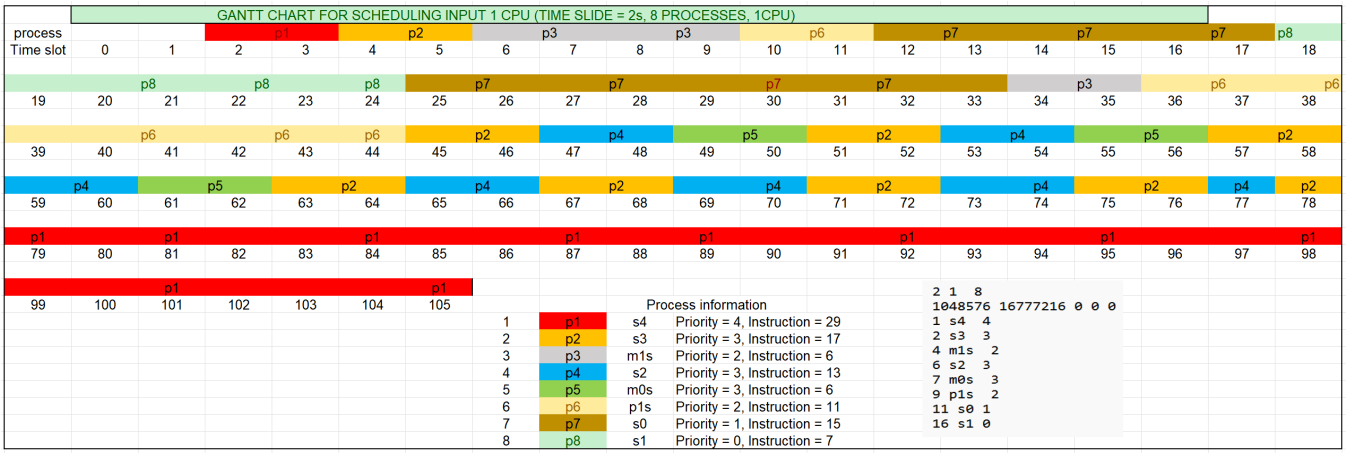

Figure 2.5: *Gantt chart for scheduling input 1 CPU.*

- **Explanation**
  Each process when dispatched will be given a time slice of 2ms (2 time slots). After this time slice expires, the process will be moved to the `ready_queue` which has similar priority. **Figure 2.5** is the example for 1 CPU.
  - At time slot 1, the CPU loads process 1 and dispatches it at time slot 2.
  - At time slot 2, the CPU loads Process 2 and dispatches it at time slot 4 due to its higher priority.
  - Process 3 arrives at time slot 6 and is executed instead of Process 2 because of its higher priority.
  - At time slot 10, Process 6 is dispatched following queue mechanism.
  - At time slot 12, Process 7 is dispatched and executed for the next 6 time slots.
  - The CPU loads the highest priority process, which is Process 8, at time slot 18, dispatches it, and completes its execution within the next 6 time slots.
  - At time slot 34, Process 7 is finished, Process 3 takes its place.
  - Process 3 completes its execution at time slot 36, and is replaced by Process 6 which runs until time slot 45 to finish.
  - At time slot 45, Processes 2, 4, and 5 have the same priority. As a result, a queue mechanism is implemented and each process will take turns running for 2 time slots until they all finish.
  - At time slot 79, Process 1 uses CPU to execute.
  - Finally, at time slot 106, Process 1 completes its execution, which marks the end of the program.
- **Output**

```
Time slot   0
ld_routine
Time slot   1
  Loaded a process at input/proc/s4, PID: 1 PRIO: 4
Time slot   2
  CPU 0: Dispatched process  1
  Loaded a process at input/proc/s3, PID: 2 PRIO: 3
Time slot   3
Time slot   4
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  2
  Loaded a process at input/proc/m1s, PID: 3 PRIO: 2
Time slot   5
Time slot   6
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  3
  Loaded a process at input/proc/s2, PID: 4 PRIO: 3
Time slot   7
  Loaded a process at input/proc/m0s, PID: 5 PRIO: 3
Time slot   8
  CPU 0: Put process  3 to run queue
```

```
  CPU 0: Dispatched process  3
Time slot   9
  Loaded a process at input/proc/p1s, PID: 6 PRIO: 2
Time slot  10
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  6
Time slot  11
  Loaded a process at input/proc/s0, PID: 7 PRIO: 1
Time slot  12
  CPU 0: Put process  6 to run queue
  CPU 0: Dispatched process  7
Time slot  13
Time slot  14
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  7
Time slot  15
Time slot  16
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  7
  Loaded a process at input/proc/s1, PID: 8 PRIO: 0
Time slot  17
Time slot  18
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  8
Time slot  19
Time slot  20
  CPU 0: Put process  8 to run queue
  CPU 0: Dispatched process  8
Time slot  21
Time slot  22
  CPU 0: Put process  8 to run queue
  CPU 0: Dispatched process  8
Time slot  23
Time slot  24
  CPU 0: Put process  8 to run queue
  CPU 0: Dispatched process  8
Time slot  25
  CPU 0: Processed  8 has finished
  CPU 0: Dispatched process  7
Time slot  26
Time slot  27
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  7
Time slot  28
Time slot  29
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  7
Time slot  30
Time slot  31
```

```
71        CPU 0: Put process  7 to run queue
72        CPU 0: Dispatched process  7
73      Time slot  32
74      Time slot  33
75        CPU 0: Put process  7 to run queue
76        CPU 0: Dispatched process  7
77      Time slot  34
78        CPU 0: Processed  7 has finished
79        CPU 0: Dispatched process  3
80      Time slot  35
81      Time slot  36
82        CPU 0: Processed  3 has finished
83        CPU 0: Dispatched process  6
84      Time slot  37
85      Time slot  38
86        CPU 0: Put process  6 to run queue
87        CPU 0: Dispatched process  6
88      Time slot  39
89      Time slot  40
90        CPU 0: Put process  6 to run queue
91        CPU 0: Dispatched process  6
92      Time slot  41
93      Time slot  42
94        CPU 0: Put process  6 to run queue
95        CPU 0: Dispatched process  6
96      Time slot  43
97      Time slot  44
98        CPU 0: Put process  6 to run queue
99        CPU 0: Dispatched process  6
100     Time slot  45
101       CPU 0: Processed  6 has finished
102       CPU 0: Dispatched process  2
103     Time slot  46
104     Time slot  47
105       CPU 0: Put process  2 to run queue
106       CPU 0: Dispatched process  4
107     Time slot  48
108     Time slot  49
109       CPU 0: Put process  4 to run queue
110       CPU 0: Dispatched process  5
111     Time slot  50
112     Time slot  51
113       CPU 0: Put process  5 to run queue
114       CPU 0: Dispatched process  2
115     Time slot  52
116     Time slot  53
117       CPU 0: Put process  2 to run queue
118       CPU 0: Dispatched process  4
119     Time slot  54
```

```
Time slot   55
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  5
Time slot   56
Time slot   57
  CPU 0: Put process  5 to run queue
  CPU 0: Dispatched process  2
Time slot   58
Time slot   59
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  4
Time slot   60
Time slot   61
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  5
write region=1 offset=20 value=25
Process ID: 5
print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000000
Time slot   62
write region=2 offset=1000 value=1
Process ID: 5
print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000000
SEGMENTATION FAULT: invalid page access
Time slot   63
  CPU 0: Processed  5 has finished
  CPU 0: Dispatched process  2
Time slot   64
Time slot   65
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  4
Time slot   66
Time slot   67
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  2
Time slot   68
Time slot   69
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  4
Time slot   70
Time slot   71
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  2
Time slot   72
Time slot   73
  CPU 0: Put process  2 to run queue
```

```
169       CPU 0: Dispatched process  4
170     Time slot  74
171     Time slot  75
172       CPU 0: Put process  4 to run queue
173       CPU 0: Dispatched process  2
174     Time slot  76
175     Time slot  77
176       CPU 0: Put process  2 to run queue
177       CPU 0: Dispatched process  4
178     Time slot  78
179       CPU 0: Processed  4 has finished
180       CPU 0: Dispatched process  2
181     Time slot  79
182       CPU 0: Processed  2 has finished
183       CPU 0: Dispatched process  1
184     Time slot  80
185     Time slot  81
186       CPU 0: Put process  1 to run queue
187       CPU 0: Dispatched process  1
188     Time slot  82
189     Time slot  83
190       CPU 0: Put process  1 to run queue
191       CPU 0: Dispatched process  1
192     Time slot  84
193     Time slot  85
194       CPU 0: Put process  1 to run queue
195       CPU 0: Dispatched process  1
196     Time slot  86
197     Time slot  87
198       CPU 0: Put process  1 to run queue
199       CPU 0: Dispatched process  1
200     Time slot  88
201     Time slot  89
202       CPU 0: Put process  1 to run queue
203       CPU 0: Dispatched process  1
204     Time slot  90
205     Time slot  91
206       CPU 0: Put process  1 to run queue
207       CPU 0: Dispatched process  1
208     Time slot  92
209     Time slot  93
210       CPU 0: Put process  1 to run queue
211       CPU 0: Dispatched process  1
212     Time slot  94
213     Time slot  95
214       CPU 0: Put process  1 to run queue
215       CPU 0: Dispatched process  1
216     Time slot  96
217     Time slot  97
```

```
218       CPU 0: Put process  1 to run queue
219       CPU 0: Dispatched process  1
220     Time slot  98
221     Time slot  99
222       CPU 0: Put process  1 to run queue
223       CPU 0: Dispatched process  1
224     Time slot 100
225     Time slot 101
226       CPU 0: Put process  1 to run queue
227       CPU 0: Dispatched process  1
228     Time slot 102
229     Time slot 103
230       CPU 0: Put process  1 to run queue
231       CPU 0: Dispatched process  1
232     Time slot 104
233     Time slot 105
234       CPU 0: Put process  1 to run queue
235       CPU 0: Dispatched process  1
236     Time slot 106
237       CPU 0: Processed  1 has finished
238       CPU 0 stopped
```

