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

OPERATING SYSTEMS (CO2017)

Operating Systems Assignment - Semester 222

Simple Operating System

Advisor:	Dr. Le Thanh Van	
Student:	Dinh Le Dung	2152483
	Tran Nhat Minh	2152177
	Dang Bao Tin	2152313

HO CHI MINH CITY, MAY 2023

Member list & Workload of Group No.35

No.	Full name	Student ID	Task	Contribution
1	Dinh Le Dung	2152483	Implementing C program for memory management section and answer questions	100%
2	Tran Nhat Minh	2152177	Implementing C program for scheduling and memory management sections	100%
3	Dang Bao Tin	2152313	Managing progress, collecting tasks, implementing and testing OS program for scheduler and memory management, writing report	100%

Contents

1 Introduction

Purpose: The purpose of this major project is to simulate the main components of a simple operating system, such as a scheduler, synchronization, and activities related to physical and virtual memory.

Content: Students will practice three main parts: the scheduler, synchronization, and the mechanism for allocating virtual memory to physical memory.

Outcome: After completing this major project, students will gain an understanding of the principles behind the operation of a simple operating system. They will grasp the roles and significance of the main modules within an operating system, as well as how they function.

In this major project, students will delve into the intricate workings of a basic operating system. They will have the opportunity to simulate key components, including a scheduler, synchronization mechanisms, and memory management.

- The first aspect of the project focuses on the scheduler. Students will gain hands-on experience in designing and implementing a scheduling algorithm. We will implementing MLQ strategies and analyze their impact on system performance. By working on this component, we will develop a deeper understanding of how processes are scheduled and managed within an operating system.

- The second part of the project centers around synchronization. Students will delve into the challenges of coordinating concurrent processes and ensuring data integrity. They will tackle issues like race conditions, deadlocks, and resource allocation. Through practical exercises, they will implement synchronization primitives such as semaphores, mutexes, and condition variables to prevent conflicts and maintain order in the system. This section of the project will enable students to appreciate the complexity involved in managing shared resources within an operating system.

- The final component of the project revolves around memory management. Students will explore the interaction between physical and virtual memory and learn about techniques like paging, segmentation, and demand paging. They will implement algorithms for virtual-to-physical address translation and study the trade-offs between memory utilization and performance. By engaging with this aspect of the project, students will gain insights into how an operating system efficiently manages memory resources and supports multitasking.

By completing this major project, students will not only acquire practical skills in operating system development but also develop a deeper understanding of the fundamental principles governing these complex systems. They will comprehend the roles and significance of the main modules within an operating system, and how they collaborate to provide a seamless user experience. Ultimately, this project will equip students with a solid foundation to further explore and contribute to the field of operating systems.

2 Scheduler and Multi-Level Queue

2.1 Mechanism

In this assignment, scheduler will be implemented by using priority policy. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. In this project, 0 will stands for ***highest*** priority and higher number will indicates ***lower*** priority.

Let take the following set of processes as an example, assumed to have arrived at time 0 in the order P_1, P_2, ..., P_5, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Figure 1: *Example of Priority process*

Using priority scheduling, we would schedule these processes according to the following Gantt chart:

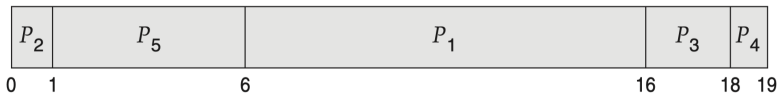


Figure 2: *Gantt chart of the Example*

A major problem with priority scheduling algorithms is **indefinit blocking**, or **starvation**. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low-priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.

In this assignment, a solution to the problem of indefinite blockage of low-priority processes is implemented via Multi-Level Queue (MLQ) policy. The system contains `MAX_PRIO` priority levels.

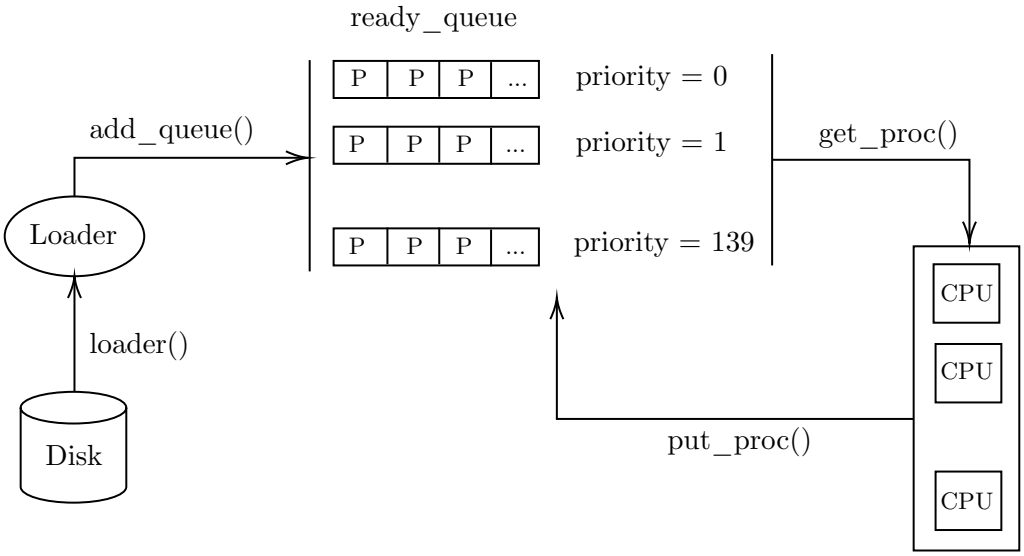


Figure 3: *MLQ policy for scheduler*

The number traversed steps of `ready_queue` list is a limited fixed formulated number based on the priority, or `slot = MAX_PRIO 0 prio`, where `MAX_PRIO` will be pre-defined before using. Since then, each queue has only fixed slot to use the CPU and when it is used up, the system must change the resource to the other process in the next queue. This can avoid the unwanted starvation that now every process can be executed.

2.2 Implementation

In this assignment, we will implement `enqueue()` and `dequeue()` in the source `queue.c`. First, we will set the `#define MAX_PRIO 140` to limit the number of priority queues to 140 queues. Each `ready_queue` has maximum of MAX_QUEUE_SIZE process entries that has been pre-defined.

