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

CO2018 - Operating Systems (LAB)

CC01 - GROUP 4 - ASSIGNMENT REPORT

Simple Operating System

Under the guidance of: Lecturer Le Thanh Van

HO CHI MINH CITY, DECEMBER 2023

Member list

No.	Full name	Student ID
1	Pham Quang Huy	2152599
2	Nguyen Minh Hieu	2152555
3	Vu Nam Binh	2152441

Contents

1 Scheduler

1.1 Background

Scheduler is an operating system module that selects the next jobs to be admitted into the system and the next process to run. This term plays an essential role in a multiprogramming operating system (it is required in multitasking OS). In case, the OS does not have this component it can only run single program, exactly as MS-DOS did before. To complete the scheduler for the simple OS in this assignment, we have to complete 2 source files.

In source file **queue.c**, we are required to implement 2 functions including **enqueue()** and **dequeue()**:

- **enqueue()**: put a new process to queue q.
- **dequeue()**: return a PCB with the highest priority out of the queue and remove it from q.

While on the source file **sched.c**, we are required to implement the function get_mlq_proc. Actually, in the assignment specification of Operating System course in this semester, we are required to implement the function get_proc, however, when we checked this function, we saw it return another function called get_mlq_proc and there is also a **"TODO"** part so we will only implement the function **get_mlq_proc** in **sched.c**. **NOTE:** There is 2 get_proc functions in this file, one is inside the preprocessor directive **#ifdef MLQ_SCHED** and one in the **#else**, but the latter is never touched because we always define **MLQ_SCHED** in this assignment (even when we run the 2nd part which belongs to Memoy Management concept). Therefore, we will skip the latter also.

- **get_mlq_proc()**: get a process from priority *ready_queue*.

1.2 Sched

- **Output results**

```
nabicold@nabicold-virtual-machine:~/OSAssignment$ ./os sched
Time slot   0
ld_routine
        Loaded a process at input/proc/p1s, PID: 1 PRIO: 1
        CPU 0: Dispatched process  1
Time slot   1
        Loaded a process at input/proc/p2s, PID: 2 PRIO: 20
        CPU 1: Dispatched process  2
Time slot   2
        Loaded a process at input/proc/p3s, PID: 3 PRIO: 7
Time slot   3
Time slot   4
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   5
        CPU 1: Put process  2 to run queue
        CPU 1: Dispatched process  3
Time slot   6
Time slot   7
Time slot   8
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   9
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  3
Time slot  10
        CPU 0: Processed  1 has finished
        CPU 0: Dispatched process  2
Time slot  11
Time slot  12
Time slot  13
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  3
Time slot  14
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  15
Time slot  16
        CPU 1: Processed  3 has finished
        CPU 1 stopped
Time slot  17
Time slot  18
        CPU 0: Processed  2 has finished
        CPU 0 stopped
```

Figure 1: Result for running SCHED input file

- **Gantt Diagrams**

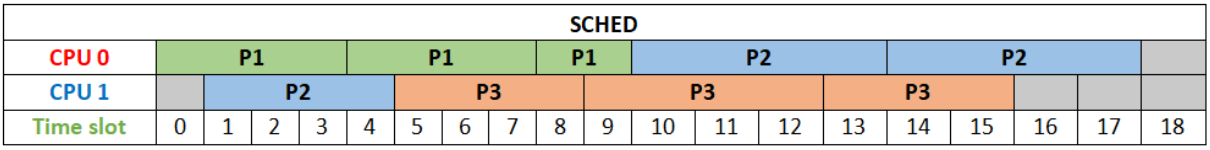


Figure 2: SCHED Gannt Chart

1.3 Sched_0

- **Output results**

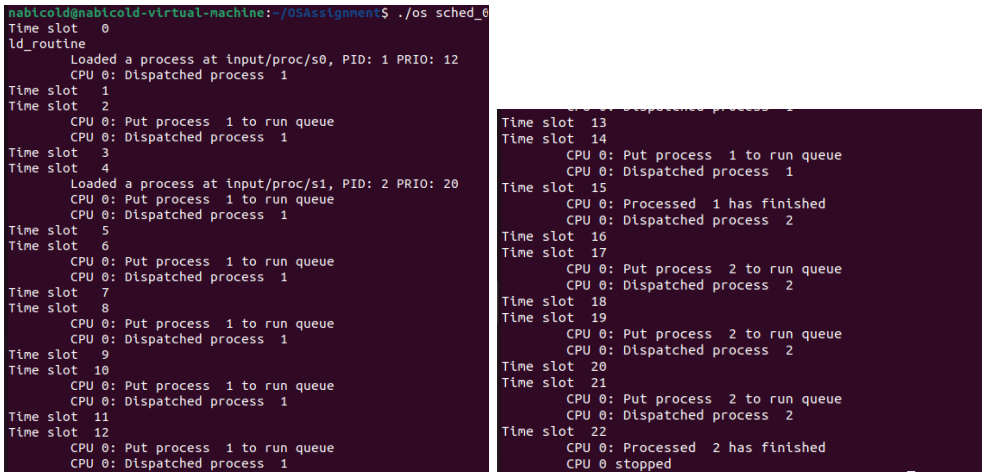


Figure 3: Result for running Sched_0 input file

- **Gantt Diagrams**

SCHED_0												
CPU 0	P1		P1		P1		P1		P1		P1	
Time slot	0	1	2	3	4	5	6	7	8	9	10	11