- **Gantt chart**

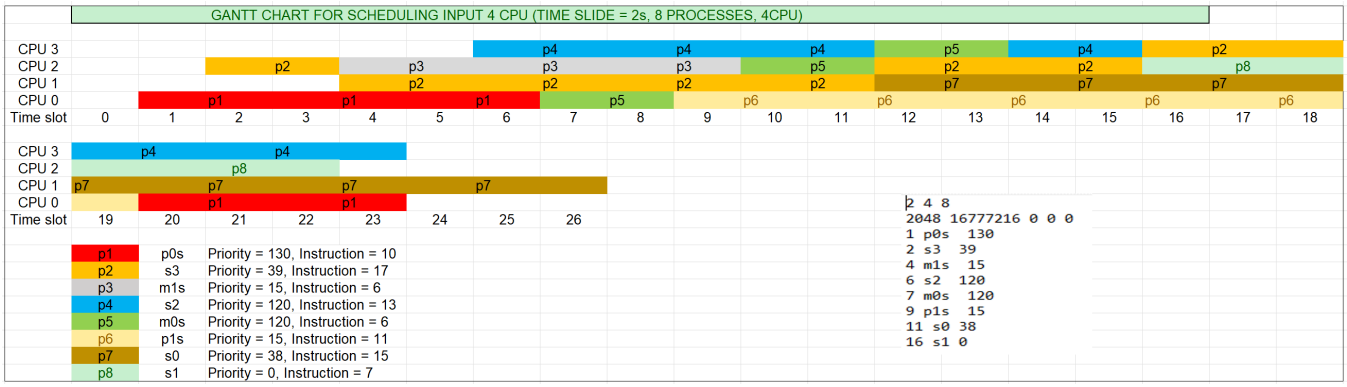

Figure 2.6: *Gantt chart for scheduling input 4 CPUs.*

- **Explanation**

  In contrast to the first example, where a single CPU was utilized for processing, the mechanism now employs four CPUs, resulting in improved system performance and efficiency. **Figure 2.6** corresponds to the example for 4 CPUs.

  – At time slot 1, process 1 gets in its ready queue and dispatched until time slot 6 by cpu 0.

  – At time slot 4, Process 3 uses CPU 2 and Process 2 uses CPU 1 because of the randomness of thread.

  – At time slot 6, Process 4 enters ready queue and dispatched by CPU 3.

- During time slot 7, Process 5 arrives and takes the place of Process 1, which was being processed by CPU 0. The decision to replace Process 1 instead of Processes 2, 3, and 4 was based on the fact that Process 1 had the lowest priority.
- During time slot 9, Process 6 replaces Process 5 for similar reasons that led to the replacement of Process 5 during time slot 7.
- At time slot 10, Process 3 completes its execution and Process 5 takes its place on CPU 2.
- At time slot 12, Process 7 comes and uses CPU 1 to execute, Process 5 is moved to its ready queue with priority 120, which is managed by CPU 3 and Process 2 is moved to CPU 2 due to the randomness of the thread.
- At time slide 14, Process 5 is finished.
- At time slot 19, Process 2 is finished and Process 4 uses the CPU 3.
- At time slot 20, Process 6 is finished and leave the CPU 0 for Process 1.
- At time slot 23, Process 8 is finished.
- At time slot 24, Process 1 and Process 4 are finished.
- At time slot 27, Process 7 is finished and all the program is finished.

- **Output**

```
Time slot   0
ld_routine
Time slot   1
  Loaded a process at input/proc/p0s, PID: 1 PRIO: 130
  CPU 0: Dispatched process  1
Time slot   2
  Loaded a process at input/proc/s3, PID: 2 PRIO: 39
  CPU 2: Dispatched process  2
Time slot   3
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot   4
  Loaded a process at input/proc/m1s, PID: 3 PRIO: 15
  CPU 2: Put process  2 to run queue
  CPU 2: Dispatched process  3
  CPU 1: Dispatched process  2
Time slot   5
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot   6
  CPU 1: Put process  2 to run queue
  CPU 1: Dispatched process  2
  Loaded a process at input/proc/s2, PID: 4 PRIO: 120
write region=1 offset=20 value=100
Process ID: 1
print_pgtbl: 0 - 1024
```

```
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
  CPU 2: Put process  3 to run queue
  CPU 2: Dispatched process  3
  CPU 3: Dispatched process  4
Time slot   7
  Loaded a process at input/proc/m0s, PID: 5 PRIO: 120
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  5
Time slot   8
  CPU 1: Put process  2 to run queue
  CPU 1: Dispatched process  2
  CPU 2: Put process  3 to run queue
  CPU 2: Dispatched process  3
  CPU 3: Put process  4 to run queue
  CPU 3: Dispatched process  4
Time slot   9
  Loaded a process at input/proc/p1s, PID: 6 PRIO: 15
  CPU 0: Put process  5 to run queue
  CPU 0: Dispatched process  6
Time slot  10
  CPU 1: Put process  2 to run queue
  CPU 1: Dispatched process  2
  CPU 2: Processed  3 has finished
  CPU 2: Dispatched process  5
  CPU 3: Put process  4 to run queue
  CPU 3: Dispatched process  4
Time slot  11
  Loaded a process at input/proc/s0, PID: 7 PRIO: 38
  CPU 0: Put process  6 to run queue
  CPU 0: Dispatched process  6
Time slot  12
  CPU 1: Put process  2 to run queue
  CPU 1: Dispatched process  7
  CPU 2: Put process  5 to run queue
  CPU 2: Dispatched process  2
  CPU 3: Put process  4 to run queue
  CPU 3: Dispatched process  5
write region=1 offset=20 value=25
Process ID: 5
print_pgtbl: 0 - 512
00000000: 80000007
00000004: 80000006
Time slot  13
  CPU 0: Put process  6 to run queue
  CPU 0: Dispatched process  6
write region=2 offset=1000 value=1
```

```
Process ID: 5
print_pgtbl: 0 - 512
00000000: 80000007
00000004: 80000006
SEGMENTATION FAULT: invalid page access
Time slot  14
  CPU 2: Put process  2 to run queue
  CPU 2: Dispatched process  2
  CPU 3: Processed  5 has finished
  CPU 1: Put process  7 to run queue
  CPU 1: Dispatched process  7
  CPU 3: Dispatched process  4
Time slot  15
  CPU 0: Put process  6 to run queue
  CPU 0: Dispatched process  6
Time slot  16
  Loaded a process at input/proc/s1, PID: 8 PRIO: 0
  CPU 2: Put process  2 to run queue
  CPU 2: Dispatched process  8
  CPU 1: Put process  7 to run queue
  CPU 1: Dispatched process  7
  CPU 3: Put process  4 to run queue
  CPU 3: Dispatched process  2
Time slot  17
  CPU 0: Put process  6 to run queue
  CPU 0: Dispatched process  6
Time slot  18
  CPU 2: Put process  8 to run queue
  CPU 2: Dispatched process  8
  CPU 1: Put process  7 to run queue
  CPU 1: Dispatched process  7
  CPU 3: Put process  2 to run queue
  CPU 3: Dispatched process  2
Time slot  19
  CPU 0: Put process  6 to run queue
  CPU 0: Dispatched process  6
  CPU 3: Processed  2 has finished
  CPU 3: Dispatched process  4
Time slot  20
  CPU 0: Processed  6 has finished
  CPU 0: Dispatched process  1
read region=1 offset=20 value=100
Process ID: 1
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
  CPU 2: Put process  8 to run queue
```

```
125       CPU 2: Dispatched process  8
126       CPU 1: Put process  7 to run queue
127       CPU 1: Dispatched process  7
128     Time slot  21
129     write region=2 offset=20 value=102
130     Process ID: 1
131     print_pgtbl: 0 - 1024
132     00000000: 80000001
133     00000004: 80000000
134     00000008: 80000003
135     00000012: 80000002
136       CPU 3: Put process  4 to run queue
137       CPU 3: Dispatched process  4
138     Time slot  22
139       CPU 0: Put process  1 to run queue
140       CPU 0: Dispatched process  1
141     read region=2 offset=20 value=102
142     Process ID: 1
143     print_pgtbl: 0 - 1024
144     00000000: 80000001
145     00000004: 80000000
146     00000008: 80000003
147     00000012: 80000002
148       CPU 2: Put process  8 to run queue
149       CPU 2: Dispatched process  8
150       CPU 1: Put process  7 to run queue
151       CPU 1: Dispatched process  7
152     Time slot  23
153     write region=3 offset=20 value=103
154     Process ID: 1
155     print_pgtbl: 0 - 1024
156     00000000: 80000001
157     00000004: 80000000
158     00000008: 80000003
159     00000012: 80000002
160       CPU 2: Processed  8 has finished
161       CPU 2 stopped
162       CPU 3: Put process  4 to run queue
163       CPU 3: Dispatched process  4
164     Time slot  24
165       CPU 0: Processed  1 has finished
166       CPU 0 stopped
167       CPU 1: Put process  7 to run queue
168       CPU 1: Dispatched process  7
169       CPU 3: Processed  4 has finished
170       CPU 3 stopped
171     Time slot  25
172     Time slot  26
173       CPU 1: Put process  7 to run queue
```

```
174       CPU 1: Dispatched process  7
175     Time slot  27
176       CPU 1: Processed  7 has finished
177       CPU 1 stopped
```

Chapter 3

Memory management

3.1 Theoretical basis – Memory management

3.1.1 Memory management

Main memory

Primary storage (also known as main memory, internal memory, or prime memory), often referred to simply as memory, is the only one directly accessible to the CPU. The CPU continuously reads instructions stored there and executes them as required. Any data actively operated on is also stored there in uniform manner.