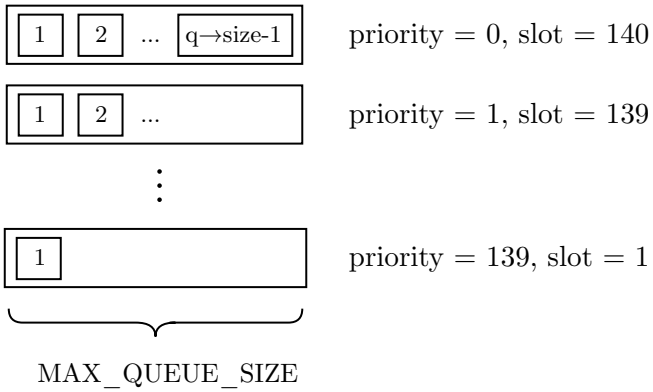


Figure 4: *MLQ ready_ queues structures*

```
void enqueue(struct queue_t *q, struct pcb_t *proc)
{
    /* TODO: put a new process to queue [q] */
    pthread_mutex_lock(&q->lock);
    if (!q->size) {
        q->proc[0] = proc;
        q->size++;
    }
    else {
        int it = 0;
        while (it < q->size) {
            if (q->proc[it]->priority > proc->priority) {
                for (int i = q->size - 1; i > it; i--)
                    q->proc[i] = q->proc[i - 1];
                q->proc[it] = proc;
                break;
            }
            it++;
        }
        if (it == q->size) {
            q->proc[it] = proc;
            q->size++;
        }
    }
    pthread_mutex_unlock(&q->lock);
}
```

In `enqueue()`, the function adds a new process to a MLQ queue (`struct queue_t`) based on its priority. The code acquires a lock to ensure exclusive access to the queue. If the queue is empty, the process is added at the beginning. If the queue is not empty, the code finds the appropriate position to insert the process based on its priority. It shifts existing processes to the right to make room for the new process, if necessary. The process is inserted at the correct position or at the end if it has the lowest priority. The lock is released to allow other threads to access the queue.

The function utilizes a mutex (`q->lock`) to ensure thread safety during enqueueing. This enqueue function ensures that processes are inserted into the queue in ascending order of priority. In normal priority policy, the higher the priority, the closer the process will be to the front of the queue. This implementation guarantees that processes with higher priorities get executed before processes with lower priorities. The MLQ policy will be implemented in `get_mlq_proc()` in `sched.c` source.

```
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
```

```
         * */
        struct pcb_t *proc = NULL;
        pthread_mutex_lock(&q->lock);
        if (q->proc[0] != NULL)
        {
                proc = q->proc[0];
                for (int i = 0; i < q->size - 1; i++)
                        q->proc[i] = q->proc[i + 1];
                q->size--;
        }
        pthread_mutex_unlock(&q->lock);

        return proc;
}
```

In `dequeue()`, we keep the traditional function of policy, since in `enqueue()`, we always re-arrange associating with its priority, so we just need to get its head process and shift all the list queue to the left and decrease the `q->size`.

```
#ifdef MLQ_SCHED
/*
 *  Stateful design for routine calling
 *  based on the priority and our MLQ policy
 *  We implement stateful here using transition technique
 *  State representation   prio = 0 .. MAX_PRIO, curr_slot = 0..(MAX_PRIO - prio)
 */
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 i;
        for (i = 0; i < MAX_PRIO; i++) {
                if (mlq_ready_queue[i].size != 0 && prioSlot[i] < MAX_PRIO - i) {
                        proc = dequeue(&mlq_ready_queue[i]);
                        prioSlot[i]++;
                        break;
                }
        }
        if (i == MAX_PRIO) {
                for (int j = 0; j < MAX_PRIO; j++)
                        prioSlot[j] = 0;
                for (int j = 0; j < MAX_PRIO; j++) {
                        if (mlq_ready_queue[j].size != 0 && prioSlot[j] < MAX_PRIO - j) {
```

```
                                proc = dequeue(&mlq_ready_queue[j]);
                                prioSlot[j]++;
                                break;
                        }
                }
        }
        pthread_mutex_unlock(&queue_lock);
        return proc;
}
```

As mentioned above, with the `#MLQ_SCHED` is enabled, the CPU will fetch the process base on MLQ policy. The function will loop through the MLQ with MAX_PRIO number of `ready_queue`. Each level of priority `ready_queue` will have a certain time of fetching its processes. If the slot reaches (MAX_PRIO − *priority*), then the CPU is blocked from keep fetching that `ready_queue` and move on fetching in the lower-priority `ready_queue`.

After an amount of time, all the queue will be full of slots, so the function will reset all the slot entries back to 0 and continue fetching from the highest priority down to the lowest one.

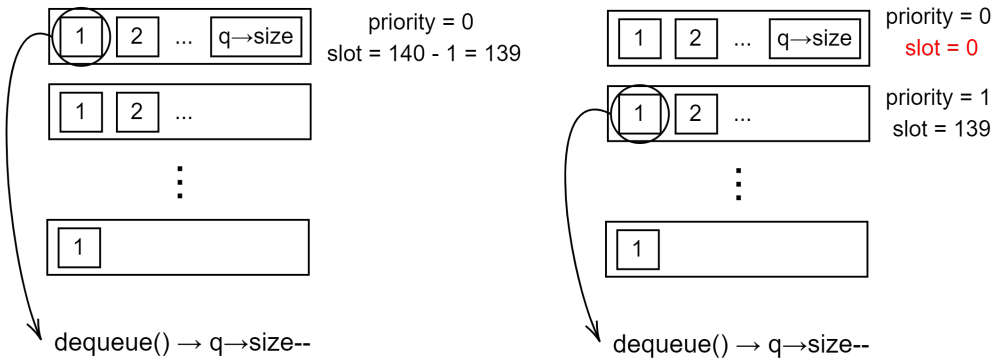


Figure 5: *Getting process order in MLQ policy*

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

After time quantum, the process which is not yet finished will be put bact to its corresponding `ready_queue`. And the CPU will continues until all the process is done.

