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

PROJECT REPORT
OPERATING SYSTEMS (CO2017)

Simple Operating System

Class CC01 - Group 5

Võ Hoàng Long - 2053192
Nguyễn Phúc Toàn - 2153902
Phan Lê Khánh Trình - 2153902

HO CHI MINH CITY, DECEMBER 2023

Contents

1 Member list & Workload **2**

2 Introduction **3**
2.1 An overview . . . 3
2.2 Scheduler . . . 3
2.3 Virtual Memory . . . 4

3 Implementation **4**
3.1 Scheduler . . . 4
3.1.1 Scheduling Init . . . 4
3.1.2 Enqueue . . . 4
3.1.3 Dequeue . . . 5
3.1.4 Get process from multilevel queue . . . 5
3.2 Memory management . . . 6
3.2.1 Helper struct and function . . . 6
3.2.2 Get free frame from the RAM . . . 11
3.2.3 Map the frame to the virtual memory . . . 13
3.2.4 Free the register . . . 15
3.2.5 Write to register . . . 16
3.2.6 Get the frame in RAM . . . 17

4 Answering questions **19**
4.1 Scheduler . . . 19
4.2 Memory Management . . . 20
4.2.1 The virtual memory mapping in each process . . . 20
4.2.2 The system physical memory . . . 21
4.2.3 Paging-based address translation scheme . . . 22
4.3 Put It All Together . . . 23

5 Some testcase of achieved results **23**
5.1 Scheduling . . . 23
5.1.1 Input and output . . . 23
5.1.2 Grantt Diagram . . . 26
5.2 Memtest: Alloc, Read, Write, Free . . . 27
5.3 Testing race condition with memory . . . 40

6 Conclusion **43**

1 Member list & Workload

No.	Fullname	Student ID	Problems	Percentage of work
1	Võ Hoàng Long	2053192	Scheduler and report	30%
2	Nguyễn Phúc Toàn	2153902	Memmory	35%
3	Phan Lê Khánh Trình	2151268	Memmory	35%

2 Introduction

2.1 An overview

The operating system manages two virtual resources: the CPU(s) and the RAM. To accomplish this, it incorporates two core components: the Scheduler (and Dispatcher) and the Virtual Memory Engine (VME). The overall goal of this operating system implementation is to allow multiple user-created processes to effectively share and utilize the virtual computing resources.

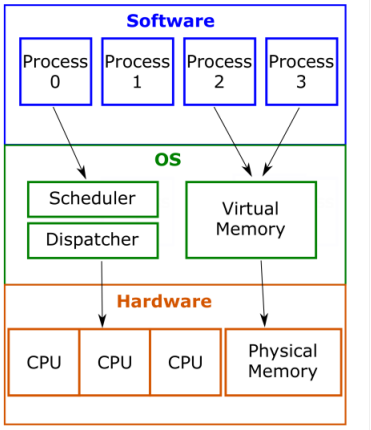

Figure 1. An overview of key modules in this assignment

2.2 Scheduler

The Scheduler and Dispatcher work together to determine which process is granted access to the CPU and when. The Scheduler makes decisions regarding process prioritization and allocation of CPU time, while the Dispatcher is responsible for actually transferring control to the selected process.

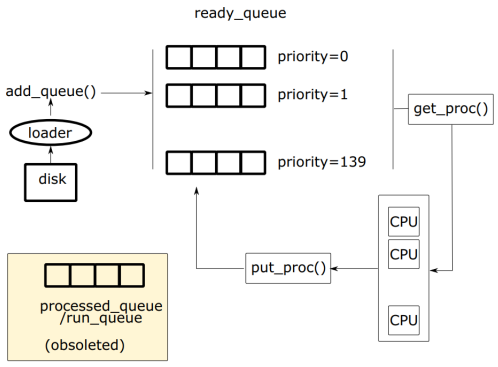

Figure 2. The operation of scheduler in the assignment

2.3 Virtual Memory

The Virtual Memory Engine serves as a crucial component for memory management. Its primary function is to isolate the memory space of each process from one another. While the physical RAM is shared among multiple processes, each process remains unaware of the existence of others. This is achieved by assigning each process its own virtual memory space, and the Virtual Memory Engine maps and translates the virtual addresses provided by processes to the corresponding physical addresses.

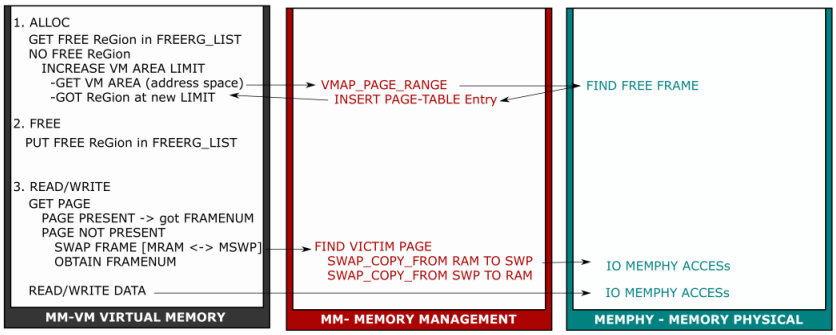

Figure 3. Memory system modules in the assignment

3 Implementation

3.1 Scheduler

3.1.1 Scheduling Init

This is a form of round robin scheduling with a fixed time quantum based on priority. The idea is to allocate a fixed time slot to each priority level -> avoid starvation by ensure the lower priority processes get a chance to execute with their allocated time slots. To avoid starvation, we add a fixed quantum based on priority (slot): give each level defined by **"slot = MAX_PRIO - prio"**. Because of the traversed step of ready queue (slot), Lower-priority processes might take longer to complete, but they are guaranteed some CPU time in each round.

```
struct queue_t {
    struct pcb_t * proc[MAX_QUEUE_SIZE];
    int size; // size of mlq queue
    int slot; // avoid starvation
    int curr_slot; // keep track of the # of process in each queue
};
```

3.1.2 Enqueue

The enqueue function is implemented by adding a process at the tail of the queue and increasing the size of the queue.

```
void enqueue(struct queue_t * q, struct pcb_t * proc) {
    /* TODO: put a new process to queue [q] */
    if(q->size == MAX_QUEUE_SIZE){
        fprintf(stderr, "Queue is full\n");
        return;
    }
    q->proc[q->size] = proc;
    q->size++;
}
```

3.1.3 Dequeue

The dequeue function is implemented by getting the first process out of the queue and reducing the size by 1.

```
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;
    int i;
    struct pcb_t * proc = q->proc[0];
    for (i = 0; i < q->size - 1; i++) {
        q->proc[i] = q->proc[i + 1];
    }
    q->size--;
    return proc;
}
```

3.1.4 Get process from multilevel queue

The below function is used to take a process from the multilevel queue with the mechanism described in the assignment. Find a process from the queue with the highest priority to the lowest with each queue having a limited access time. When the number of access times is all gone, then set all to default again.

```
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);
        unsigned long prio;
        int found = 0;
        for (prio = 0; prio < MAX_PRIO; prio++)
        {
                if (!empty(&mlq_ready_queue[prio]) &&
                ↪  (mlq_ready_queue[prio].count <
                ↪  mlq_ready_queue[prio].fixedSlot))
                {
                        proc = dequeue(&mlq_ready_queue[prio]);
                        mlq_ready_queue[prio].count++;
                        found = 1;
                        break;
                }
        }
        // set count to 0 if it do not call dequeue
        if (found == 0)
        {
                for (prio = 0; prio < MAX_PRIO; prio++)
                        mlq_ready_queue[prio].count = 0;
        }
        pthread_mutex_unlock(&queue_lock);
        return proc;
}
```

3.2 Memory management

3.2.1 Helper struct and function

- struct LRU_struct

This struct helps managing the paging algorithms.

```
struct LRU_struct
{
        struct LRU_struct *lru_next;
        struct LRU_struct *lru_pre;
        uint32_t *pte;
        int fpn;
};
```