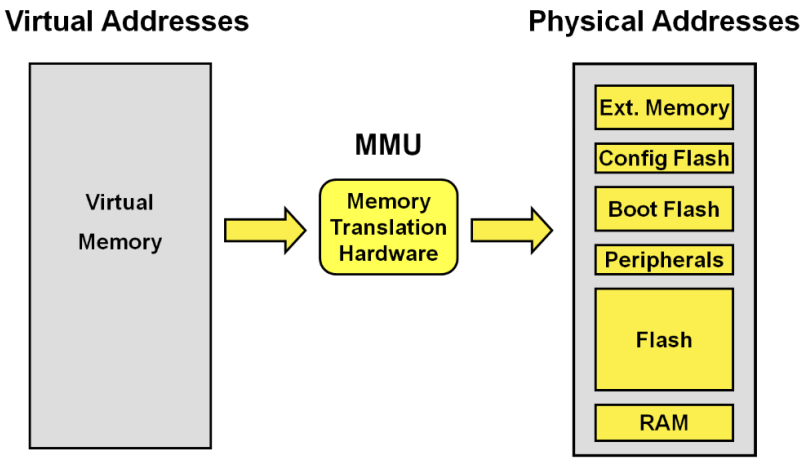

Figure 3.1: *MMU as an address translator.*

Main memory is directly or indirectly connected to the central processing unit via a memory bus. It is actually two buses (not on the diagram): an address bus and a data bus. The CPU firstly sends a number through an address bus, a number called memory address, that indicates the desired location of data. Then it reads or

writes the data in the memory cells using the data bus. Additionally, a memory management unit (MMU) is a small device between CPU and RAM recalculating the actual memory address, for example to provide an abstraction of virtual memory or other tasks (depicted in **Figure 3.1**).

Dynamic Loading and Dynamic Linking are two important mechanisms that can significantly improve the performance of computer programs. Loading a program into the main memory is a necessary step before its execution. However, it may not be optimal to load the entire program at once. Dynamic Loading is a mechanism that allows a program to load a certain part or routine into the main memory only when it is called by the program. By doing so, the overall performance of the system can be improved.

In addition, some programs may depend on other programs to run correctly. Rather than loading all the dependent programs at once, Dynamic Linking is a mechanism that links the dependent programs to the main executing program only when required by the CPU. This can help reduce the memory usage and improve the overall efficiency of the system.

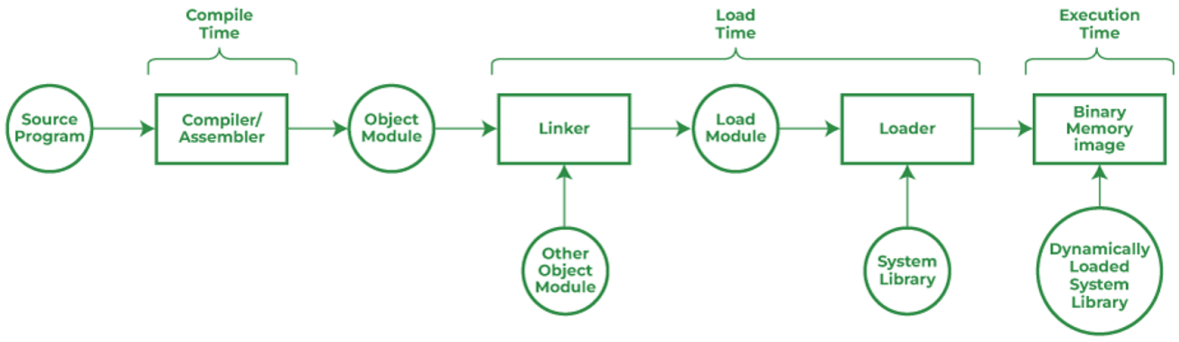

Figure 3.2: *Dynamic Loading and Dynamic Linking.*

What is memory management?

Memory management is a form of resource management applied to computer memory. The essential requirement of memory management is to provide ways to dynamically allocate portions of memory to programs at their request, and free it for reuse when no longer needed. This is critical to any advanced computer system where more than a single process might be underway at any time.

In a computer system, memory refers to the temporary storage area that is used by the processor to store program instructions and data. It can be classified into two categories: primary memory (also known as main memory) and secondary memory (such as hard disks or solid-state drives).

Memory management is crucial to ensure that the system runs smoothly and efficiently. Some of the key tasks involved in memory management include:

- **Allocation:** The process of assigning memory blocks to programs or processes that require them.

- **Deallocation:** The process of freeing up memory blocks that are no longer needed by a program or process.
- **Fragmentation management:** The process of managing memory fragmentation, which occurs when a program or process requests more memory than is available in contiguous blocks.
- **Protection:** Ensuring that programs or processes do not access memory that does not belong to them or that they are not authorized to access.
- **Swapping:** The process of moving data between primary and secondary memory to ensure that there is enough memory available for running programs.

Effective memory management is critical for optimizing the performance and stability of computer systems, and it is a fundamental aspect of operating system design.

Methods use in memory management

There are several methods involved in memory management, which are implemented by the operating system to effectively manage and allocate memory resources to programs. Some of the commonly used methods include:

- **Paging:** Paging involves dividing the physical memory into fixed-size page frames, and dividing each program into equal-sized pages. These pages are then loaded into the available page frames in physical memory. When a program needs more memory, additional pages are loaded into the available page frames, and when memory is no longer needed, pages are swapped out of memory.
- **Segmentation:** Segmentation involves dividing the program into variable-sized logical segments, such as code segments, data segments, and stack segments. Each segment is loaded into a separate area in physical memory. This method allows for more flexible memory allocation, as different segments of the program can be resized or moved as needed.
- **Fragmentation management:** This involves managing memory fragmentation, which can occur when memory is divided into small chunks, resulting in wasted memory that cannot be used. To reduce fragmentation, the operating system can use techniques such as memory compaction, where the physical memory is rearranged to create larger blocks of free memory.
- **Swapping:** Swapping involves moving entire processes or pages of a process between main memory and secondary storage (such as a hard disk) to free up space in the main memory.
- **Virtual memory:** Virtual memory is a technique where the operating system creates a virtual address space that is larger than the physical memory available in the system. The operating system then manages the mapping of virtual addresses to physical addresses, allowing programs to access memory that may not be available in physical memory.

In this assignment, we mainly focus on swapping methods and virtual memory methods.

3.1.2 Multiple memory segment

A memory segment is a contiguous block of memory in a computer system that has been allocated for a specific purpose. In some operating systems, multiple memory segments may be used to manage different types of data and program code in memory.

For example, in a typical program, there may be separate memory segments allocated for code (executable instructions), data (initialized and uninitialized variables), and a stack (used for function calls and local variable storage). Each segment may have its own size, location in memory, and access permissions.

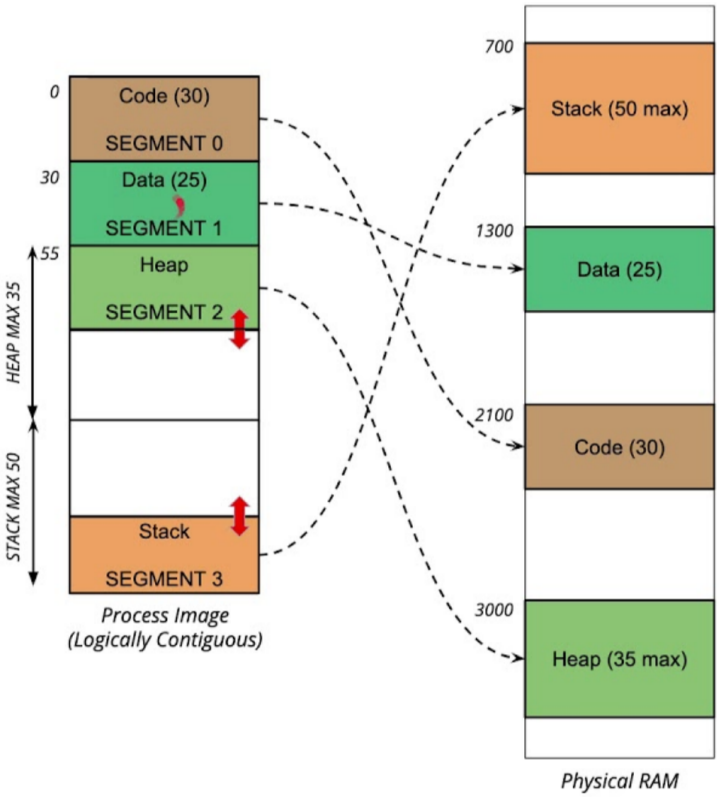

Figure 3.3: *Logical and physical view of memory segments.*

Question 2

In this simple OS, we implement a design of multiple memory segments or memory areas in source code declaration. What is the advantage of the proposed design of multiple segments?