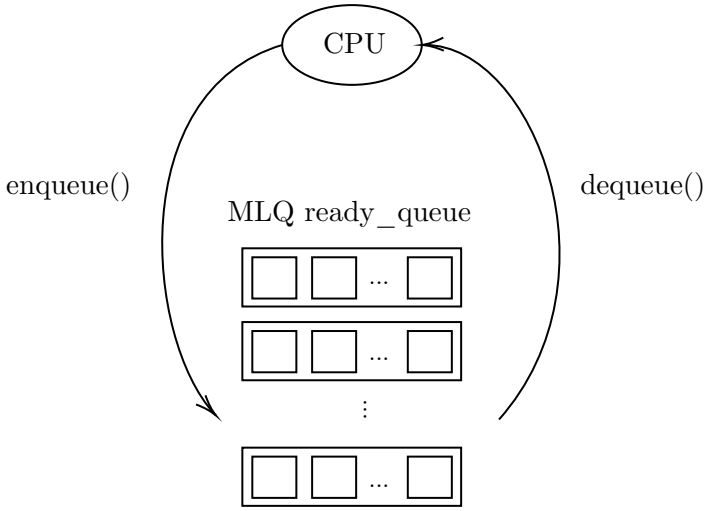


Figure 6: *The implementation of scheduler*

2.3 Results

Before testing, we will enable two macros `#MLQ_SCHED` and `#MM_FIXED_MEMSZ` for this scheduling test section, further testing will be included in the next section. And then we compile our code by the command instruction in the terminal:

```
make clean
make all
```

We run the test by simply type the following command in the terminal:

```
./os sched_0
./os sched_1
```

Testcase sched_0:

The input of `sched_0` is given below:

```
2 1 2
0 s0 137
4 s1 138
```

Because we want to test the scheduling using MLQ policy, we have added a third argument in each instructions stands for the priority of the process. And after we run the input, we can obtain the output in the terminal below:

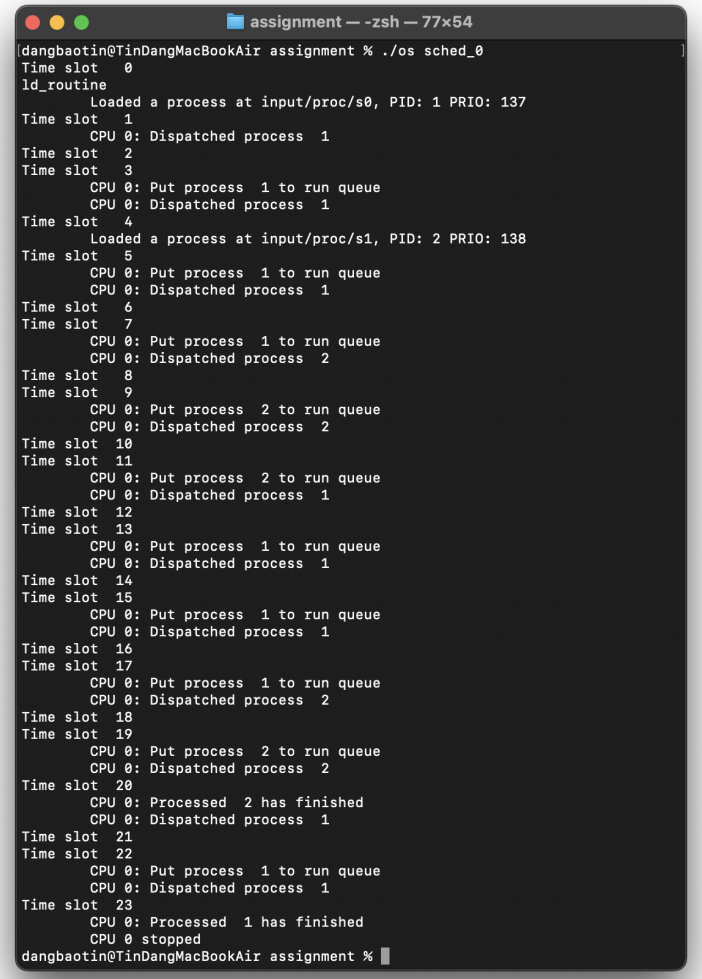


Figure 7: *Sched_0 output*

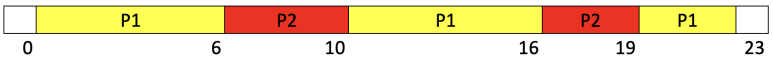


Figure 8: *Gantt chart Sched_0 output*

As for `sched0`, `process_1` with priority 137 and a maximum slot of 3 will be loaded into the corresponding `ready_queue` (`PRIO` 137). At time slot 1, CPU takes out process 1 for execution. After running for 2 seconds (time stamp), at time plot 3, process 3 will be put back into the ready queue used to store it (`ready_queue PRIO` 137), and `slot_137` (count index) is increased by 1. CPU similarly continues to process with process 1 at time slots 3 and 5. After running at time slot 7, `slot_137` reaches its peak at 3. The CPU immediately returns process 1 to its own ready

queue and waits for other lower-priority processes to run. There is process 2 waiting in sentence 138. The CPU will load process 2 instead and keep running as it did with process 1, but with a maximum slot of 2. Up to time slot 11 when process 2 reaches its maximum slot. Now there aren't any processes that are waiting in lower-priority queues, so we reset all index counts to 0, and the CPU will start at the highest priority, which is 137. And running the same way we did until there is no process in all of the ready queue.