- void Add_LRU_page(uint32_t *pte_add);

```
void Add_LRU_page(uint32_t *pte_add)
{
  struct LRU_struct *tmp = malloc(sizeof(struct LRU_struct));
  tmp->pte = pte_add;
  tmp->fpn = PAGING_PTE_FPN(*pte_add);

  if (lru_head == NULL)
  {
    lru_head = tmp;
    lru_tail = lru_head;
    tmp->lru_pre = NULL;
    tmp->lru_next = NULL;
  }
  else
  {
    struct LRU_struct *p = lru_head;
    int flag = 0;
    while (p != NULL)
    {
      if (p->fpn == tmp->fpn)
      {
        flag = 1;
        printf("FLAG 1");
        break;
      }
      p = p->lru_next;
    }
    if (flag == 1)
    {
      if (p == lru_head)
      {
        if (lru_head == lru_tail)
        {
          return;
        }
        lru_head = lru_head->lru_next;
        lru_head->lru_pre = NULL;
      }
      else if (p == lru_tail)
```

```
      {
        return;
      }
      else
      {
        p->lru_pre->lru_next = p->lru_next;
        p->lru_next->lru_pre = p->lru_pre;
      }

      lru_tail->lru_next = p;
      p->lru_pre = lru_tail;
      p->lru_next = NULL;
      lru_tail = p;
    }
    else
    {
      lru_tail->lru_next = tmp;
      tmp->lru_pre = lru_tail;
      tmp->lru_next = NULL;
      lru_tail = tmp;
      return;
    }
  }
}
```

- Update_LRU_lst(uint32_t *pte_rm)

```
void Update_LRU_lst(uint32_t *pte_rm)
{
  struct LRU_struct *p = lru_head;
  int fpn = PAGING_PTE_FPN(*pte_rm);
  while (p->lru_next != NULL)
  {
    if (p->fpn == fpn)
    {
      break;
    }
    p = p->lru_next;
  }
  if (p == lru_head)
```

```
  {
    if (lru_head == lru_tail)
    {
      return;
    }
    lru_head = lru_head->lru_next;
    lru_head->lru_pre = NULL;
  }
  else if (p == lru_tail)
  {
    return;
  }
  else
  {
    p->lru_pre->lru_next = p->lru_next;
    p->lru_next->lru_pre = p->lru_pre;

  }
  lru_tail->lru_next = p;
  p->lru_pre = lru_tail;
  p->lru_next = NULL;
  lru_tail = p;

}
```

- LRU_print_page()

This function print out the LRU list if they have been alloced in memory.

```
void LRU_print_page()
{
  struct LRU_struct *temp = lru_head;
  printf("=================LRU LIST=================\n");
  printf("FPN: \n");
  if (temp == NULL)
    printf("\nEMPTY page directory\n");
  else
  {
    while (temp != NULL)
    {
      printf("[%d]", temp->fpn);
```

```
      if (temp->lru_next != NULL)
      {
        printf(" -> ");
      }
      temp = temp->lru_next;
    }
    printf("\n========================================\n\n");
  }
}
```

- LRU_find_victim_page()

The function implements a LRU page replacement strategy for managing pages in memory. It checks if the linked list representing the LRU order is empty, returning an error code if so. If not empty, it extracts the head of the list, adjusts the linked list to remove that page, and returns that page.

```
uint32_t *LRU_find_victim_page()
{
  if (lru_head == NULL)
    return -1;
  struct LRU_struct *temp = lru_head;
  uint32_t *pte_res;
  pte_res = temp->pte;
  if (lru_head == lru_tail)
  {
    lru_head = lru_tail = NULL;
  }
  else
  {
    lru_head = lru_head->lru_next;

    if (lru_head != NULL)
    {
      lru_head->lru_pre = NULL;
    }

    temp->lru_next = NULL;
  }
  free(temp);
```

```
  return pte_res;
}
```

3.2.2 Get free frame from the RAM

This function gets a list of free frames from the RAM, if it can not find them, it will use the mechanism of page replacement to take a frame from the ram (update the page table as well).

```
int alloc_pages_range(struct pcb_t *caller, int req_pgnum, struct
↪  framephy_struct **frm_lst)
{
#ifdef CHECK
  printf("alloc_pages_range\n");
#endif

  int pgit, fpn;
  // struct framephy_struct *newfp_str;

  struct framephy_struct *newfp_str = NULL;
  struct framephy_struct *temp;
  if ((caller->mram->maxsz / PAGING_PAGESZ) < req_pgnum)
  {
    printf("Process %d alloc error: not enough RAM space!\n", caller->pid);
    return -3000;
  }//...

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

      // Create new node
      struct framephy_struct *newnode = malloc(sizeof(struct
      ↪  framephy_struct));
      newnode->fpn = fpn;
      newnode->owner = caller->mm;

      if (newfp_str == NULL)
      {
        newfp_str = newnode;
```

```
      }
      else
      {
        struct framephy_struct *temp = newfp_str;
        while (temp->fp_next != NULL)
        {
          temp = temp->fp_next;
        }

        temp->fp_next = newnode;
      }
    }
    else  { // ERROR CODE of obtaining somes but not enough frames
      // swapping to get the page
      int swpfpn;
      uint32_t *vicpte;

      // get free frame in SWAP
      if (MEMPHY_get_freefp(caller->active_mswp, &swpfpn) < 0)
      {
        // not enough frame
        printf("Error -3000: Out of frame/n");
        return -3000;
      }

      // get victim_frame and victim_pte
      vicpte = LRU_find_victim_page(); // LRU function used

      // Get frame of victim_page
      int vicfpn = GETVAL(*vicpte, PAGING_PTE_FPN_MASK, PAGING_PTE_FPN_LOBIT);
#ifdef RAM_STATUS_DUMP
      printf("[Page Replacement]\tPID #%d:\tVictim:%d\tPTE:%08x\n",
      ↪  caller->pid, vicfpn, *vicpte);
#endif

      // Swap from RAM to SWAP
      __swap_cp_page(caller->mram, vicfpn, caller->active_mswp, swpfpn);

      //notice vicpte that is swapped*/
      pte_set_swap(vicpte, 0, swpfpn);
```

```
#ifdef RAM_STATUS_DUMP
      printf("[After Swap]\tPID #%d:\tVictim:%d\tPTE:%08x\n", caller->pid,
      ↪  swpfpn, *vicpte);
#endif
      //create new node
      struct framephy_struct *newnode = malloc(sizeof(struct
      ↪  framephy_struct));
      newnode->fpn = vicfpn;
      newnode->owner = caller->mm;

      // add frame into frame list (frm_lst)
      if (newfp_str == NULL){
        newfp_str = newnode;
      }
      else{
        struct framephy_struct *temp = newfp_str;
        while (temp->fp_next != NULL)
        {
          temp = temp->fp_next;
        }
        temp->fp_next = newnode;
      }
    }
  }
  *frm_lst = newfp_str;

  return 0;
}
```

3.2.3 Map the frame to the virtual memory

This function takes the last page number then increase and maps it to the frames of the free list that have been taken from the above function and update it to the page table.

```
int vmap_page_range(struct pcb_t *caller,
                    int addr,
                    int pgnum,
                    struct framephy_struct *frames,
                    struct vm_rg_struct *ret_rg)
```

```
{
#ifdef CHECK
  printf("vmap_page_range\n");
#endif

  struct framephy_struct *fpit = malloc(sizeof(struct framephy_struct));
  int pgit = 0;
  int pgn = PAGING_PGN(addr);

  ret_rg->rg_end = ret_rg->rg_start = addr; // at least the very first space
   ↪  is usable

  fpit = frames;

  /* Tracking for later page replacement activities (if needed)
   * Enqueue new usage page */

  // increase limit of rg_end
  ret_rg->rg_end = addr + pgnum * PAGING_PAGESZ;

  while (fpit != NULL && pgit < pgnum)
  {
    //owned frame
    fpit->owner = caller;

    pte_set_fpn(&(caller->mm->pgd[pgn + pgit]), fpit->fpn);

#ifdef RAM_STATUS_DUMP
    printf("[Page mapping]\tPID #%d:\tFrame:%d\tPTE:%08x\tPGN:%d\n",
     ↪  caller->pid, fpit->fpn, caller->mm->pgd[pgn + pgit], pgn + pgit);
#endif


  //add frame to global LRU
  Add_LRU_page(&(caller->mm->pgd[pgn + pgit]));
    fpit = fpit->fp_next;
    pgit++;
  }

  return 0;
```

```
}
```