Answer

- **Flexibility:** By using multiple memory segments, different parts of a program can be allocated to separate

segments, allowing for greater flexibility in memory management. This can be useful when dealing with complex data structures, such as arrays or linked lists, that require different types of memory allocation.

- **Security:** By assigning different access permissions to different memory segments, it is possible to provide better security for sensitive data. For example, a segment containing password data could be marked as read-only to prevent unauthorized modification.
- **Modular programming:** Using multiple memory segments can allow for a more modular programming approach, where different parts of a program are allocated to separate segments based on their function. This can make it easier to develop, debug, and maintain complex programs.
- **Improved memory utilization:** By allocating memory in separate segments, it is possible to avoid memory fragmentation and improve memory utilization. This can lead to better performance and more efficient use of system resources.
- **Platform portability:** By using multiple memory segments, it is possible to write code that is more easily ported to different platforms with varying memory architectures. This can make it easier to develop software that runs on a variety of different systems.

3.1.3 Segmentation with paging

The method of memory management known as Segmented Paging involves dividing physical memory into individual pages and then assigning each logical address used by a process to a corresponding physical page. Unlike other techniques that divide virtual memory into pages, Segmented Paging uses segments to map virtual memory addresses to physical memory addresses.

Traditionally, a program is divided into four segments, namely code segment, data segment, stack segment and heap segment. For the purposes of this assignment, our attention will be primarily focused in the data segment.

In each segment, a region is mapped to a frame in main memory through a page table. The page table contains mappings between virtual addresses used by the program and physical addresses in main memory. When a program accesses a virtual address, the page table is consulted to determine the corresponding physical address in main memory. This physical address is used to retrieve or store data in the main memory.

When a process needs to perform tasks such as reading, writing, or calculating, it must access main memory. To do this, the CPU generates a virtual memory address, which is then passed to the Memory Management Unit (MMU) to access frames in the main memory. In real world, CPU address has something similar to this:

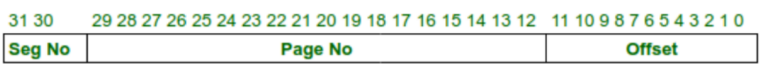

Figure 3.4: *An example of CPU address structure.*

The **Seg No** refers to the segment that a process desires, **Page No** is the specific page within that segment from which the process wishes to read data, and **Offset** indicates the location of the desired data within the page.

In our assignment, the CPU produces a CPU address which does not have a segment number because we only deal with 1 segment. So, CPU address has the structure similar to this:

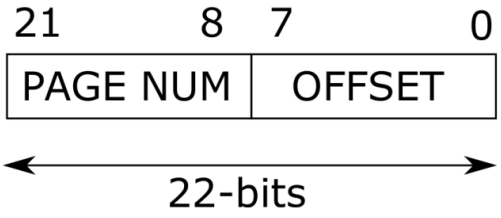

Figure 3.5: *Page table for each segment.*

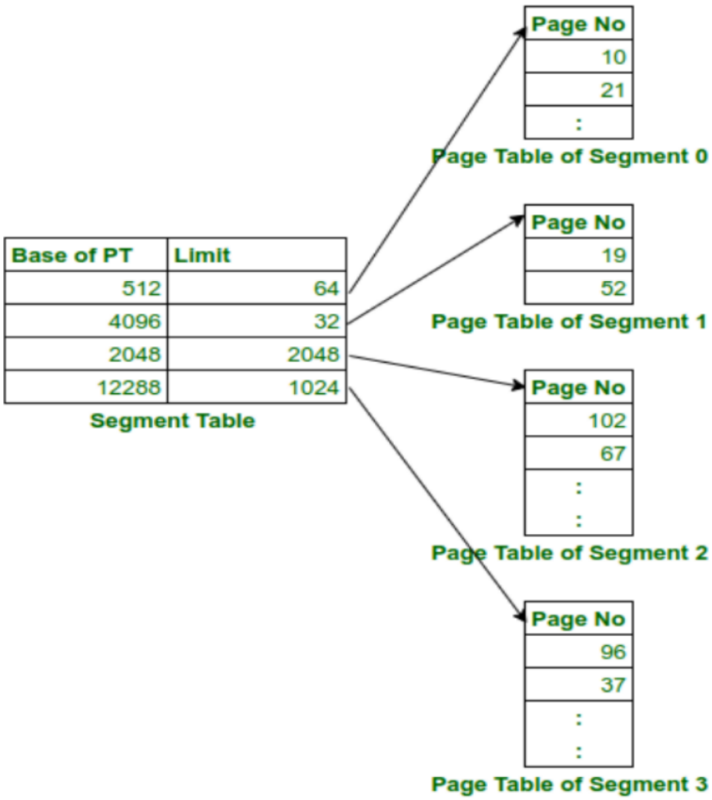

Figure 3.6: *Segmentation with paging.*

Question 3

What will happen if we divide the address to more than 2-levels in the paging memory management system?

Answer

- **Advantage:**
  - One advantage is that it allows for a larger address space, as more levels of page tables can be used to map a larger virtual address space to a smaller physical address space. This can be useful for applications that require a large amount of memory, such as scientific simulations or video processing.
  - Another advantage is that it can reduce the size of the page tables, as each level can have a smaller number of entries. This can improve the efficiency of the memory management system by reducing the time and space needed to access and store the page tables.
- **Disadvantage:**
  - One disadvantage is that it can increase the time needed to access memory, as more levels of page tables need to be traversed to translate a virtual address to a physical address. This can lead to slower performance and increased latency.
  - Another disadvantage is that it can increase the complexity of the memory management system, as more levels of page tables need to be managed and maintained. This can make the system more difficult to design, implement, and debug.

Question 4

What is the advantage and disadvantage of segmentation with paging?

Answer

- **Advantage:**
  - **Segment table contains only one entry corresponding to each segment:** Each entry in the segment table of segmentation with paging method only contains the pointer to the corresponding page table rather than multiple information like segment number, base address, and relevant data in the segment table of segmentation method.
  - **The size of the page table is limited by the segment size:** In the paging method, the size of the page table can grow dramatically when dealing with a large amount of data. Whereas, in segmentation with the paging method, the process is first split into segments. Then, each segment is further divided into pages. This procedure ensures that the page table's size cannot exceed the segment's size.

- **Memory usage reduction:**As the segment table is simplified and the size of page tables is put under control, it is clear that segmentation with a paging method will consume less memory.
- **It solves the problem of external fragmentation:** By incorporating paging, the application can harness its benefits, allowing processes to be loaded and removed from fixed-size pages rather than variable-size segments. This approach ensures that no additional gaps are created, effectively mitigating the issue of external fragmentation.

- **Disadvantage:**
  - **Suffering from internal fragmentation:** In segmentation with the paging method, data are stored in fixed-size pages, resulting in the same problem as in the paging method - internal fragmentation.
  - **The complexity level is much higher as compared to paging and segmentation:** Obviously, combining 2 different methods together will require a more sophisticated implementation, which leads to a higher complexity.

3.2 Algorithm and code

3.2.1 Alloc

- **Alloc function**

1. **Code**

```
int __alloc(struct pcb_t* caller, int vmaid, int rgid, int size, int* alloc_addr)
{
    /*Allocate at the toproof */
    struct vm_rg_struct rgnode;
    /*First of all, check free region list*/
    if (get_free_vmrg_area(caller, vmaid, size, &rgnode) == 0)
    {
        int pgn = PAGING_PGN(rgnode.rg_start);
        uint32_t pte = caller->mm->pgd[pgn];

        int isSwp = GETBIT(pte, 30, 31);
        if (isSwp) {

            int inc_sz = rgnode.rg_end - rgnode.rg_start;
            int inc_amt = PAGING_PAGE_ALIGNSZ(inc_sz);
            int incnumpage = inc_amt / PAGING_PAGESZ;

            for (int i = 0; i < incnumpage; i++) {
                int vicpage;

                int isFound = find_victim_page(caller->mm, &vicpage);
                if (isFound < 0) {
                    return -1;//already swapped -> must enough frame in RAM
```

```
24                }
25                uint32_t vicframe = GETVAL(caller->mm->pgd[vicpage],
    PAGING_PTE_FPN_MASK, PAGING_PTE_FPN_LOBIT);
26                uint32_t swpfpn = GETVAL(caller->mm->pgd[pgn + i],
    PAGING_PTE_FPN_MASK, PAGING_PTE_FPN_LOBIT);
27
28                __swap_cp_page(caller->mram, vicframe, caller->active_mswp, swpfpn);
29                enlist_pgn_node(&caller->mm->fifo_pgn, pgn+i);
30
31                pte_set_swap(&caller->mm->pgd[vicpage], 0, swpfpn);
32                pte_set_fpn(&caller->mm->pgd[pgn + i], vicframe);
33            }
34        }
35        caller->mm->symrgtbl[rgid].rg_start = rgnode.rg_start;
36        caller->mm->symrgtbl[rgid].rg_end = rgnode.rg_end;
37
38        *alloc_addr = rgnode.rg_start;
39
40        return 0;
41    }
42
43    /* get_free_vmrg_area FAILED handle the region management \*/
44
45    /* Attempt to increate limit to get space */
46    struct vm_area_struct* cur_vma = get_vma_by_num(caller->mm, vmaid);
47
48    int inc_sz = PAGING_PAGE_ALIGNSZ(size); // CALCULATING THE TOTAL PAGE SIZE THAT "
    size" NEED TO USED.
49
50    int old_sbrk, old_end;
51    old_sbrk = cur_vma->sbrk;
52    old_end = cur_vma->vm_end;
53
54    /* INCREASE THE LIMIT */
55    if (old_sbrk + size <= cur_vma->vm_end)
56    {
57        cur_vma->sbrk += size;
58        caller->mm->symrgtbl[rgid].rg_start = old_sbrk;
59        caller->mm->symrgtbl[rgid].rg_end = old_sbrk + size;
60        *alloc_addr = old_sbrk;
61    }
62    else
63    {
64        int increase = inc_vma_limit(caller, vmaid, size);
65        if (increase == -1)
66        {
67            return -1;
68        }
69        /*Successful increase limit */
```

```
70          cur_vma->sbrk = old_end + size;
71          caller->mm->symrgtbl[rgid].rg_start = old_end;
72          caller->mm->symrgtbl[rgid].rg_end = old_end + size;
73          *alloc_addr = old_end;
74      }
75      return 0;
76  }
77
```