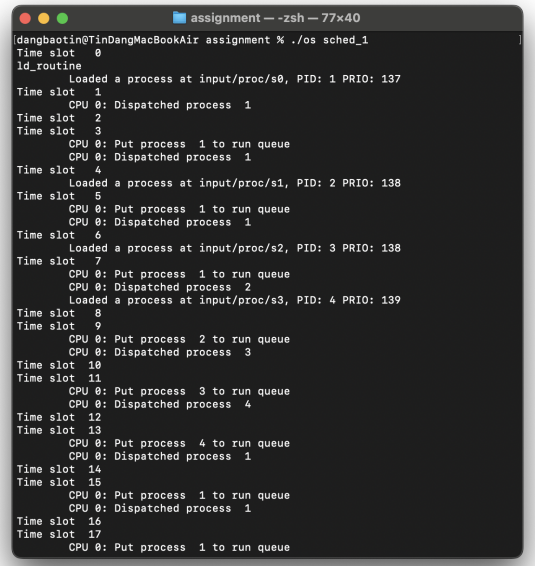
Testcase sched_1:

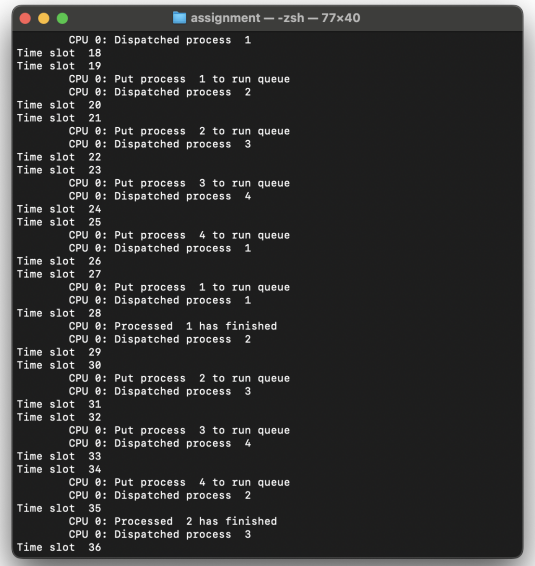
The input of `sched_0` is given below:

```
2 1 4
0 s0 137
4 s1 138
6 s2 138
7 s3 139
```

And the content for each process is given in folder `proc`, for example the content of the first process `s0` is:

```
12 15
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
```





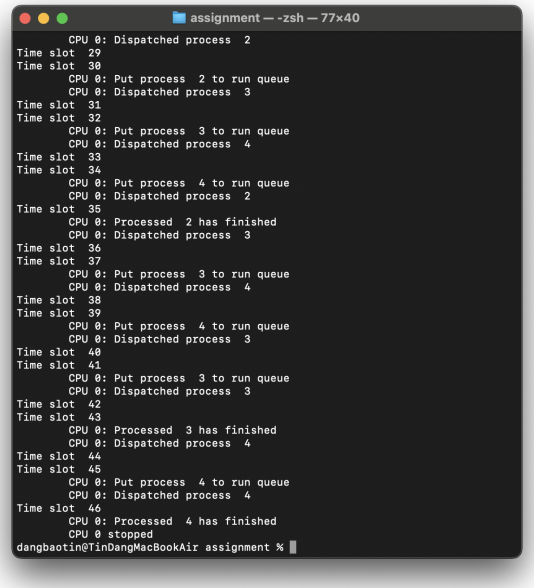


Figure 9: *Sched_1 output*

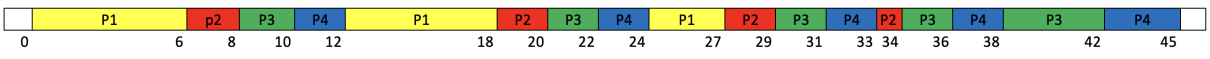


Figure 10: *Gantt chart Sched_1 output*

As for `sched1`, by implying the algorithm was used in `shed0`, `shed1` is likely the same as what we get in `shed0`. The difference is that process 2's priority and process 3's priority are the same. If there are many processes in a ready queue, we apply round-robin arrangement. In this example, when CPU begins working on ready queue with priority 138 at time slot 7, CPU dispatches process 2 at the same time process 3 is loading ready queue, so after process 2 runs in 2 seconds (time stamp), according to round robin, process 2 will be put back and process 3 taken out instead of keeping using process 2.

2.4 Answer the question

Question: What is the advantage of using priority queue in comparison with other scheduling algorithms you have learned?
Answer: When compared to other scheduling methods, the use of a priority queue gives the following benefits:

1. Flexibility: We can use the priority queue to apply different scheduling algorithms to distinct processes.

2. Select tasks effectively: Priority queues provide effective selection of the task with the highest priority. The task with the highest priority can be accessed quickly with a time complexity of just $O(1)$ and executed without the need to traverse the entire list of tasks. This leads to faster response times for high-priority jobs.

3. We do not need to wait too long for a certain process in the ready queue to be executed, even though it may have low priority. As there is a time slice to limit the execution time, low-priority progress can run after a certain time.

4. Priority-oriented: Processes with higher priority get executed first, while ones with lower priority get executed later.

3 Memory Management

3.1 Mechanism and implementation

3.1.1 ALLOC instruction

For `ALLOC` operation, it will first checks if the macro `#MM_PAGING` is defined or not. In this project, we will mainly focus on memory paging, so we let this macro enable for the whole assignment. When the macro `#MM_PAGING` is defined, to have the free space for memory allocation, it will call `get_free_vmrg_area()`.

- If a free region is found, it updates the memory management structures (`symrgtbl`) of the caller process with the allocated region's start and end addresses. It also assigns the start address to the `alloc_addr` pointer, and returns 0 to indicate a successful allocation.
- If a free region of the desired size is not available, it first retrieves the current VMA by using `get_vma_by_num()`. It then tries to increase the limit of the virtual memory area and calls `get_free_vmrg_area()` again to search for a free memory region with the desired size. If a free region is found (the returned value is 0), it updates the memory management structures with the appropriate start and end addresses, in which start address is set to `old_sbrk`, and the end address is set to `old_sbrk + size`. If both attempts fail to allocate memory, the function returns -1 to indicate an allocation failure.