3.2.4 Free the register

This function takes the region that is freed by the register and enlists its content to the free region list of the process.

```
int __free(struct pcb_t *caller, int vmaid, int rgid)
{
#ifdef TDBG
  printf("__free\n");
#endif
  struct vm_rg_struct *rgnode;

  if (rgid < 0 || rgid > PAGING_MAX_SYMTBL_SZ)
  {
  LRU_print_page();
    printf("Process %d free error: Invalid region\n", caller->pid);
    return -1;
  }

  /* TODO: Manage the collect freed region to freerg_list */

  rgnode = get_symrg_byid(caller->mm, rgid);
  if (rgnode->rg_start == rgnode->rg_end)
  {
  LRU_print_page();
    printf("Process %d FREE Error: Region wasn't alloc or was freed before\n",
    ↪  caller->pid);
    return -1;
  }
  struct vm_rg_struct *rgnode_temp = malloc(sizeof(struct vm_rg_struct));

  BYTE value;
  value = 0;
  for (int i = rgnode->rg_start; i < rgnode->rg_end; i++)
    pg_setval(caller->mm, i, value, caller);

  // Create new node for region
  rgnode_temp->rg_start = rgnode->rg_start;
  rgnode_temp->rg_end = rgnode->rg_end;
```

```
  rgnode->rg_start = rgnode->rg_end = 0;


  /*enlist the obsoleted memory region */
  enlist_vm_freerg_list(caller->mm, *rgnode_temp);
  return 0;
}
```

3.2.5 Write to register

```
int __write(struct pcb_t *caller, int vmaid, int rgid, int offset, BYTE value)
{
#ifdef TDBG
  printf("__write\n");
#endif
  struct vm_rg_struct *currg = get_symrg_byid(caller->mm, rgid);

  struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);

  if (currg == NULL || cur_vma == NULL) /* Invalid memory identify */
  {
    printf("Process %d write error: Invalid region\n", caller->pid);
    return -1;
  }

  /*------------------Bat dau phan lam----------------*/
  if (currg->rg_start >= currg->rg_end)
  {
    printf("Process %d write error: Region not found (freed or
    ↪  unintialized)\n", caller->pid);
  }
  else if (currg->rg_start + offset >= currg->rg_end || offset < 0)
  {
    printf("Process %d write error: Invalid offset when write!\n",
    ↪  caller->pid);
    return -1;
  }
  else
  {
```

```
    struct vm_rg_struct rgnode=*get_symrg_byid(caller->mm, rgid);
    int *alloc_addr = rgnode.rg_start;
    int current_pgn = PAGING_PGN(rgnode.rg_start);
    uint32_t *current_pte = caller->mm->pgd[current_pgn];

    Update_LRU_lst(&current_pte);
    pg_setval(caller->mm, currg->rg_start + offset, value, caller);
  }
  /*------------------Ket thuc phan lam---------------*/

  return 0;
}
```

3.2.6 Get the frame in RAM

This function gets the page table entry. If it is in ram, then take the frame out and end. If it is not in the ram, then check if the ram has any available frame, if there is an available frame then move data from the swap to the ram and return that frame of ram. If the ram is full now, we need to find a victim page and move the data in that frame to a free frame in the swap and move the data that we want from swap to the ram and return that frame of ram.

```
int pg_getpage(struct mm_struct *mm, int pgn, int *fpn, struct pcb_t *caller)
{
#ifdef TDBG
  printf("pg_getpage\n");
#endif
  // pthread_mutex_lock(&MEM_in_use);
  uint32_t pte = mm->pgd[pgn];
  if (!PAGING_PAGE_PRESENT(pte))
  { /* Page is not online, make it actively living */
    /* TODO: Play with your paging theory here */
    int fpn_temp = -1;
    if (MEMPHY_get_freefp(caller->mram, &fpn_temp) == 0)
    {

      // lay gia tri tgtfpn
      int tgtfpn = GETVAL(pte, GENMASK(20, 0), 5);

      // Copy frame from SWAP to RAM
      __swap_cp_page(caller->active_mswp, tgtfpn, caller->mram, fpn_temp);
```

```
      // Cap nhat gia tri pte
      pte_set_fpn(&mm->pgd[pgn], fpn_temp);

      // printf("DEBUG GIA: pte = %08x", mm->pgd[pgn]);
      Add_LRU_page(&mm->pgd[pgn]);
    }
    else
    {
      int tgtfpn = GETVAL(pte, GENMASK(20, 0), 5);
      // printf("DEBUG GIA: pte = %08x\n", mm->pgd[pgn]);
      int vicfpn, swpfpn;
      uint32_t *vicpte;
      /* Find pointer to pte of victim frame*/
      vicpte = LRU_find_victim_page();

      /* Variable for value of pte*/
      uint32_t vicpte_temp = *vicpte;
      /*Get victim frame*/
      vicfpn = GETVAL(vicpte_temp, PAGING_PTE_FPN_MASK, PAGING_PTE_FPN_LOBIT);
      ↪  // 8191 in decimal is 0->12 bit =1 in binary (total 13bit)
      /* Get free frame in MEMSWP */
      if (MEMPHY_get_freefp(caller->active_mswp, &swpfpn) < 0)
      {
        printf("Out of SWAP");
        return -3000;
      }
#ifdef RAM_STATUS_DUMP
      printf("[Page Replacement]\tPID
      ↪  #%d:\tVictim:%d\tPTE:%08x\tTarget:%d\t\n", caller->pid, vicfpn,
      ↪  *vicpte, tgtfpn);
#endif
      /* Copy victim frame to swap */
      __swap_cp_page(caller->mram, vicfpn, caller->active_mswp, swpfpn);

      /* Copy target frame from swap to mem */
      __swap_cp_page(caller->active_mswp, tgtfpn, caller->mram, vicfpn);

      // Cap nhat cho pte tro den page vua bi thay rang du lieu do da chuyen
      ↪  vao SWAP
      pte_set_swap(vicpte, 0, swpfpn);
```

```
      // Cap nhat gia tri frame number moi (trong Ram) cho page entry (bao
      ↪  rang pte da co frame number moi)
      pte_set_fpn(&mm->pgd[pgn], vicfpn);

#ifdef RAM_STATUS_DUMP
      printf("[After Swap]\tPID #%d:\tVictim:%d\tPTE:%08x\tTarget:%d\t\n",
      ↪  caller->pid, swpfpn, *vicpte, vicfpn);
#endif
      Add_LRU_page(&mm->pgd[pgn]);

      // Put frame trong trong swap vao free frame list
      MEMPHY_put_freefp(caller->active_mswp, tgtfpn);
    }

  }
  *fpn = GETVAL(mm->pgd[pgn], PAGING_PTE_FPN_MASK, PAGING_PTE_FPN_LOBIT);
  // pthread_mutex_unlock(&MEM_in_use);
  return 0;
}
```

4 Answering questions

4.1 Scheduler

Question: What is the advantage of using a priority queue compared to other scheduling algorithms you have learned?

Answer:

The advantage of using a priority queue in comparison with other scheduling algorithms is that it allows for the efficient prioritization and selection of processes based on their priority levels. A priority queue is a data structure where elements are assigned priorities, and the element with the highest priority is always selected first.