2. **Explanation**
   - First of all, we check for available free regions of the process using the function `get_free_vmrg_area`. This function traverses the process's list of free regions, applying the first-fit algorithm to identify the first node with sufficient size for the required allocation.
   - If the function returns zero, it indicates the existence of free frames in the physical memory already belongs to the current process. In this scenario, there are two cases to consider:
     - If the frames are already present in RAM, we retrieve the returned region node and map it to the process's symbol region table (`symrgtbl`) to record the start and end points of the region.
     - If the frames are located in secondary memory (previously swapped out), we need to swap them back into primary memory (RAM). To do this, we find some victim frames in RAM (equivalent in quantity to the number of swapped frames), and use the `__swap_cp_page` function to copy the contents of those victim frames to the free swapped frames in SWAP. Additionally, we update the swap bit and offset swap values for the page table entries corresponding to the victim frames, while simultaneously, we assign frame numbers to the page table entries associated with the swapped frames. As a result, our free region now maps with a physical region located in RAM, then our final job is to keep track of their start and end points in the process's symbol region table.
   - If the function returns a negative value, it indicates that we have no available free regions. In such a case, we first calculate the total number of pages required for the allocation. Then, we verify if the remaining virtual space within our segment (the difference between the end point and the sbrk of the segment) is sufficient to accommodate the calculated page size.
     - If there is enough space, we update the area's sbrk and map the new region (from the old sbrk to the new sbrk) to the symbol region table of the current process.
     - If there is insufficient space, we invoke the `increase_vma_limit` function. If this function returns a negative value, it means we fail to expand the limit (end point) of the current segment. Otherwise, it indicates a successful increase in the segment's limit. Consequently, we update the new sbrk and limit values for the segment and record the new region information in the symbol region table.

- **Alloc page range function**

1. **Code**

```c
int alloc_pages_range(struct pcb_t *caller, int req_pgnum, struct framephy_struct** frm_lst)
{
    int pgit, fpn;

    for(pgit = 0; pgit < req_pgnum; pgit++)
    {
        int is_getfreefp_success = MEMPHY_get_freefp(caller->mram, &fpn);

        /* In case cannot get a free frame -> Swapping */
        if(is_getfreefp_success < 0)
        {
            /* Choose a victim page -> swap it to the SWAP SPACE (pg_getpage) */
            int vicpgn, swpfpn;

            int isFound = find_victim_page(caller->mm, &vicpgn);

            if(isFound < 0)
            {
              return -1;
            }
            uint32_t vicfpn = GETVAL(caller->mm->pgd[vicpgn], PAGING_PTE_FPN_MASK, PAGING_PTE_FPN_LOBIT);

            /* Get free frame in MEMSWP to store the value. */
            MEMPHY_get_freefp(caller->active_mswp, &swpfpn);

            /* Do swap frame from MEMRAM to MEMSWP and vice versa*/

            /* Copy victim frame to swap */
            __swap_cp_page(caller->mram, vicfpn, caller->active_mswp, swpfpn);

            /* Update page table for the victim page = swap frame*/
            pte_set_swap(&caller->mm->pgd[vicpgn], 0, swpfpn);

            fpn = vicfpn;
        }

        // Put the frame_number to the frm_lst to store them and return after get all the frame
        struct framephy_struct* currentFrame = malloc(sizeof(struct framephy_struct));
        currentFrame->fpn = fpn;
        currentFrame->owner = caller->mm;
        currentFrame->fp_next = *frm_lst;

        *frm_lst = currentFrame;
    }
    return 0;
```

```
46 }
47
```

2. **Explanation**
   - In order to allocate a specific number of frames in RAM, we initially check if there are available free frames by calling the function `MEMPHY_get_free_fp`. If we are unable to find a free frame in primary memory, we proceed with finding victim pages to be swapped to the swap space.
   - This is achieved through the function `find_victim_page`, which utilizes the FIFO page replacement algorithm to find victim pages within the process's page list (`fifo_pgn`). After finding the corresponding victim frames, we search for free frames in MEMSWAP to store the contents of the victim frames. Once the contents are copied, we set the swap bit and swap offset for the page table entries associated with the victim pages. The new allocated nodes with frame numbers are then added to the returned frame list to store them, and we continue this process until we obtain the required number of RAM frames. Finally, we return the completed frame list after iterating through the loop.

- **Vmap page range function**

1. **Code**

```
1  int vmap_page_range(struct pcb_t *caller, int addr, int pgnum, struct framephy_struct
       *frames, struct vm_rg_struct *ret_rg)
2  {
3      struct framephy_struct *fpit = frames;
4      /* map range of frame to address space
5      * [addr to addr + pgnum*PAGING_PAGESZ
6      * in page table caller->mm->pgd[]
7      */
8      while(fpit)
9      {
10         int pgn = PAGING_PGN(addr);
11         ret_rg->rg_start = addr;
12         ret_rg->rg_end = addr + PAGING_PAGESZ;
13
14         pte_set_fpn(&caller->mm->pgd[pgn], fpit->fpn); // set the property of the
       page into the page table
15
16         addr += PAGING_PAGESZ;
17         ret_rg->rg_next = malloc(sizeof(struct vm_rg_struct));
18         ret_rg = ret_rg->rg_next;
19
20         struct framephy_struct* temp = fpit;
21         fpit = fpit->fp_next;
22
23         free(temp);
24
25         /* Tracking for later page replacement activities (if needed)
```

```
26          * Enqueue new usage page */
27          enlist_pgn_node(&caller->mm->fifo_pgn, pgn);
28      }
29      return 0;
30  }
31
```

2. **Explanation**

   – Once we have obtained a list of free frames in RAM using the `alloc_page_range` function, the next step is to map the range of pages in our process's virtual memory region to these available frames in RAM. The mapping process involves the following steps:

   ○ We iterate over the number of aligned addresses and determine the corresponding page numbers for each address.

   ○ Using the process's page table (pgd), we map each page number to its respective frame number in RAM. This is achieved by utilizing the `pte_set_fpn` function to set the frame number bits for each page table entry.

   ○ During each iteration, we extend the end address of the return region to include the newly mapped region, and enqueue the newly used pages to the process's page list (`fifo_pgn`).

   – After completing the loop, we obtain the actual mapped region, where the pages in the process's virtual memory are now mapped to the corresponding free frames in RAM.

3.2.2 Free

1. **Code**

```
1  int __free(struct pcb_t* caller, int vmaid, int rgid) {
2    if (rgid < 0 || rgid >= PAGING_MAX_SYMTBL_SZ) {
3      return -1;
4    }
5    struct vm_rg_struct rgnode;
6    struct vm_rg_struct* res = get_symrg_byid(caller->mm, rgid);
7      if (res->rg_end <= res->rg_start) {
8          printf("SEGMENTATION FAULT: free invalid region\n");
9          return -1;
10     }
11   rgnode.rg_start = res->rg_start;
12   rgnode.rg_end = res->rg_end;
13   rgnode.rg_next = res->rg_next;
14
15   enlist_vm_freerg_list(caller->mm, rgnode);
16   return 0;
17 }
18
```

2. **Explanation**

- First, we check if the memory region ID, which is used to identify the variable in the symbol table, is valid or not. If it is a negative number or is greater than or equal to `PAGING_MAX_SYMTBL_SZ`, the region ID is invalid, we return -1.
- If the region ID is valid, we call the function `get_symrg_byid` to retrieve the actual pointer to the allocated memory region. Once we have the returned region, we check if it belongs to the current process. If it does not, we print "SEGMENTATION FAULT" and return.
- If we pass the two validity checks and ensure that the region belongs to the current process, we proceed to call the function `enlist_vm_freerg_list`. This function adds the returned region node to the process's free region list.
- By following these steps, we can successfully free the memory region.