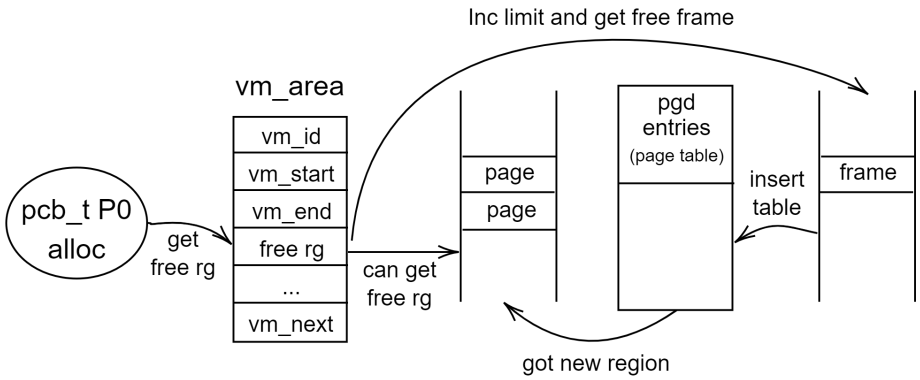


Figure 11: *Alloc instruction behaviour*

```
1 int inc_vma_limit(struct pcb_t *caller, int vmaid, int inc_sz)
2 {
```

```
  struct vm_rg_struct *newrg = malloc(sizeof(struct vm_rg_struct));
  int inc_amt = PAGING_PAGE_ALIGNSZ(inc_sz);
  int incnumpage = inc_amt / PAGING_PAGESZ;
  struct vm_rg_struct *area = get_vm_area_node_at_brk(caller, vmaid, inc_sz, inc_amt);
  struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);

  int old_end = cur_vma->vm_end;

  /*Validate overlap of obtained region */
  if (validate_overlap_vm_area(caller, vmaid, area->rg_start, area->rg_end) < 0)
    return -1; /*Overlap and failed allocation */

  /* The obtained vm area (only)
   * now will be alloc real ram region */
  cur_vma->vm_end += inc_amt;

  if (enlist_vm_rg_node(&caller->mm->mmap->vm_freerg_list, newrg) == 0) {}

  if (vm_map_ram(caller, area->rg_start, area->rg_end,
                 old_end, incnumpage, newrg) < 0)
    return -1; /* Map the memory to MEMRAM */

  return 0;
}
```

3.1.2 FREE instruction

With the macro `#MM_PAGING` is defined, when the process calls *free* instruction, it first checks the validity of the provided `rgid`. It then retrieving the start and end addresses of a specific memory region identified by the `rgid` from the symbol region table and assigning those values to the corresponding fields in the `rgnode` struct. This is done to store the information of the memory region that will be enlisted as an obsoleted (freed) region in the `vm_freerg_list`.

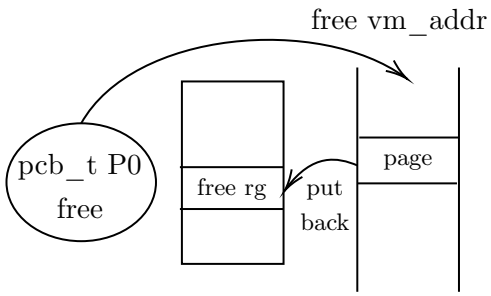


Figure 12: *Free instruction behaviour*

3.1.3 READ/WRITE instruction

The CPU first retrieves the page table entry (`pte`) from the page directory (`pgd`) in the memory region using the given page number. In order to read/write, the page frame must be presented in RAM, so we must checks if the page is not present (`PAGING_PAGE_SWAPPED_PRESENT(pte)` or it is in SWAP memory), indicating that the page is not currently in RAM memory and needs to be loaded.

- If the page is not present, we first check if there is a free frame available in the physical memory (`MEMPHY_get_freefp()` function) and assigns it to the `swpfpn` variable. We then find the victim page by calling the `find_victim_page()` function. We retrieve the page table entry (`vicpte`) and the frame number (`vicfpn`) from the victim page table entry. We performs the swap operation by using the `__swap_cp_page()` function, where it copies the victim frame to the swap space and the target frame to the memory. We updates the page table entries of the victim page and the target page, setting the appropriate frame numbers and swap flags. We updates the free frame list and used frame list of the active memory swap space (`caller->active_mswp`). After that, we enlists the page number (`pgn`) in the FIFO list of page numbers (`caller->mm->fifo_pgn`).
- If the page is already present, it simply retrieves the frame number from the page table entry and stores it in the `fpn` variable.

Assume that the page retrieval is successful, we then calculate the physical address (`phyaddr`) by combining the frame number (`fpn`) and the offset (`off`). This calculation is simply performed by shifting the `fpn` value left by `PAGING_ADDR_FPN_LOBIT` bits and adding the off value.

Finally, base on the operation, with the calculated physical address (`phyaddr`), the corresponding function on physical memory will be executed (`MEMPHY_read/MEMPHY_write`). As for the `READ` operation, the retrieved data is stored in the data variable.

(a) *If page is in RAM* **(b)** *If page is not in RAM*

Figure 13: *READ/WRITE instruction operation*

Now, we will implement the `pg_getpage()` function provide for this operation.

```c
int pg_getpage(struct mm_struct *mm, int pgn, int *fpn, struct pcb_t *caller)
{
  uint32_t pte = mm->pgd[pgn];

  if (PAGING_PAGE_SWAPPED_PRESENT(pte))
  { /* Page is not online, make it actively living */
    int vicpgn, swpfpn;
    int vicfpn;
    uint32_t vicpte;

    int tgtfpn = GETVAL(pte, PAGING_PTE_SWPOFF_MASK, PAGING_SWPFPN_OFFSET);

    // Find free frame in RAM
    if (MEMPHY_get_freefp(caller->mram, &swpfpn) == 0) {
      __swap_cp_page(caller->active_mswp, tgtfpn, caller->mram, swpfpn);

      pte_set_fpn(&caller->mm->pgd[pgn], tgtfpn);

      struct framephy_struct *fp = caller->mram->used_fp_list;
      struct framephy_struct *newnode = malloc(sizeof(struct framephy_struct));

      /* Create new node with value fpn */
      newnode->fpn = swpfpn;
      newnode->fp_next = fp;
      caller->mram->used_fp_list = newnode;
    }
    else {

      /* TODO: Play with your paging theory here */
      /* Find victim page */
      if (find_victim_page(caller->mm, &vicpgn) != 0) {
        printf("ERROR: Cannot find vitim page  -  pg_getpage()\n");
        return -1;
      }

      /* Get free frame in MEMSWP */
      if (MEMPHY_get_freefp(caller->active_mswp, &swpfpn) != 0) {
        printf("ERROR: Cannot find free frame in RAM  -  pg_getpage\n");
        return -1;
      }

      vicpte = caller->mm->pgd[vicpgn];  // victim pte from victim page number
      vicfpn = GETVAL(vicpte, PAGING_PTE_FPN_MASK, 0);


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