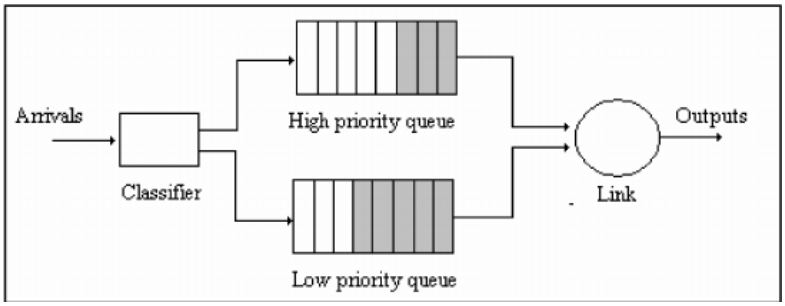

Figure 4. Describe priority queue

- **Priority-based scheduling:** A priority queue enables priority-based scheduling, where processes with higher priorities are executed first. This allows for the implementation of priority-driven policies, such as preemptive scheduling, where a higher-priority process can interrupt the execution of a lower-priority process.
- **Flexible and dynamic:** Priority queues can be easily updated and modified during runtime. As the priorities of processes change or new processes are created, the queue can be adjusted accordingly. This flexibility allows for efficient handling of dynamically changing priorities in real-time systems.
- **Efficient selection:** With a priority queue, the process with the highest priority can be selected in constant time, regardless of the number of processes in the queue. This ensures efficient selection of the next process to be executed, reducing the time complexity of the scheduling algorithm.
- **Customizable priorities:** Priority queues provide the flexibility to assign and modify priorities based on specific criteria or policies. The priority of a process can be determined by factors such as process importance, deadline constraints, resource requirements, or any other application-specific metric.
- **Fairness and responsiveness:** By assigning different priorities to processes, a priority queue can ensure fairness and responsiveness in scheduling. Higher-priority processes receive more CPU time, leading to better responsiveness for critical tasks or time-sensitive operations. It's important to note that while priority queues offer advantages in terms of priority-based scheduling, they may not be suitable for all scenarios. The choice of scheduling algorithm depends on the specific requirements of the operating system or application, considering factors such as fairness, throughput, response time, and system load.

4.2 Memory Management

4.2.1 The virtual memory mapping in each process

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?

CPU bus	PAGE size	PAGE bit	No pg entry	PAGE Entry sz	PAGE TBL	OFFSET bit	PGT mem	MEMPHY	fram bit
20	256B	12	~4000	4byte	16KB	8	2MB	1MB	12
22	256B	14	~16000	4byte	64KB	8	8MB	1MB	12
22	512B	13	~8000	4byte	32KB	9	4MB	1MB	11
22	512B	13	~8000	4byte	32KB	9	4MB	128kB	8
16	512B	8	256	4byte	1kB	9	128K	128kB	4

Table 1: Various CPU address bus configuration value

Answer: The proposed design of multiple memory segments in the source code declaration of the simple operating system offers several advantages:

- **Memory Organization:** By dividing the memory into multiple segments, the design provides a structured and organized layout for different types of data and code. Each segment can be dedicated to a specific purpose, such as the text segment for storing program instructions, the data segment for storing initialized data, and the heap segment for dynamic memory allocation. This organization improves the overall management and accessibility of different types of memory.
- **Memory Protection:** The use of multiple memory segments allows for memory protection and isolation. Each segment can have its own access permissions, such as read-only, read-write, or execute-only. This ensures that processes or segments cannot unintentionally modify or access memory areas that they are not supposed to, enhancing system security and stability.
- **Modularity and Flexibility:** The design of multiple memory segments enables modularity and flexibility in managing memory. It provides a mechanism to allocate and deallocate memory in a granular manner, allowing efficient memory utilization and avoiding fragmentation. It also allows for dynamic resizing of specific segments based on the needs of processes or applications.
- **Virtual Memory Mapping:** The virtual memory engine in the simple operating system maps virtual addresses provided by processes to corresponding physical addresses. The design of multiple memory segments facilitates this mapping process by providing clear boundaries and mappings between virtual and physical memory. This enables efficient address translation and memory management.
- **Code Readability and Maintainability:** By explicitly declaring multiple memory segments in the source code, the design improves code readability and maintainability. It provides a clear understanding of the memory layout and usage, making it easier to debug and modify the code in the future.

Overall, the design of multiple memory segments in the simple operating system enhances memory organization, protection, modularity, and code maintainability, contributing to the overall efficiency and reliability of the operating system.

4.2.2 The system physical memory

Question: What will happen if we divide the address into more than 2 levels in the paging memory management system?

Answer: If we divide the address into more than 2 levels in the paging memory management system, the RAM and SWAP devices, dividing the address into more levels may have some additional considerations:

If we divide the address into more than 2 levels in the paging memory management system,

it allows for a larger address space to be represented. Each additional level of paging provides more bits for addressing, enabling a larger number of virtual and physical pages.

Dividing the address into more levels increases the flexibility and scalability of the memory management system. It allows for a finer-grained mapping of virtual pages to physical pages, which can improve memory utilization and reduce fragmentation. It also enables the system to handle larger amounts of memory efficiently.

However, increasing the number of levels in the paging system also introduces additional complexity and overhead. Each level requires additional memory for storing page tables or page directory entries, and it adds extra levels of indirection during address translation, which can impact performance.

Overall, dividing the address into more than 2 levels in the paging memory management system can provide benefits in terms of addressing capacity and memory utilization, but it also comes with increased complexity and potential performance trade-offs. The decision to use more levels in the paging system should consider the specific requirements and constraints of the operating system and hardware architecture.

4.2.3 Paging-based address translation scheme

Question: What is the advantage and disadvantage of segmentation with paging?
Answer: Segmentation with paging combines the advantages of both segmentation and paging memory management techniques. Here are the advantages and disadvantages of segmentation with paging:

- *Advantage:*
  - Flexibility in Memory Management: Segmentation allows for logical division of the address space into variable-sized segments, which can represent different parts of a program, such as code, data, stack, etc. This provides a more flexible memory allocation scheme compared to a flat memory model.
  - Protection and Sharing: Segmentation allows for each segment to have its own access rights and permissions, providing a mechanism for memory protection. Segments can be shared between multiple processes, enabling efficient sharing of code and data.
  - Address Space Expansion: Paging enables the address space to be larger than the physical memory available. It allows for the allocation of memory in smaller fixed-sized units called pages. This enables efficient use of physical memory and supports larger address spaces.
  - Virtual Memory: Segmentation with paging enables the system to implement virtual memory, which allows processes to use more memory than what is physically available. It provides the illusion of a larger address space to processes by intelligently swapping pages in and out of physical memory.
- *Disadvantage:*
  - Increased Complexity: Segmentation with paging introduces additional complexity to the memory management system. It requires the management of both segment tables and page tables, adding overhead to address translation and memory access.

- Increased Overhead: The use of both segmentation and paging introduces additional memory overhead. Each segment and page requires additional metadata in the form of segment tables and page tables, which consume memory resources..
- Fragmentation: Combining segmentation and paging can lead to fragmentation of memory. External fragmentation can occur due to variable-sized segments, and internal fragmentation can occur due to fixed-sized pages.
- Performance Impact: The additional levels of indirection in address translation can introduce overhead and impact system performance. Accessing memory through multiple levels of tables adds latency to memory accesses.

Overall, segmentation with paging offers flexibility, protection, sharing, and support for virtual memory. However, it also introduces complexity, fragmentation, overhead, and potential performance trade-offs. The decision to use segmentation with paging should consider the specific requirements and trade-offs of the system and the hardware architecture.

4.3 Put It All Together

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: A race condition is a situation that occurs in concurrent systems, where the behavior or outcome of the system depends on the relative timing or interleaving of multiple concurrent operations. In other words, it is a race between two or more concurrent threads or processes to access and modify shared resources, leading to unexpected and erroneous results.