3.2.3 Read and write

- **Get page function**

1. **Code**

```
int pg_getpage(struct mm_struct *mm, int pgn, int *fpn, struct pcb_t *caller)
{
    uint32_t pte = mm->pgd[pgn];
    if (!PAGING_PAGE_PRESENT(pte)) {
        return -1;
    }
    if (GETBIT(pte,30,31))
    { /* Page is not online, make it actively living */
        {
            /*Find victim page */

            int vicpgn, swpfpn;

            /*CHANGING FROM EXTRACTING BIT 8 -> 13 TO BIT 5 -> 25 */
            int tgtfpn = GETVAL(pte, PAGING_PTE_FPN_MASK, PAGING_PTE_FPN_LOBIT); //
    The target frame storing our variable

            find_victim_page(caller->mm, &vicpgn);
            uint32_t vicframe = GETVAL(mm->pgd[vicpgn], PAGING_PTE_FPN_MASK,
    PAGING_PTE_FPN_LOBIT);

            /* Get free frame in MEMSWP to store the value. */
            MEMPHY_get_freefp(caller->active_mswp, &swpfpn);

            /* Do swap frame from MEMRAM to MEMSWP and vice versa*/

            /* Copy victim frame to swap */
            __swap_cp_page(caller->mram, vicframe, caller->active_mswp, swpfpn);

```

```
28              /* Copy target frame from swap to mem */
29              __swap_cp_page(caller->active_mswp, tgtfpn, caller->mram, vicframe);
30
31              /* Update page table for the victim page = swap frame*/
32              pte_set_swap(&mm->pgd[vicpgn], 0, swpfpn);
33              pte_set_fpn(&pte, vicframe);
34
35              /* Update its online status of the target page
36               * Set the frame number of the victim page to the index which the process
      need to use */
37
38              enlist_pgn_node(&caller->mm->fifo_pgn, pgn);
39
40              /* Put the tgt frame to the swap free frame list*/
41              MEMPHY_put_freefp(caller->active_mswp, tgtfpn);
42    }
43  }
44  *fpn = GETVAL(pte, PAGING_PTE_FPN_MASK, PAGING_PTE_FPN_LOBIT);
45  return 0;
46 }
47
```

2. **Explanation**
   - To begin, we first check the present bit of the page table entry corresponding to the given page number. If the page table entry is not present, indicating that it does not belong to the current process, we return -1 as it accesses an invalid region.
   - On the other hand, if the page is present, we proceed to check its swap status by examining the swap bit of the page table entry (pte). If the page is not currently in RAM, we need to make it actively living.
   - First, we locate the target frame in the SWAP space that is mapped to our page number. After finding a victim frame in RAM, we search for a new free frame in the SWAP space and write the contents of the victim frame to this new frame. At the same time, we copy the content of our target frame to the space previously occupied by the victim frame. The space that initially stored our target frame is now freed up so we put it to the active SWAP's free frame list. It is important to set the respective frame number for the pte associated with our required page (now available in RAM), and set swap status for the victim page's pte. Finally, enqueue the handled page into the process's page list (`fifo_pgn`).

- **Read value and write value functions**

1. **Code**

```
1 //Read value function in mm-vm.c
2 int pg_getval(struct mm_struct *mm, int addr, BYTE *data, struct pcb_t *caller)
3 //Write value function in mm-vm.c
4 int pg_setval(struct mm_struct *mm, int addr, BYTE value, struct pcb_t *caller)
```

```
5
```

2. **Explanation**
   - These functions are used for the purpose of reading and writing values to a specified offset, acting as helper functions for our read and write functions (read and write in a memory region). Within these functions, we begin by checking if the `pg_getpage` function returns a negative value, which means our read or write action results in an invalid page access.
   - Otherwise, we proceed to call the `MEMPHY_read` or `MEMPHY_write` function, depending on our required operation, to read from or write to the given address and offset in RAM.

3.2.4 Results and Conclusion

- **2048B RAM**

1. **Output RAM status**

```
Time slot   0
ld_routine
        Loaded a process at input/proc/p0s, PID: 1 PRIO: 130
Time slot   1
        CPU 1: Dispatched process  1
Time slot   2
--------------------------------- RAM STATUS ---------------------------------
Ram size: 2048 byte
Free list in ram: 2 -> 3 -> 4 -> 5 -> 6 -> 7
------------------------------------------------------------------------------
        Loaded a process at input/proc/s3, PID: 2 PRIO: 39
        CPU 3: Dispatched process  2
        CPU 1: Put process  1 to run queue
Time slot   3
     CPU 1: Dispatched process  1
--------------------------------- RAM STATUS ---------------------------------
Ram size: 2048 byte
Free list in ram: 4 -> 5 -> 6 -> 7
------------------------------------------------------------------------------
        Loaded a process at input/proc/m1s, PID: 3 PRIO: 15
Time slot   4
        CPU 0: Dispatched process  3
--------------------------------- RAM STATUS ---------------------------------
Ram size: 2048 byte
Free list in ram: 6 -> 7
------------------------------------------------------------------------------
        CPU 3: Put process  2 to run queue
        CPU 3: Dispatched process  2
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
--------------------------------- RAM STATUS ---------------------------------
Ram size: 2048 byte
```

```
Free list in ram: 6 -> 7
------------------------------------------------------------------------
Time slot   5
-------------------------------- RAM STATUS --------------------------------
Ram size: 2048 byte
Free list in ram: 6 -> 7
------------------------------------------------------------------------
        Loaded a process at input/proc/s2, PID: 4 PRIO: 120
        CPU 2: Dispatched process  4
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  3
-------------------------------- RAM STATUS --------------------------------
Ram size: 2048 byte
Free list in ram: 6 -> 7
------------------------------------------------------------------------
write region=1 offset=20 value=100
Process ID: 1
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Time slot   6
        CPU 3: Put process  2 to run queue
Time slot   7
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
        CPU 3: Dispatched process  2
read region=1 offset=20 value=100
Process ID: 1
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
-------------------------------- RAM STATUS --------------------------------
Ram size: 2048 byte
Free list in ram: 6 -> 7
------------------------------------------------------------------------
        Loaded a process at input/proc/m0s, PID: 5 PRIO: 120
        CPU 2: Put process  4 to run queue
        CPU 2: Dispatched process  5
write region=2 offset=20 value=102
        CPU 0: Put process  3 to run queue
-------------------------------- RAM STATUS --------------------------------
Ram size: 2048 byte
Free list in ram:
------------------------------------------------------------------------
Time slot   8
```

```
Process ID: 1
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
        CPU 0: Dispatched process  3
--------------------------------- RAM STATUS ---------------------------------
Ram size: 2048 byte
Free list in ram:
------------------------------------------------------------------------------
        Loaded a process at input/proc/p1s, PID: 6 PRIO: 15
        CPU 3: Put process  2 to run queue
--------------------------------- RAM STATUS ---------------------------------
Ram size: 2048 byte
Free list in ram:
------------------------------------------------------------------------------
        CPU 3: Dispatched process  6
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  2
--------------------------------- RAM STATUS ---------------------------------
Ram size: 2048 byte
Free list in ram:
------------------------------------------------------------------------------
Time slot   9
Time slot  10
        CPU 2: Put process  5 to run queue
        CPU 2: Dispatched process  4
        CPU 0: Processed  3 has finished
        CPU 0: Dispatched process  5
--------------------------------- RAM STATUS ---------------------------------
Ram size: 2048 byte
Free list in ram: 4 -> 5
------------------------------------------------------------------------------
        Loaded a process at input/proc/s0, PID: 7 PRIO: 38
Time slot  11
--------------------------------- RAM STATUS ---------------------------------
Ram size: 2048 byte
Free list in ram: 4 -> 5
------------------------------------------------------------------------------
        CPU 3: Put process  6 to run queue
        CPU 3: Dispatched process  6
        CPU 1: Put process  2 to run queue
        CPU 1: Dispatched process  7
        CPU 2: Put process  4 to run queue
        CPU 2: Dispatched process  2
        CPU 0: Put process  5 to run queue
        CPU 0: Dispatched process  4
Time slot  12
```

```
        CPU 3: Put process  6 to run queue
        CPU 3: Dispatched process  6
Time slot  13
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
        CPU 2: Put process  2 to run queue
        CPU 2: Dispatched process  2
Time slot  14
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  5
write region=1 offset=20 value=25
Process ID: 5
print_pgtbl: 0 - 512
00000000: 80000007
00000004: 80000006
        CPU 3: Put process  6 to run queue
        CPU 3: Dispatched process  6
        CPU 1: Put process  7 to run queue
write region=2 offset=1000 value=1
Process ID: 5
print_pgtbl: 0 - 512
00000000: 80000007
00000004: 80000006
SEGMENTATION FAULT: invalid page access
        CPU 1: Dispatched process  7
Time slot  15
        Loaded a process at input/proc/s1, PID: 8 PRIO: 0
Time slot  16
        CPU 2: Put process  2 to run queue
        CPU 2: Dispatched process  8
        CPU 0: Processed  5 has finished
        CPU 0: Dispatched process  2
        CPU 3: Put process  6 to run queue
        CPU 3: Dispatched process  6
Time slot  17
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
        CPU 2: Put process  8 to run queue
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
        CPU 2: Dispatched process  8
Time slot  18
        CPU 3: Put process  6 to run queue
        CPU 3: Dispatched process  6
Time slot  19
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
        CPU 3: Processed  6 has finished
        CPU 2: Put process  8 to run queue
```

```
        CPU 2: Dispatched process  8
        CPU 3: Dispatched process  4
Time slot  20
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  1
read region=2 offset=20 value=102
Process ID: 1
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Time slot  21
        CPU 3: Put process  4 to run queue
Time slot  22
        CPU 3: Dispatched process  4
write region=3 offset=20 value=103
Process ID: 1
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
        CPU 2: Put process  8 to run queue
        CPU 2: Dispatched process  8
        CPU 2: Processed  8 has finished
        CPU 0: Processed  1 has finished
        CPU 0 stopped
        CPU 2 stopped
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
Time slot  23
        CPU 3: Put process  4 to run queue
        CPU 3: Dispatched process  4
Time slot  24
Time slot  25
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
        CPU 3: Put process  4 to run queue
        CPU 1: Processed  7 has finished
        CPU 1 stopped
        CPU 3: Dispatched process  4
Time slot  26
Time slot  27
        CPU 3: Processed  4 has finished
```

```
229         CPU 3 stopped
230
```