```
48        __swap_cp_page(caller->mram, vicfpn, caller->active_mswp, swpfpn);
49        /* Copy target frame from swap to mem */
50        __swap_cp_page(caller->active_mswp, tgtfpn, caller->mram, vicfpn);
51
52        /* Update page table */
53        pte_set_swap(&caller->mm->pgd[vicpgn], 0, swpfpn);
54
55        /* Update its online status of the target page */
56        pte_set_fpn(&caller->mm->pgd[pgn], vicfpn);
57
58        enlist_pgn_node(&caller->mm->fifo_pgn, pgn);
59      }
60    }
61
62    *fpn = GETVAL(mm->pgd[pgn], PAGING_PTE_FPN_MASK, 0);
63
64    return 0;
65  }
```

3.2 Results

Before testing, we will disable macro `#MM_FIXED_MEMSZ` for this memory test section, since the size of RAM is not fixed any longer but can be varied in real life. And then we compile our code by the command instruction in the terminal:

```
make clean
make all
```

We have create a new test case only for this memory test section, we will allow one CPU and will print out the RAM status to see the operation of the memory management.

The input of `os_new` is:

```
2 1 1
2048 16777216 0 0 0
1 p00s 130
```

And the content of proc `p00s` is:

```
1 9
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
```

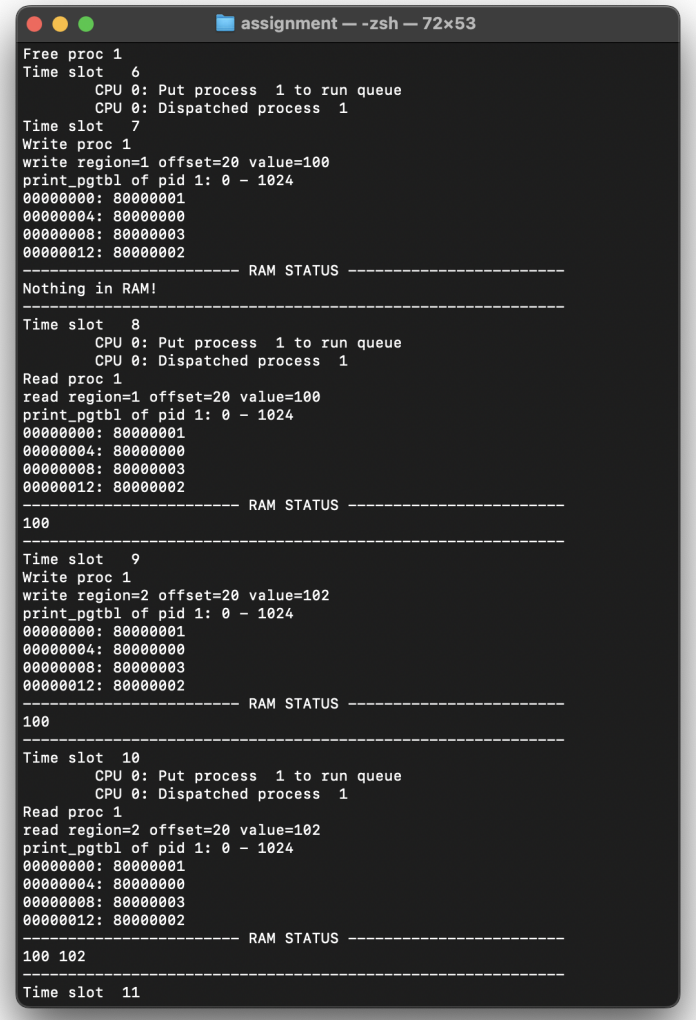


Figure 14: *OS_ new output*

Because our source code print out the RAM status before write to it so the RAM of the first write instruction is empty. Next, when the process reads from RAM, it prints out the RAM status again, this time, the RAM contains value 100 which is being writen from the previous `write`

instruction. The CPU continues to execute the next `write` instruction of the process, and when prints out the RAM when `read` again at time slot 10, the RAM contains 102, which is the same as our expectation the CPU to write. And the CPU will continues until the process is done.

3.3 Answer the question

Question: 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: While we imply multiple memory segments, this give us the following benefits:

1. Simplify such complex operations on memory by working on separated blocks known as memory segments, as segmentation provides a way to easily implement object-oriented programs.
2. Allow programmers to partition their programs into modules that operate independently of one another.
3. Segmentation allows the use of a 16-bit register to give an addressing capability of 1 MB; without segmentation, we would require a 20-bit register.
4. It is possible to increase the memory size of code, data, and stack segments beyond 64 KB by allowing more than one segment for each area.
5. Segmentation makes it possible to write programs that are position-independent or dynamically relocatable.

Question: What will happen if we divide the address to more than 2-levels in the paging memory management system?
Answer: If there are more than 2-level paging memory, we will both have advantages and disadvantages.
Advantage:

1. Reducing the memory overhead associated with the page table. This is because each level contains fewer entries; the more levels we divide, the less memory is required to store the page table.
2. Faster page table lookup: With a smaller number of entries per level, it takes less time to perform a page table lookup. As a result, system performance is overall faster.
3. Flexibility: With more than 3-level paging, there is greater flexibility in terms of how the memory space is organized. This can be especially useful in systems with varying memory requirements, as it allows the page table to be adjusted to accommodate changing needs.