Based on the definition of race condition, the resource that shares among processes is the physical memory. One typical example that when 2 processes run in 2 different CPU access the free frame list of ram and take out the same frame number.

The race condition can be made by not protecting functions MEMPHY_put_freefp and MEMPHY_get_freefp and when the frame is taken out the program sleep 3 seconds to wait for the other alloc instruction. The result of race condition will be represent at section 5.3.

5 Some testcase of achieved results

5.1 Scheduling

5.1.1 Input and output

Here we will take out a *sched_1* file in the input folder as an example:

```
1 2 3
0 p1s 1
1 p2s 2
2 p3s 0
```

The description file is defined in the following format:

```
    [time slice] [N = number of CPU] [M = Number of Process to be run]
    [time 0] [path 0] [priority 0]
    [time 1] [path 1] [priority 1]
    ...
    [time M - 1] [path M - 1] [priority M - 1]
```

Based on the above figure, we can determine that the time slice and number of CPUs are equal to 2 and there are 3 processes.

This will return the output (there is a new line printout to double-check the status of the program):

```
    Time slot 0
        ld_routine
        Loaded a process at input / proc / p1s , PID : 1 PRIO : 1
Time slot 1
        CPU 0: Dispatched process 1
        Loaded a process at input / proc / p2s , PID : 2 PRIO : 2
Time slot 2
        CPU 0: Put process 1 to run queue
        CPU 0: Dispatched process 1
        CPU 1: Dispatched process 2
        Loaded a process at input / proc / p3s , PID : 3 PRIO : 0
Time slot 3
        CPU 0: Put process 1 to run queue
        CPU 0: Dispatched process 3
        CPU 1: Put process 2 to run queue
        CPU 1: Dispatched process 1
Time slot 4
        CPU 0: Put process 3 to run queue
        CPU 0: Dispatched process 3
        CPU 1: Put process 1 to run queue
        CPU 1: Dispatched process 1
Time slot 5
        CPU 0: Put process 3 to run queue
        CPU 0: Dispatched process 3
        CPU 1: Put process 1 to run queue
        CPU 1: Dispatched process 1
Time slot 6
        CPU 0: Put process 3 to run queue
        CPU 0: Dispatched process 3
        CPU 1: Put process 1 to run queue
```

```
        CPU 1: Dispatched process 1
Time slot 7
        CPU 0: Put process 3 to run queue
        CPU 0: Dispatched process 3
        CPU 1: Put process 1 to run queue
        CPU 1: Dispatched process 1
Time slot 8
        CPU 0: Put process 3 to run queue
        CPU 0: Dispatched process 3
        CPU 1: Put process 1 to run queue
        CPU 1: Dispatched process 1
Time slot 9
        CPU 0: Put process 3 to run queue
        CPU 0: Dispatched process 3
        CPU 1: Put process 1 to run queue
        CPU 1: Dispatched process 1
Time slot 10
        CPU 0: Put process 3 to run queue
        CPU 0: Dispatched process 3
Faculty of Computer Science & Computer Engineering
        CPU 1: Put process 1 to run queue
        CPU 1: Dispatched process 1
Time slot 11
        CPU 0: Put process 3 to run queue
        CPU 0: Dispatched process 3
        CPU 1: Processed 1 has finished
        CPU 1: Dispatched process 2
Time slot 12
        CPU 1: Put process 2 to run queue
        CPU 1: Dispatched process 2
        CPU 0: Put process 3 to run queue
        CPU 0: Dispatched process 3
Time slot 13
        CPU 1: Put process 2 to run queue
        CPU 1: Dispatched process 2
        CPU 0: Put process 3 to run queue
        CPU 0: Dispatched process 3
Time slot 14
        CPU 0: Processed 3 has finished
        CPU 1: Put process 2 to run queue
```

```
        CPU 1: Dispatched process 2
        CPU 0 stopped
Time slot 15
        CPU 1: Put process 2 to run queue
        CPU 1: Dispatched process 2
Time slot 16
        CPU 1: Put process 2 to run queue
        CPU 1: Dispatched process 2
Time slot 17
        CPU 1: Put process 2 to run queue
        CPU 1: Dispatched process 2
Time slot 18
        CPU 1: Put process 2 to run queue
        CPU 1: Dispatched process 2
Time slot 19
        CPU 1: Put process 2 to run queue
        CPU 1: Dispatched process 2
Time slot 20
        CPU 1: Put process 2 to run queue
        CPU 1: Dispatched process 2
Time slot 21
        CPU 1: Put process 2 to run queue
        CPU 1: Dispatched process 2
Time slot 22
        CPU 1: Processed 2 has finished
        CPU 1 stopped
```

5.1.2 Grantt Diagram

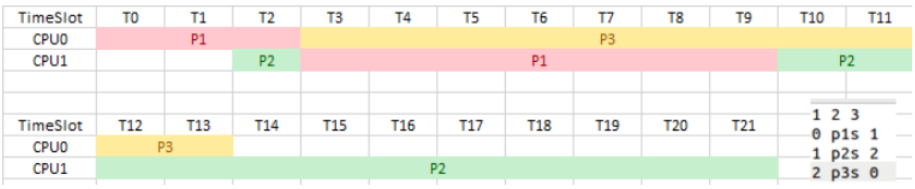

Figure 5. Grantt diagram

5.2 Memtest: Alloc, Read, Write, Free

For testing the behaviour of memory allocation, read, write and free, we use this test case for simulation:

```
2 1 1
2048 16777216 0 0 0
1 t4 0
```

- with t4:

```
1 12
calc
alloc 300 0
alloc 300 4
alloc 100 1
write 100 4 20
read 1 20 20
write 102 0 20
alloc 300 5
alloc 300 6
alloc 400 7
write 104 4 20
calc
free 4
calc
```

Results:

```
Time slot   0
ld_routine
Time slot   1
        Loaded a process at input/proc/t4, PID: 1 PRIO: 0
Time slot   2
        CPU 0: Dispatched process  1
```

```
Time slot   3
++++++++++++++++++++++++++++++++++++++++++++++++++++
Process 1 ALLOC CALL | SIZE = 300

>>>>>Alloc Case>>>>>No free region. #RGID: 0
```

```
[Page mapping]        PID #1:        Frame:0        PTE:80000000        PGN:0
[Page mapping]        PID #1:        Frame:1        PTE:80000001        PGN:1
>>>>>Done>>>>> #RGID: 0
=======================================
Region id 0 : start = 0, end = 300
=======================================
Process 1 Free Region list
Start = 300, end = 512
=======================================
VMA id 1 : start = 0, end = 512, sbrk = 512
----------- RAM mapping status -----------
Number of mapped frames:        2
Number of remaining frames:        6
-----------------------------------------
=================LRU LIST================
FPN:
[0] -> [1]
=======================================
```

Explain: *The above time slot 3 results indicates that*

Process 1 makes an ALLOC CALL with a requested size of 300. This means that the process is requesting a block of memory with a size of 300 units.The system attempts to allocate memory for process 1 but encounters a "No free region" situation for region ID (RGID) 0. This suggests that there is no available memory region of sufficient size to fulfill the allocation request.

The page mapping information provides details about the association between virtual memory pages and physical memory frames for Process 1. In this case, two page mappings are shown: Frame 0 is mapped to Page Table Entry (PTE) 80000000 and Page Number (PGN) 0, and Frame 1 is mapped to PTE 80000001 and PGN 1. These mappings help establish the relationship between the virtual memory address space used by Process 1 and the corresponding physical memory addresses.

The Process 1 Free Region list shows the available memory regions that can be allocated for Process 1. In this case, there is a single entry in the Free Region list, starting at address 300 and ending at address 512. This means that there is one free memory region available for Process 1, with a size of 212 units.
The Virtual Memory Area (VMA) with ID 1 is defined with a start address of 0, an end address of 512, and an sbrk value of 512. The VMA represents a contiguous range of virtual memory addresses that are allocated for this process. The sbrk value indicates the current program break, which is the boundary between the initialized data and the heap.