2. **Explanation**

p0s	m1s	m0s
1 10 calc alloc 300 0 alloc 300 4 free 0 alloc 100 1 write 100 1 20 read 1 20 20 write 102 2 20 read 2 20 20 write 103 3 20 read 3 20 20 calc free 4 calc	1 6 alloc 300 0 alloc 100 1 free 0 alloc 100 2 free 2 free 1	1 6 alloc 300 0 alloc 100 1 free 0 alloc 100 2 write 25 1 20 write 1 2 1000

Figure 3.7: *Only three process p0s, m1s, m0s require RAM allocation.*

- During time slot 2, CPU 1 executes the second instruction of p0s (Alloc 300), which involves allocating the first two frames in RAM for p0s. These frames correspond to addresses 0-300 in RAM.
- At time slot 3, CPU 1 continues to execute the third instruction of p0s, allocating an additional 300 bytes in RAM for p0s. This allocation corresponds to addresses 512-812 in RAM.
- At time slot 4, CP0 begins executing the first instruction of process 3. In this instruction, process 3 (m1s) requests 300 bytes in RAM. As a result, frames 4 and 5 in RAM are assigned to this process, covering addresses 1024-1324 in RAM.
- At time slot 4, CP0 begins executing the first instruction of process 3. In this instruction, process 3 (m1s) requests 300 bytes in RAM. As a result, frames 4 and 5 in RAM are assigned to this process, covering addresses 1024-1324 in RAM.
- At time slot 7, after being dispatched by CPU2, process 5 (m0s) calls alloc 300. Consequently, the last two available frames in RAM are allocated for this process. At this point, there are no more free frames remaining in RAM.
- In time slot 8, CPU2 proceeds with the second instruction of process 5 (alloc 300) and allocates the remaining space of frame 7 for this process.
- Following the requirement of the assignment, when a process finishes all of its instructions, we return all frames associated with that process back to physical memory. However, when the free function is called during the process execution, only the corresponding pages are added to the process's free region list. The freed frames are retained for future allocations by the same process.

– During time slots 4 - 8, CPU0 dispatches process 3 (m0s), and m0s calls free in the address saved in register number 0. However, the CPU does not return the frames to RAM as the process has not yet finished. In a subsequent time slot, CP0 completes the last instruction of process m0s, and we observe that all of its frames (frame 4 and 5) are returned to RAM.

– It is worth mentioning that at time slot 14, process m0s attempts to write to an invalid region in RAM, one that does not belong to the process. Consequently, we have printed out the error message "SEGMENTATION FAULT: invalid page access."

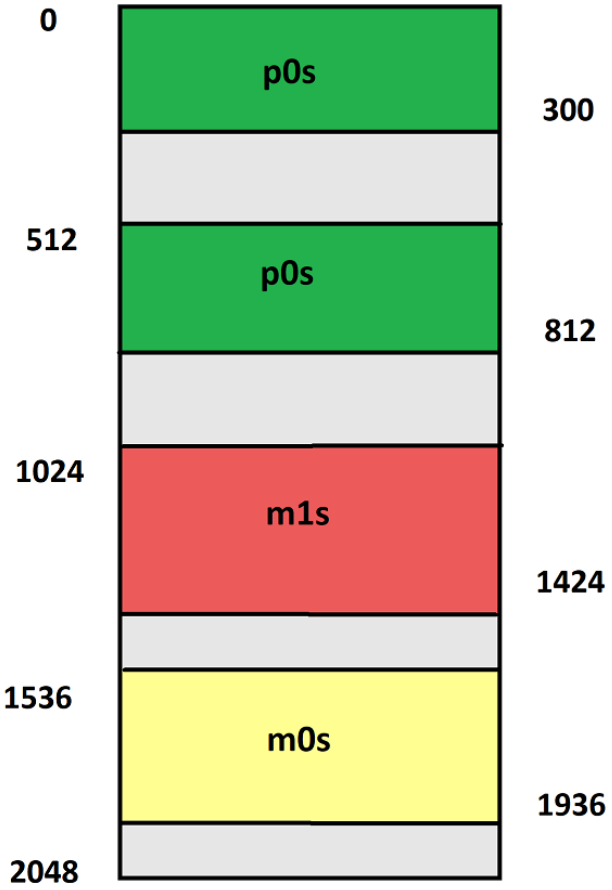

Figure 3.8: *RAM status in our demonstration .*

- **4096B RAM**

1. **Output RAM status**

```
Time slot   0
ld_routine
```

```
        Loaded a process at input/proc/p0s, PID: 1 PRIO: 130
        CPU 1: Dispatched process  1
Time slot   1
Time slot   2
--------------------------------- RAM STATUS ---------------------------------
Ram size: 4096 byte
Free list in ram: 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14
    -> 15
-----------------------------------------------------------------------------
        Loaded a process at input/proc/s3, PID: 2 PRIO: 39
        CPU 3: Dispatched process  2
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
--------------------------------- RAM STATUS ---------------------------------
Ram size: 4096 byte
Free list in ram: 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15
-----------------------------------------------------------------------------
Time slot   3
        Loaded a process at input/proc/m1s, PID: 3 PRIO: 15
--------------------------------- RAM STATUS ---------------------------------
Ram size: 4096 byte
Free list in ram: 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15
-----------------------------------------------------------------------------
Time slot   4
        CPU 2: Dispatched process  3
--------------------------------- RAM STATUS ---------------------------------
Ram size: 4096 byte
Free list in ram: 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15
-----------------------------------------------------------------------------
        CPU 3: Put process  2 to run queue
        CPU 3: Dispatched process  2
--------------------------------- RAM STATUS ---------------------------------
Ram size: 4096 byte
Free list in ram: 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15
-----------------------------------------------------------------------------
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
--------------------------------- RAM STATUS ---------------------------------
Ram size: 4096 byte
Free list in ram: 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15
-----------------------------------------------------------------------------
Time slot   5
        Loaded a process at input/proc/s2, PID: 4 PRIO: 120
Time slot   6
write region=1 offset=20 value=100
        CPU 2: Put process  3 to run queue
        CPU 2: Dispatched process  3
--------------------------------- RAM STATUS ---------------------------------
Ram size: 4096 byte
```

```
Free list in ram: 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15
------------------------------------------------------------------------
Process ID: 1
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
        CPU 0: Dispatched process  4
        CPU 3: Put process  2 to run queue
        CPU 3: Dispatched process  2
-------------------------------- RAM STATUS --------------------------------
Ram size: 4096 byte
Free list in ram: 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15
------------------------------------------------------------------------
Time slot   7
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
read region=1 offset=20 value=100
Process ID: 1
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
        Loaded a process at input/proc/m0s, PID: 5 PRIO: 120
        CPU 2: Put process  3 to run queue
        CPU 2: Dispatched process  3
-------------------------------- RAM STATUS --------------------------------
Ram size: 4096 byte
Free list in ram: 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15
------------------------------------------------------------------------
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  5
Time slot   8
-------------------------------- RAM STATUS --------------------------------
Ram size: 4096 byte
Free list in ram: 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15
------------------------------------------------------------------------
write region=2 offset=20 value=102
Process ID: 1
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
        Loaded a process at input/proc/p1s, PID: 6 PRIO: 15
        CPU 3: Put process  2 to run queue
        CPU 3: Dispatched process  6
```

```
        CPU 1: Put process  1 to run queue
-------------------------------- RAM STATUS --------------------------------
Ram size: 4096 byte
Free list in ram: 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15
--------------------------------------------------------------------------
Time slot   9
        CPU 1: Dispatched process  2
-------------------------------- RAM STATUS --------------------------------
Ram size: 4096 byte
Free list in ram: 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15
--------------------------------------------------------------------------
        CPU 2: Processed  3 has finished
        CPU 0: Put process  5 to run queue
        CPU 0: Dispatched process  5
-------------------------------- RAM STATUS --------------------------------
Ram size: 4096 byte
Free list in ram: 4 -> 5 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15
--------------------------------------------------------------------------
        CPU 2: Dispatched process  4
Time slot  10
        Loaded a process at input/proc/s0, PID: 7 PRIO: 38
        CPU 3: Put process  6 to run queue
        CPU 3: Dispatched process  6
Time slot  11
        CPU 1: Put process  2 to run queue
        CPU 1: Dispatched process  7
-------------------------------- RAM STATUS --------------------------------
Ram size: 4096 byte
Free list in ram: 4 -> 5 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15
--------------------------------------------------------------------------
        CPU 2: Put process  4 to run queue
        CPU 2: Dispatched process  2
Time slot  12
        CPU 0: Put process  5 to run queue
        CPU 0: Dispatched process  4
        CPU 3: Put process  6 to run queue
        CPU 3: Dispatched process  6
Time slot  13
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
        CPU 2: Put process  2 to run queue
        CPU 2: Dispatched process  2
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  5
write region=1 offset=20 value=25
Process ID: 5
print_pgtbl: 0 - 512
00000000: 80000007
00000004: 80000006
```

```
Time slot   14
        CPU 3: Put process  6 to run queue
        CPU 3: Dispatched process  6
write region=2 offset=1000 value=1
Process ID: 5
print_pgtbl: 0 - 512
00000000: 80000007
00000004: 80000006
SEGMENTATION FAULT: invalid page access
Time slot   15
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
        Loaded a process at input/proc/s1, PID: 8 PRIO: 0
        CPU 2: Put process  2 to run queue
        CPU 2: Dispatched process  8
Time slot   16
        CPU 0: Processed  5 has finished
        CPU 0: Dispatched process  2
        CPU 3: Put process  6 to run queue
        CPU 3: Dispatched process  6
Time slot   17
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
        CPU 2: Put process  8 to run queue
        CPU 2: Dispatched process  8
Time slot   18
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
        CPU 3: Put process  6 to run queue
        CPU 3: Dispatched process  6
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
Time slot   19
        CPU 3: Processed  6 has finished
        CPU 3: Dispatched process  4
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot   20
        CPU 2: Put process  8 to run queue
        CPU 2: Dispatched process  8
Time slot   21
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  1
read region=2 offset=20 value=102
Process ID: 1
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
```

```
00000012: 80000002
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
        CPU 3: Put process  4 to run queue
        CPU 3: Dispatched process  4
Time slot  22
        CPU 2: Put process  8 to run queue
        CPU 2: Dispatched process  8
write region=3 offset=20 value=103
Process ID: 1
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
        CPU 2: Processed  8 has finished
Time slot  23
        CPU 2 stopped
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
        CPU 0: Processed  1 has finished
        CPU 0 stopped
Time slot  24
        CPU 3: Put process  4 to run queue
        CPU 3: Dispatched process  4
Time slot  25
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
        CPU 3: Put process  4 to run queue
        CPU 3: Dispatched process  4
        CPU 1: Processed  7 has finished
        CPU 1 stopped
Time slot  26
Time slot  27
        CPU 3: Processed  4 has finished
        CPU 3 stopped