Disadvantage:

1. All of the page tables are kept in memory. As a result, getting the physical address of the page frame requires more than one memory access, one for each level required. Although it speeds up lookups in theory, extra memory references to access address translation can slow down a program in memory by a factor of two or more, which is a considerable drawback.
2. Increased complexity: Multilevel paging adds complexity to the memory management system, which can make it more difficult to design, implement, and debug.
3. Increased overhead: Although multilevel paging can reduce the memory overhead associated with the page table, it can also increase the overhead associated with page table lookups. This is because each level must be traversed to find the desired page table entry.
4. Fragmentation: Multilevel paging can lead to fragmentation of the memory space, which can reduce overall system performance. This is because the page table entries may not be contiguous, which can result in additional overhead when accessing memory.

Conclusion:
It is necessary to consider how many table levels we need for a particular problem so that we can efficiently run our program. Preventing using so many tables unless it may end up with complexity for the programmer and fragmentation. In this case, 2-level paging is ideal for students like us.

Question: What is the advantage and disadvantage of segmentation with paging?
Answer:
Advantage:

1. The main advantage of segmented paging is the reduction in memory usage. Since it allocates fixed-size pages, it does not cause external fragmentation. It makes memory allocations simpler.
2. Reduces page table size as it is limited to the number of segments. Therefore, reducing memory requirements.
3. Data can be scattered all over physical memory.
4. Allows for demand paging and pre-paging. If we need memory, we just swap the pages that are least likely to be used, rather than the whole segment.

Disadvantage:

1. The problem of internal fragmentation is still not solved completely. There are occasions where internal fragmentation can occur, but the probability is lower. There are delays when we access memory. Also, the hardware required for the implementation of the paged segmentation technique is complex.

2. The segmented paging technique also requires more hardware resources.
3. If the memory required is less than or not divisible by the page size, there will be a RAM frame that is not put to total use.
4. The translation is sequential, which increases the memory access time, leading to higher complexity. Page tables have to be contiguous to be stored in memory.

4 Linking up together

4.1 Synchronization

Now, we combine scheduler and memory management to form a complete OS. For simplicity, we will only use 1 virtual memory area in this assignment. Figure below shows the complete organization of the OS memory management.

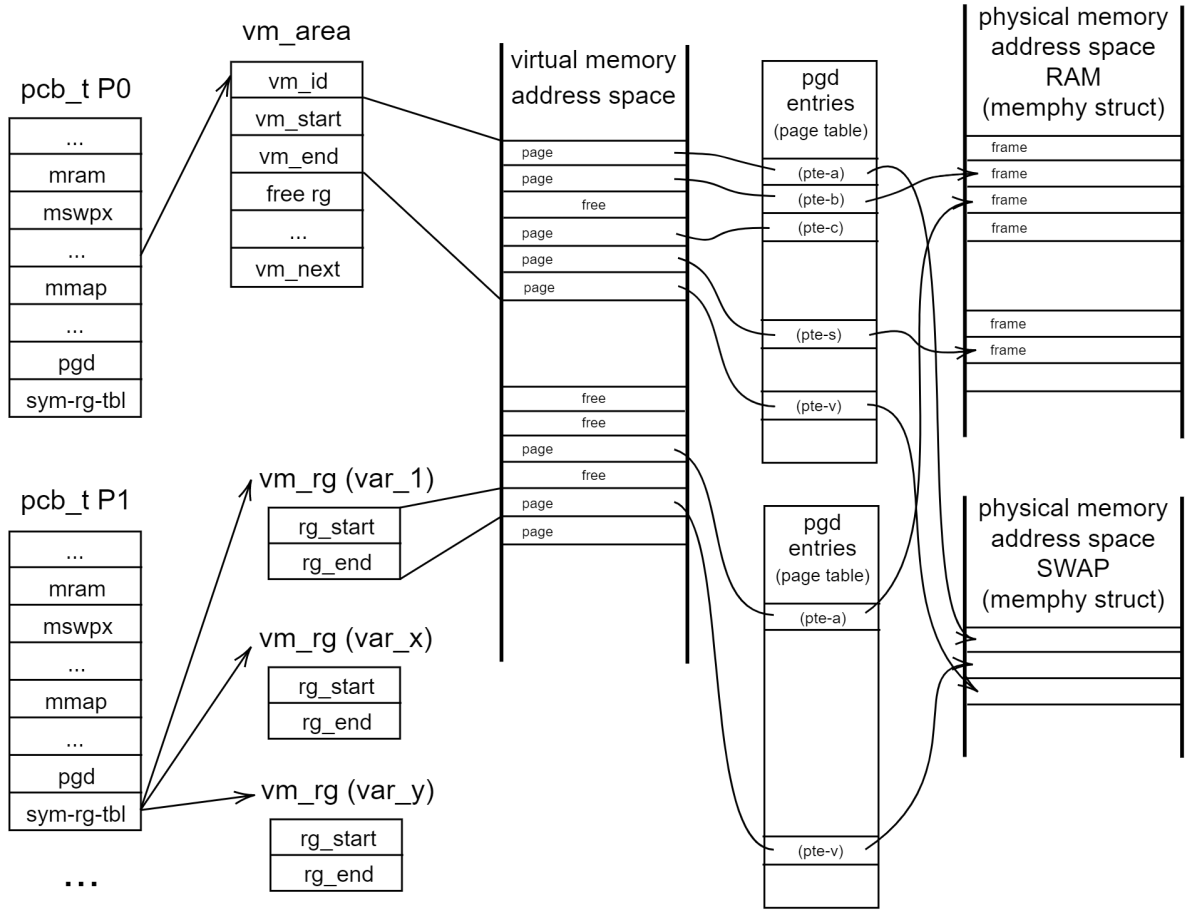


Figure 15: *The operation related to virtual memory in the assignment*

Since the OS runs on multiple processors, it is possible that share resources could be concurrently accessed by more than one process at a time. To solve this, we will initialize and use a `pthread_mutex_t lock_memphy` to protect the shared physic memory, in this assignment that is RAM and SWAP, and put it in 2 functions below in source `mm-memphy.c`:

```
1 int MEMPHY_get_freefp(struct memphy_struct *mp, int *retfpn)
2 {
3    struct framephy_struct *fp = mp->free_fp_list;
```