The RAM mapping status provides an overview of the memory utilization. It states that there are currently 2 frames mapped, meaning that 2 physical memory frames are in use. Additionally,

it indicates that there are 6 remaining frames available for allocation.

==> The LRU (Least Recently Used) list of frames shows the order in which frames have been accessed or used. In this case, Frame 0 is listed as the most recently used frame, followed by Frame 1. The LRU list is used in memory management to determine which frames should be replaced when new memory allocations are required. By tracking the order of frame usage, the system can optimize memory utilization.

```
Time slot   4
        CPU 0: Put process  1 to ready queue
        CPU 0: Dispatched process  1
++++++++++++++++++++++++++++++++++++++++++++++++++
Process 1 ALLOC CALL | SIZE = 300

>>>>>Alloc Case>>>>>No free region. #RGID: 4
[Page mapping]        PID #1:        Frame:2        PTE:80000002        PGN:2
[Page mapping]        PID #1:        Frame:3        PTE:80000003        PGN:3
>>>>>Done>>>>> #RGID: 4
========================================
Region id 0 : start = 0, end = 300
Region id 4 : start = 512, end = 812
========================================
Process 1 Free Region list
Start = 300, end = 512
Start = 812, end = 1024
========================================
VMA id 1 : start = 0, end = 1024, sbrk = 1024
----------- RAM mapping status -----------
Number of mapped frames:        4
Number of remaining frames:        4
----------------------------------------
=================LRU LIST=================
FPN:
[0] -> [1] -> [2] -> [3]
========================================
```

Explain: *The above time slot 4 results indicates that*
Process 1 makes an ALLOC CALL with SIZE = 300.
It is determined that there is no free region available for allocation with the Region ID (RGID) 4.
The page mapping for Process 1 shows that Frame 2 is mapped to Page Table Entry (PTE) 80000002 and Page Number (PGN) 2, and Frame 3 is mapped to PTE 80000003 and PGN 3.
The regions in the system are as follows:

- Region ID 0: Start = 0, End = 300
- Region ID 4: Start = 512, End = 812

The free regions for Process 1 are:

- Start = 300, End = 512
- Start = 812, End = 1024

The Virtual Memory Address (VMA) ID 1 has the following properties:

- Start = 0, End = 1024
- sbrk = 1024

The RAM mapping status shows that there are 4 frames mapped and 4 frames remaining.
The LRU list (Least Recently Used) for the frames is as follows:
FPN: [0] -> [1] -> [2] -> [3]

```
Time slot   5
++++++++++++++++++++++++++++++++++++++++++++++++++
Process 1 ALLOC CALL | SIZE = 100

>>>>>Alloc Case>>>>> GET FREE RG in FREERG LIST. #RGID: 1
current_pgn = 1
print head: 80000000
Looking for:         80000001
Found:         80000001

>>>>>DONE>>>>>  #RGID: 1
FOUND A FREE region to alloc.
========================================
Region id 0 : start = 0, end = 300
Region id 1 : start = 300, end = 400
Region id 4 : start = 512, end = 812
VMA id 1 : start = 0, end = 1024, sbrk = 1024
========================================
Process 1 Free Region list
Start = 400, end = 512
Start = 812, end = 1024
========================================
----------- RAM mapping status -----------
Number of mapped frames:        4
Number of remaining frames:        4
----------------------------------------
=================LRU LIST=================
```

```
FPN:
[0] -> [2] -> [3] -> [1]
=======================================
```

Explain: *The above time slot 5 results indicates that*
Process 1 makes an ALLOC CALL with SIZE = 100, indicating that it is requesting a memory allocation of 100 units.
The system looks for a free region to allocate the memory. It searches the FREERG LIST to find an available region.
It finds a free region in Region ID (RGID) 1. This region starts at address 300 and ends at address 400.The page table lookup process searches for the specific virtual address (80000001) in the page table.
The allocation process for RGID 1 is completed. The system confirms that a free region is available for allocation.
The system updates the region information as follows:

- Region ID 0: Start = 0, End = 300
- Region ID 1: Start = 300, End = 400
- Region ID 4: Start = 512, End = 812

The Virtual Memory Address (VMA) ID 1 remains the same:

- VMA ID 1: Start = 0, End = 1024, sbrk = 1024

Process 1 now has two free regions available for further allocations:

- Start = 400, End = 512
- Start = 812, End = 1024

The RAM mapping status remains the same:

- Number of mapped frames: 4
- Number of remaining frames: 4

The order of frames in the LRU list is now: [0] -> [2] -> [3] -> [1]

```
Time slot   6
        CPU 0: Put process  1 to ready queue
        CPU 0: Dispatched process  1
++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Process 1 write region=4 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003
```

```
Print content of RAM (only print nonzero value)
--------------------------------
Address 0x00000214: 100
--------------------------------
=================LRU LIST=================
FPN:
[0] -> [3] -> [1] -> [2]
=========================================
```

Explain: *The above time slot 6 results indicates that*
CPU 0: Process 1 was put into the ready queue. This means that Process 1 is now in a state where it is ready to be executed by the CPU.
CPU 0: Process 1 was dispatched, which means that the CPU selected Process 1 to execute.
Process 1 executed a write operation. It wrote a value of 100 to Region 4 at offset 20.The system printed the page table contents. The page table ranges from virtual address 0 to 1024. The page table entries (PTEs) are as follows:

- PTE at virtual address 0x00000000: Corresponds to physical frame 80000000.
- PTE at virtual address 0x00000004: Corresponds to physical frame 80000001.
- PTE at virtual address 0x00000008: Corresponds to physical frame 80000002.
- PTE at virtual address 0x00000012: Corresponds to physical frame 80000003.

The system printed the content of physical memory (RAM) but only displayed nonzero values. In this case, the value at address 0x00000214 is 100.
The order of frames in the LRU list is now: [0] -> [3] -> [1] -> [2]

```
Time slot   7
++++++++++++++++++++++++++++++++++++++++++++++++++
Process 1 read region=1 offset=20 value=0
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003

Print content of RAM (only print nonzero value)
--------------------------------
Address 0x00000214: 100
--------------------------------
=================LRU LIST=================
FPN:
```

```
[0] -> [3] -> [2] -> [1]
========================================
```

Explain: *The above time slot 7 results indicates that*
Process 1 executed a read operation. It read the value from Region 1 at offset 20. However, the value it retrieved was 0.
The system printed the page table contents. The page table ranges from virtual address 0 to 1024. The page table entries (PTEs) are as follows:

- PTE at virtual address 0x00000000: Corresponds to physical frame 80000000.
- PTE at virtual address 0x00000004: Corresponds to physical frame 80000001.
- PTE at virtual address 0x00000008: Corresponds to physical frame 80000002.
- PTE at virtual address 0x00000012: Corresponds to physical frame 80000003.

The system printed the content of physical memory (RAM) but only displayed nonzero values. In this case, the value at address 0x00000214 is 100.
The order of frames in the LRU list is now: [0] -> [3] -> [2] -> [1]

```
Time slot   8
        CPU 0: Put process  1 to ready queue
        CPU 0: Dispatched process  1
++++++++++++++++++++++++++++++++++++++++++++++++++
Process 1 write region=0 offset=20 value=102
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003

Print content of RAM (only print nonzero value)
--------------------------------
Address 0x00000014: 102
--------------------------------
Address 0x00000214: 100
--------------------------------
=================LRU LIST=================
FPN:
[3] -> [2] -> [1] -> [0]
========================================
```

Explain: *The above time slot 8 results indicates that*
Process 1 executed a write operation. It wrote a value of 102 to Region 0 at offset 20.The system printed the page table contents. The page table ranges from virtual address 0 to 1024. The page table entries (PTEs) are as follows:

- PTE at virtual address 0x00000000: Corresponds to physical frame 80000000.
- PTE at virtual address 0x00000004: Corresponds to physical frame 80000001.
- PTE at virtual address 0x00000008: Corresponds to physical frame 80000002.
- PTE at virtual address 0x00000012: Corresponds to physical frame 80000003.