```

2. **Explanation**

   As we have analyzed in the case using 2048B RAM, the three processes that request memory allocation can fit within the available physical space. Consequently, in the case using 4096B RAM, where the input remains the same, we have a larger physical address space, and there is no need to utilize the SWAP memory.

Chapter 4

Put it all together

4.1 Theoretical basis – Synchronization

4.1.1 The need for synchronization

Synchronization is the process of coordinating the execution of multiple threads, processes or devices to ensure that they operate correctly and avoid conflicts. The need for synchronization arises in situations where multiple entities are accessing a shared resource, such as a file, database, network connection, or hardware device.

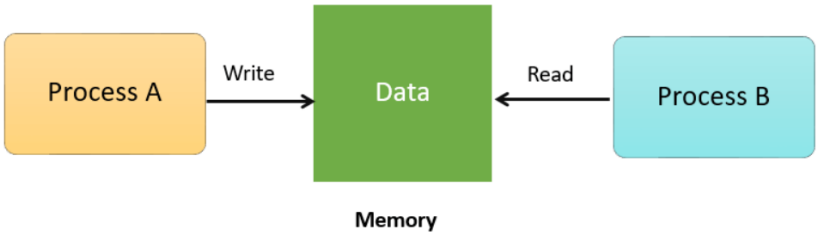

Figure 4.1: *An example of readers writers problem.*

Without synchronization, multiple entities can access the shared resource at the same time, leading to race conditions, deadlocks, and other errors. For example, if two threads attempt to modify the same variable at the same time, one thread may overwrite the changes made by the other thread, leading to inconsistent results. Similarly, if multiple threads attempt to read from a file at the same time, the file may become corrupted.

Synchronization ensures that only one entity can access the shared resource at a time, preventing race conditions and other errors. This is typically achieved using synchronization primitives such as locks, semaphores, and monitors, which allow threads or processes to request access to the shared resource and wait until it becomes available before proceeding.

4.1.2 Errors in synchronization

Race condition

A race condition is a type of concurrency bug that occurs when two or more threads or processes access a shared resource or variable in an unexpected order, leading to unpredictable and incorrect behavior. A race condition occurs when the correctness of the program's output or behavior depends on the relative timing or interleaving of the operations performed by different threads or processes.

For example, consider two threads, T_1 and T_2, that access a shared variable, x. If T_1 reads the value of x, then T_2 reads the value of x, then T_1 writes a new value to x, and finally, T_2 writes a new value to x, the final value of x may be different from what is expected, depending on the relative timing of the thread execution.

Race conditions can be challenging to detect and reproduce, as they are often timing-dependent and non-deterministic. They can lead to a wide range of errors, including data corruption, crashes, and security vulnerabilities.

To prevent race conditions, synchronization techniques such as locks, semaphores, and monitors can be used to ensure that only one thread or process can access a shared resource at a time. Alternatively, techniques such as atomic operations and message passing can be used to eliminate the need for shared resources altogether.

Deadlock

In operating systems, a deadlock occurs when two or more processes are waiting for each other to release resources or locks, resulting in a circular dependency that prevents any of the processes from making progress. Deadlocks can occur in systems that use shared resources such as memory, CPU time, and I/O devices.

A deadlock typically arises when each process holds a resource and is also waiting for another resource held by another process, leading to a "deadly embrace" or a "deadly lock." For example, consider two processes, A and B, where A is holding a lock on a resource X and waiting for a lock on a resource Y, and B is holding a lock on resource Y and waiting for a lock on resource X. In this scenario, both processes can proceed, and the system is deadlocked.

Deadlocks can be challenging to detect and resolve, as they often occur intermittently and can be difficult to reproduce. Deadlocks can cause the system to become unresponsive, as the processes involved in the deadlock cannot make progress, leading to a loss of system resources and a degraded user experience.

To prevent deadlocks, operating systems use various techniques, including resource preemption, where the system forcibly takes a resource from a process to allow other processes to proceed, and resource allocation ordering, where resources are allocated in a predetermined order to prevent circular dependencies. Other techniques include timeout mechanisms and deadlock detection and recovery algorithms. By using these techniques, operating systems can ensure that deadlocks are prevented or resolved quickly to maintain system stability and responsiveness.

Starvation

Starvation occurs when a thread or process is prevented from accessing a shared resource indefinitely, even though the resource is available. This can happen when a high-priority thread monopolizes a shared resource or when a low-priority thread is constantly preempted by higher-priority threads.

4.1.3 Critical section

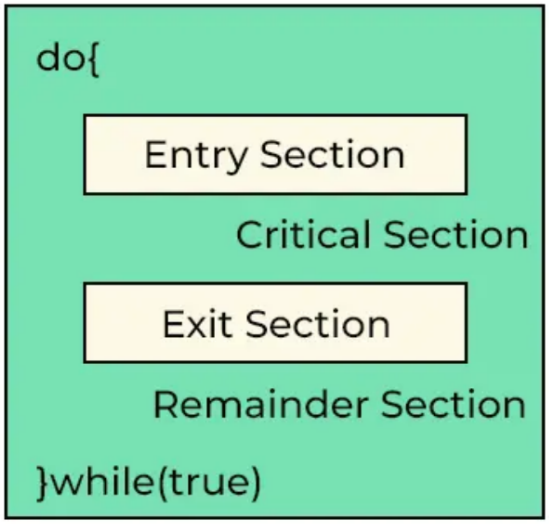

Figure 4.2: *Critical section.*

The critical section is a code segment that restricts access to shared resources to only one process at a time. This section contains the resources that can be accessed by multiple processes. By utilizing critical sections, the issue of concurrent access to shared resources can be effectively resolved. Essentially, when multiple programs attempt to access and modify a single shared data, only one process will be granted access at any given time, while the remaining processes will have to wait for the data to become available. Removing the critical section could result in inconsistent outcomes if all processes attempt to execute simultaneously.

There are multiple of solutions to solve critical section problem:

- Peterson's solution
- Monitors
- Mutex locks
- Semaphore

For this assignment, our attention will be directed towards Mutex locks as a means of managing shared data. A Mutex lock is a synchronization mechanism utilized to regulate access to a resource located in the critical

section. The approach involves the implementation of a lock on the critical section. When a process enters from the entry section, the lock is enabled, and it is disabled when the process exits from the exit section.

Question 5

What will happen if the synchronization is not handled in your simple OS? Illustrate by example the problem of your simple OS if you have any.

Answer

Without synchronization, it can lead to several problems such as deadlocks, race conditions, starvation, and priority inversion. All of these issues can cause system instability, application crashes, and data loss. Therefore, it is essential to handle synchronization properly in an operating system to ensure the correct and reliable execution of programs and system functions.

An illustrative example can be presented by utilizing the custom input parameter while disabling the synchronization mechanism for multi-level queue in our operating system. In this scenario, one process will be assigned to both of the two CPUs for execution, effectively allowing the process to be processed simultaneously in two separate locations with identical instructions.

At time slot 8, CPU 1 failed to reinsert process 1 into the system, but the execution of process 1 was already completed. Consequently, CPU 1 encounters an error when attempting to add a non-existent (a freed) process to the multi-level queue.

- **Input**

```
6 2 2
1048576 16777216 0 0 0
1 p1s 0
2 p1s 1

```

- **Output**

```
Time slot   0
ld_routine
Time slot   1
        Loaded a process at input/proc/p1s, PID: 1 PRIO: 0
Time slot   2
        CPU 0: Dispatched process  1
        CPU 1: Dispatched process  1
        Loaded a process at input/proc/p1s, PID: 2 PRIO: 0
Time slot   3
Time slot   4
Time slot   5
Time slot   6
Time slot   7
        CPU 0: Processed  1 has finished
```

```
        CPU 0: Dispatched process  2
Time slot   8
Segmentation fault

```