```
4
5     if (fp == NULL)
6       return -1;
7
8     pthread_mutex_lock(&mp->lock_memphy);
9
10    *retfpn = fp->fpn;
11    mp->free_fp_list = fp->fp_next;
12
13
14    /* MEMPHY is iteratively used up until its exhausted
15     * No garbage collector acting then it not been released
16     */
17    free(fp);
18    pthread_mutex_unlock(&mp->lock_memphy);
19    return 0;
20  }
21
22  int MEMPHY_put_freefp(struct memphy_struct *mp, int fpn)
23  {
24    pthread_mutex_lock(&mp->lock_memphy);
25    struct framephy_struct *fp = mp->free_fp_list;
26    struct framephy_struct *newnode = malloc(sizeof(struct framephy_struct));
27
28    /* Create new node with value fpn */
29    newnode->fpn = fpn;
30    newnode->fp_next = fp;
31    mp->free_fp_list = newnode;
32
33    pthread_mutex_unlock(&mp->lock_memphy);
34    return 0;
35  }
```

4.2 Test OS

Now, we will test serveral cases for our OS operation to finalized our operating systems.

Test os_0_mlq_paging:

```
6 2 4
1048576 16777216 0 0 0
0 p0s 0
2 p1s 15
```

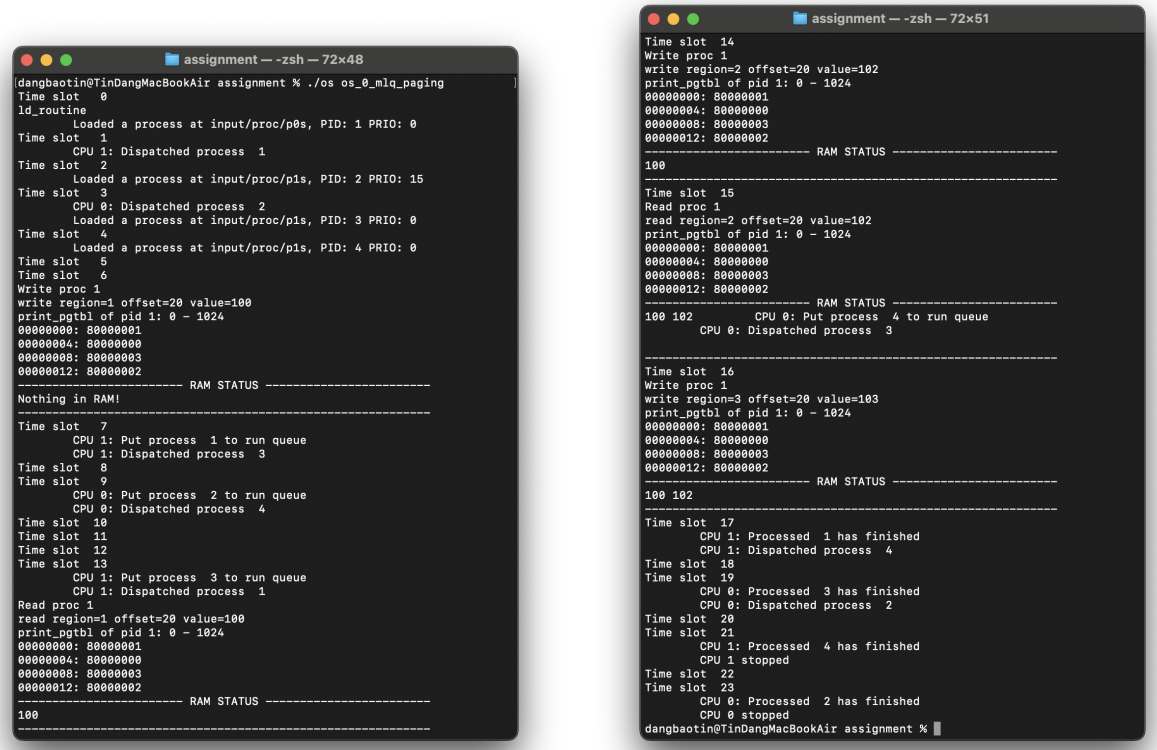


Figure 16: *OS_0_mlq_paging output*

Test os_1_mlq_paging:

```
2 4 8
1048576 16777216 0 0 0
1 p0s 130
2 s3 39
4 m1s 15
6 s2 120
7 m0s 120
9 p1s 15
11 s0 38
16 s1 0
```

Figure 17: *OS_1_mlq_paging output*

Test os_1_mlq_paging_small_1K:

```
2 4 8
2048 16777216 0 0 0
1 p0s 130
2 s3 39
4 m1s 15
6 s2 120
7 m0s 120
9 p1s 15
11 s0 38
16 s1 0
```

Figure 18: *OS_1_mlq_paging_small_1K output*

In general, all three test have yield results that performing well and following our implementation. The CPU fetches the process using MLQ priority queue policy and the memory management in RAM and SWAP in the result matches the mechanism that we explained and implemented in the previous section.

4.3 Answer the question

Question: 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:

1. It is specially needed in a multi-process system when multiple processes are running together, and more than one processes try to gain access to the same shared resource or data at the same time. This can lead to the inconsistency of shared data. So the change made by one process not necessarily reflected when other processes accessed the same shared data. To avoid this type of inconsistency of data, the processes need to be synchronized with each other.
   Example: Under race conditions, when a process calculates data and writes it to a variable that another process has just read and is in the process of calculating, this creates differences in results that, on a larger level, will affect significantly.

2. In terms of memory allocation, if synchronization is not handled, two processes will access the same page for allocation, leading to a conflict.

3. On the other hand, the timer will not work correctly without a mutex mechanism. That leads to the incrementation of timeslots, although tasks are not done.

References

[1] Silberschatz, A., Galvin, P., & Gagne, G. (1983). *Operating System Concepts*. http://ci.nii.ac.jp/ncid/BA35822586.

[2] Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. C. (2015). *Operating Systems: Three Easy Pieces*. https://dblp.uni-trier.de/db/journals/usenix-login/usenix-login42.html#Arpaci-Dusseau17.

[3] GeeksforGeeks. (2023). Multilevel Queue MLQ CPU Scheduling. *GeeksforGeeks*. https://www.geeksforgeeks.org/multilevel-queue-mlq-cpu-scheduling/.

[4] GeeksforGeeks. (2023a). Virtual Memory in Operating System. *GeeksforGeeks*. https://www.geeksforgeeks.org/virtual-memory-in-operating-system/.