The system printed the content of physical memory (RAM) but only displayed nonzero values. In this case, the values at addresses 0x00000014 and 0x00000214 are 102 and 100, respectively.
The order of frames in the LRU list is now: [3] -> [2] -> [1] -> [0]

```
Time slot   9
++++++++++++++++++++++++++++++++++++++++++++++++++
Process 1 ALLOC CALL | SIZE = 300

>>>>>Alloc Case>>>>>No free region. #RGID: 5
[Page mapping]        PID #1:        Frame:4        PTE:80000004        PGN:4
[Page mapping]        PID #1:        Frame:5        PTE:80000005        PGN:5
>>>>>Done>>>>> #RGID: 5
========================================
Region id 0 : start = 0, end = 300
Region id 1 : start = 300, end = 400
Region id 4 : start = 512, end = 812
Region id 5 : start = 1024, end = 1324
========================================
Process 1 Free Region list
Start = 400, end = 512
Start = 812, end = 1024
Start = 1324, end = 1536
========================================
VMA id 1 : start = 0, end = 1536, sbrk = 1536
----------- RAM mapping status -----------
Number of mapped frames:        6
Number of remaining frames:        2
----------------------------------------
================LRU LIST================
FPN:
[3] -> [2] -> [1] -> [0] -> [4] -> [5]
========================================
```

Explain: *The above time slot 9 results indicates that*
Process 1 made an ALLOC call, requesting a memory allocation of size 300.
The system encountered the "No free region" case with Region ID 5. This means that there is no

available free memory region to satisfy the allocation request.
The system printed page mappings for Process 1. The page mappings indicate the frame and page table entry (PTE) information for the allocated pages. In this case:

- Page mapping for PID 1: Frame 4, PTE 80000004, Page Number (PGN) 4
- Page mapping for PID 1: Frame 5, PTE 80000005, Page Number (PGN) 5

The system indicated that the allocation process for Region ID 5 is completed.The system provided information about the memory regions:

- Region ID 0: Starts at address 0 and ends at address 300.
- Region ID 1: Starts at address 300 and ends at address 400.
- Region ID 4: Starts at address 512 and ends at address 812.
- Region ID 5: Starts at address 1024 and ends at address 1324.

The system displayed the free region list for Process 1, indicating the available memory regions for future allocations.
The system provided information about the virtual memory areas (VMA). In this case, VMA ID 1 starts at address 0 and ends at address 1536. The "sbrk" value indicates the current program break, which is at address 1536. **The order of frames in the LRU list is now: [3] -> [2] -> [1] -> [0] -> [4] -> [5]**

```
Time slot  10
        CPU 0: Put process  1 to ready queue
        CPU 0: Dispatched process  1
++++++++++++++++++++++++++++++++++++++++++++++++++
Process 1 ALLOC CALL | SIZE = 300

>>>>>Alloc Case>>>>>No free region. #RGID: 6
[Page mapping]        PID #1:        Frame:6        PTE:80000006        PGN:6
[Page mapping]        PID #1:        Frame:7        PTE:80000007        PGN:7
>>>>>Done>>>>> #RGID: 6
======================================
Region id 0 : start = 0, end = 300
Region id 1 : start = 300, end = 400
Region id 4 : start = 512, end = 812
Region id 5 : start = 1024, end = 1324
Region id 6 : start = 1536, end = 1836
======================================
Process 1 Free Region list
Start = 400, end = 512
Start = 812, end = 1024
Start = 1324, end = 1536
Start = 1836, end = 2048
```

```
========================================
VMA id 1 : start = 0, end = 2048, sbrk = 2048
----------- RAM mapping status -----------
Number of mapped frames:        8
Number of remaining frames:        0
----------------------------------------
=================LRU LIST=================
FPN:
[3] -> [2] -> [1] -> [0] -> [4] -> [5] -> [6] -> [7]
========================================
```

Explain: *The above time slot 10 results indicates that*
Process 1 made an ALLOC call, requesting a memory allocation of size 300.
The system encountered the "No free region" case with Region ID 6. This means that there is no available free memory region to satisfy the allocation request.
The system printed page mappings for Process 1. The page mappings indicate the frame and page table entry (PTE) information for the allocated pages. In this case:

- Page mapping for PID 1: Frame 6, PTE 80000006, Page Number (PGN) 6
- Page mapping for PID 1: Frame 7, PTE 80000007, Page Number (PGN) 7

The system provided information about the memory regions:

- Region ID 0: Starts at address 0 and ends at address 300.
- Region ID 1: Starts at address 300 and ends at address 400.
- Region ID 4: Starts at address 512 and ends at address 812.
- Region ID 5: Starts at address 1024 and ends at address 1324.
- Region ID 6: Starts at address 1536 and ends at address 1836.

The system displayed the free region list for Process 1, indicating the available memory regions for future allocations.The system provided information about the virtual memory areas (VMA). In this case, VMA ID 1 starts at address 0 and ends at address 2048. The "sbrk" value indicates the current program break, which is at address 2048.
The system displayed the RAM mapping status, showing the number of mapped frames and the number of remaining frames. In this case, all 8 frames are mapped, and there are no remaining frames available.
The order of frames in the LRU list is now: [3] -> [2] -> [1] -> [0] -> [4] -> [5] -> [6] -> [7]

```
Time slot  11
++++++++++++++++++++++++++++++++++++++++++++++++++
Process 1 ALLOC CALL | SIZE = 400

>>>>>Alloc Case>>>>>No free region. #RGID: 7
```

```
[Page Replacement]         PID #1:         Victim:3         PTE:80000003
[After Swap]         PID #1:         Victim:0         PTE:40000000
[Page Replacement]         PID #1:         Victim:2         PTE:80000002
[After Swap]         PID #1:         Victim:1         PTE:40000020
[Page mapping]         PID #1:         Frame:3         PTE:80000003         PGN:8
[Page mapping]         PID #1:         Frame:2         PTE:80000002         PGN:9
>>>>>Done>>>>> #RGID: 7
========================================
Region id 0 : start = 0, end = 300
Region id 1 : start = 300, end = 400
Region id 4 : start = 512, end = 812
Region id 5 : start = 1024, end = 1324
Region id 6 : start = 1536, end = 1836
Region id 7 : start = 2048, end = 2448
========================================
Process 1 Free Region list
Start = 400, end = 512
Start = 812, end = 1024
Start = 1324, end = 1536
Start = 1836, end = 2048
Start = 2448, end = 2560
========================================
VMA id 1 : start = 0, end = 2560, sbrk = 2560
----------- RAM mapping status -----------
Number of mapped frames:         8
Number of remaining frames:         0
----------------------------------------
================LRU LIST================
FPN:
[1] -> [0] -> [4] -> [5] -> [6] -> [7] -> [3] -> [2]
========================================
```

Explain: *The above time slot 11 results indicates that*
Process 1 made an ALLOC call, requesting a memory allocation of size 400.
The system encountered the "No free region" case with Region ID 7. This means that there is no available free memory region to satisfy the allocation request.
The system performed page replacement to free up memory for the allocation. The page replacement algorithm selected Victim Frame 3 from Process 1, which had Page Table Entry (PTE) 80000003. After the swap, Victim Frame 0 with PTE 40000000 was replaced in the process. Then, Victim Frame 2 with PTE 80000002 was selected, and Victim Frame 1 with PTE 40000020 was replaced.
The system printed the page mappings for Process 1 after the page replacements:

- Page mapping for PID 1: Frame 3, PTE 80000003, Page Number (PGN) 8
- Page mapping for PID 1: Frame 2, PTE 80000002, Page Number (PGN) 9

The system provided information about the memory regions:

- Region ID 0: Starts at address 0 and ends at address 300.
- Region ID 1: Starts at address 300 and ends at address 400.
- Region ID 4: Starts at address 512 and ends at address 812.
- Region ID 5: Starts at address 1024 and ends at address 1324.
- Region ID 6: Starts at address 1536 and ends at address 1836.
- Region ID 7: Starts at address 2048 and ends at address 2448.

The system provided information about the virtual memory areas (VMA). In this case, VMA ID 1 starts at address 0 and ends at address 2560. The "sbrk" value indicates the current program break, which is at address 2560.
The system displayed the RAM mapping status, showing the number of mapped frames and the number of remaining frames. In this case, all 8 frames are mapped, and there are no remaining frames available.
The order of frames in the LRU list is now: [1] -> [0] -> [4] -> [5] -> [6] -> [7] -> [3] -> [2]

```
Time slot  12
        CPU 0: Put process  1 to ready queue
        CPU 0: Dispatched process  1
++++++++++++++++++++++++++++++++++++++++++++++++++
Process 1 write region=4 offset=20 value=104
[Page Replacement]        PID
↪  #1:        Victim:1        PTE:80000001        Target:1
[After Swap]        PID
↪  #1:        Victim:2        PTE:40000040        Target:1
print_pgtbl: 0 - 2560
00000000: 80000000
00000004: 40000040
00000008: 80000001
00000012: 40000000
00000016: 80000004
00000020: 80000005
00000024: 80000006
00000028: 80000007
00000032: 80000003
00000036: 80000002
```

```
Print content of RAM (only print nonzero value)
---------------------------------
Address 0x00000014: 102
---------------------------------
Address 0x00000114: 104
---------------------------------
=================LRU LIST=================
FPN:
[0] -> [4] -> [5] -> [6] -> [7] -> [3] -> [2] -> [1]
==========================================
```

Explain: *The above time slot 12 results indicates that*
Process 1 performed a write operation on Region 4, offset 20, with a value of 104. This means that Process 1 wrote the value 104 at the specified location in memory.
The system performed page replacement to free up memory for the write operation. The page replacement algorithm selected Victim Frame 1 from Process 1, which had Page Table Entry (PTE) 80000001. After the swap, Victim Frame 2 with PTE 40000040 was replaced, and the target PTE was set to 1.
The system printed the page table for Process 1 using the print_pgtbl command. The page table entries (PTEs) for the corresponding virtual addresses are shown:

- Virtual address 0x00000000: PTE 80000000
- Virtual address 0x00000004: PTE 40000040
- Virtual address 0x00000008: PTE 80000001
- Virtual address 0x00000012: PTE 40000000
- Virtual address 0x00000016: PTE 80000004
- Virtual address 0x00000020: PTE 80000005
- Virtual address 0x00000024: PTE 80000006
- Virtual address 0x00000028: PTE 80000007
- Virtual address 0x00000032: PTE 80000003
- Virtual address 0x00000036: PTE 80000002

The system printed the content of RAM (physical memory) for non-zero values. In this case, two addresses had non-zero values:

- Address 0x00000014: Value 102
- Address 0x00000114: Value 104

The order of frames in the LRU list is now: [0] -> [4] -> [5] -> [6] -> [7] -> [3] -> [2] -> [1]

```
Time slot  13
Time slot  14
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

5.3 Testing race condition with memory

- Testcase:

```
2 2 2
2048 16777216 0 0 0
0 t1 1
0 t1 1
```

with t1:

```
1 2
alloc 300 0
alloc 300 1
```

Result:

```
Time slot   0
ld_routine
        Loaded a process at input/proc/t1, PID: 1 PRIO: 1
Time slot   1
        CPU 0: Dispatched process  1
++++++++++++++++++++++++++++++++++++++++++++++++++
Process 1 ALLOC CALL | SIZE = 300

>>>>>Alloc Case>>>>>No free region. #RGID: 0
[Page mapping]        PID #1:        Frame:0        PTE:80000000        PGN:0
[Page mapping]        PID #1:        Frame:1        PTE:80000001        PGN:1
>>>>>Done>>>>> #RGID: 0
========================================
Region id 0 : start = 0, end = 300
========================================
Process 1 Free Region list
Start = 300, end = 512
========================================
VMA id 1 : start = 0, end = 512, sbrk = 512
```

```
----------- RAM mapping status -----------
Number of mapped frames:        2
Number of remaining frames:        6
------------------------------------------
=================LRU LIST=================
FPN:
[0] -> [1]
==========================================

        Loaded a process at input/proc/t1, PID: 2 PRIO: 1
Time slot   2
        CPU 1: Dispatched process  2
++++++++++++++++++++++++++++++++++++++++++++++++++
Process 2 ALLOC CALL | SIZE = 300

>>>>>Alloc Case>>>>>No free region. #RGID: 0
[Page mapping]        PID #2:        Frame:2        PTE:80000002        PGN:0
[Page mapping]        PID #2:        Frame:3        PTE:80000003        PGN:1
>>>>>Done>>>>> #RGID: 0
==========================================
Region id 0 : start = 0, end = 300
==========================================
Process 2 Free Region list
Start = 300, end = 512
==========================================
VMA id 1 : start = 0, end = 512, sbrk = 512
----------- RAM mapping status -----------
Number of mapped frames:        4
Number of remaining frames:        4
------------------------------------------
=================LRU LIST=================
FPN:
[0] -> [1] -> [2] -> [3]
==========================================

++++++++++++++++++++++++++++++++++++++++++++++++++
Process 1 ALLOC CALL | SIZE = 300

>>>>>Alloc Case>>>>>No free region. #RGID: 0
[Page mapping]        PID #1:        Frame:4        PTE:80000004        PGN:2
```

```
[Page mapping]        PID #1:        Frame:5        PTE:80000005        PGN:3
>>>>>Done>>>>> #RGID: 0
=======================================
Region id 0 : start = 512, end = 812
=======================================
Process 1 Free Region list
Start = 300, end = 512
Start = 812, end = 1024
=======================================
VMA id 1 : start = 0, end = 1024, sbrk = 1024
----------- RAM mapping status -----------
Number of mapped frames:        6
Number of remaining frames:        2
-----------------------------------------
=================LRU LIST================
FPN:
[0] -> [1] -> [2] -> [3] -> [4] -> [5]
=======================================

Time slot   3
++++++++++++++++++++++++++++++++++++++++++++++++
Process 2 ALLOC CALL | SIZE = 300

>>>>>Alloc Case>>>>>No free region. #RGID: 0
[Page mapping]        PID #2:        Frame:6        PTE:80000006        PGN:2
[Page mapping]        PID #2:        Frame:7        PTE:80000007        PGN:3
>>>>>Done>>>>> #RGID: 0
=======================================
Region id 0 : start = 512, end = 812
=======================================
Process 2 Free Region list
Start = 300, end = 512
Start = 812, end = 1024
=======================================
VMA id 1 : start = 0, end = 1024, sbrk = 1024
----------- RAM mapping status -----------
Number of mapped frames:        8
Number of remaining frames:        0
-----------------------------------------
=================LRU LIST================
```

```
FPN:
[0] -> [1] -> [2] -> [3] -> [4] -> [5] -> [6] -> [7]
=====================================

        CPU 0: Processed  1 has finished
        CPU 0 stopped
Time slot   4
        CPU 1: Processed  2 has finished
        CPU 1 stopped
```

As can be seen, the behavior of memory allocation is not correct, region 0 & 1 do not have enough space but it still alloc 300 bytes. We don't know how to solve this problem, synchronize tool will be add later.

6 Conclusion

In conclusion, the OS works pretty well in some general cases, but there are some exceptions that users can read before writing, read and write before alloc or re-alloc, and so on. These exceptions are out of scope and should be handled by the compiler or other mechanism, in this assignment the input from users is expected to be correct in logic.