SCHED_0											
CPU 0	P1		P1	P2		P2		P2		P2	
Time slot	12	13	14	15	16	17	18	19	20	21	22

Figure 4: SCHED_0 Gann Chart

1.4 Sched_1

- **Output results**

```
nabicold@nabicold-virtual-machine:~/OSAssignment$ ./os sched_1
Time slot   0
ld_routine
        Loaded a process at input/proc/s0, PID: 1 PRIO: 12
Time slot   1
        CPU 0: Dispatched process  1
Time slot   2
Time slot   3
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   4
        Loaded a process at input/proc/s1, PID: 2 PRIO: 20
Time slot   5
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   6
        Loaded a process at input/proc/s2, PID: 3 PRIO: 20
Time slot   7
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
        Loaded a process at input/proc/s3, PID: 4 PRIO: 7
Time slot   8
Time slot   9
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
Time slot  10
Time slot  11
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  4
Time slot  12
Time slot  13
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  4
Time slot  14
Time slot  15
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  4
Time slot  16
Time slot  17
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  4
Time slot  18
Time slot  19
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  4
Time slot  20
        CPU 0: Processed  4 has finished
        CPU 0: Dispatched process  1
Time slot  21
Time slot  22
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  23
Time slot  24
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  25
Time slot  26
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  27
        CPU 0: Processed  1 has finished
        CPU 0: Dispatched process  2
Time slot  28
Time slot  29
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
Time slot  30
Time slot  31
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  2
Time slot  32
Time slot  33
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
Time slot  34
Time slot  35
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  2
Time slot  36
Time slot  37
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
Time slot  38
Time slot  39
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  2
Time slot  40
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  3
Time slot  41
Time slot  42
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  3
Time slot  43
Time slot  44
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  3
Time slot  45
Time slot  46
        CPU 0: Processed  3 has finished
        CPU 0 stopped
```

Figure 5: Result for running Sched_1 input file

- **Gantt Diagrams**

SCHED_1																	
CPU 0		P1		P1		P1		P1		P4		P4		P4		P4	
Time slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

SCHED_1																
CPU 0	P4		P4		P1		P1		P1		P1		P2		P3		P2	
Time slot	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32

SCHED_1														
CPU 0	P3		P2		P3		P2	P3		P3		P3		
Time slot	33	34	35	36	37	38	39	40	41	42	43	44	45	46

Figure 6: SCHED_1 Gannt Chart

1.5 Implementation

1.5.1 queue.c

In this file, we implemented 2 basic functions for a queue, namely enqueue and dequeue:

- ***enqueue***: Firstly, check if the queue or the process passed in is NULL or not (if NULL, return), then check if the queue is FULL or not (if FULL, return). If not, we push the new process to the end of the queue the same ass the idea behind a normal queue that we've already known and increase the size of this queue.
- ***dequeue***: Initially, we put an if_condition to check if the queue is empty or NULL (if YES, return NULL). If not, we continued running a for loop to find the pcb whose priority is the highest (mean that we find the smallest value because the lower value, the higher priority is). After that, we create a struct pcb_t temp as a temporary variable to store the pcb at the highest priority and shift left all other elements in the queue that are on the right side of the highest priority element. However, we only do this when the highest priority does not place at the tail of the queue. Finally, we assign a NULL value to the final element of the queue, decrease the size of queue and return the temp variable that stores the pcb we mentioned before.

1.5.2 sched.c

In this file, we implemented funciton get_mlq_proc, this function is called by the funcion get_proc:

- ***get_mlq_proc***: Started with a pointer pcb_t proc=NULL, we use mutex_lock to protect the queue, then we run a for loop from the beginning to MAX_PRIO, which is corresponding to the maximum elements of a multilevel queue. In each loop, we check a condition if the ready_queue is not empty and its slot is not used up, we allow it to dequeue, store in the pointer proc above and increase its used slot by 1 then break the loop. If not, we will check if it's not empty and its slot reached max slot it can use, we reassign it to 0. This step is to reset the slot for future use of this queue, if we do not do this, after running a while, all the queue will be full. After breaking out of the loop we unlock_mutex and return the pcb_t proc.

1.6 Output explanations

NOTE: To run for testing scheduler through 3 inputs SCHED, SCHED_0 and SCHED_1, we had to modify something:

- First, we uncommented the line #define MM_FIXED_MEMSZ in the header file os-cfg.h (in the original source code you gave us, this line is commented).
- Second, we define #SCHED on the header file os-cfg.h also. This will be used with the next thing we introduce.
- Thirdly, we modify the read_config function in os.c source file. We commented the segment of code starting from the line ***#ifdef MLQ_SCHED*** to ***strcat(ld_processes.path[i], proc);*** in the last for-loop in this function and change to use #ifdef SCHED with some other logical lines of code that we added ourselves. If we do not do this, the input can not be read correctly by default.

1.6.1 Explain result

Now, we will explain the result, just take the output for SCHED_1 input file as an example (SCHED and SCHED_0 are the same as SCHED_1). As we use the idea of multilevel queue, we will see on the result that the processes are executed based on their priority:

- At time slot 0, there is only the process P1 so it will be executed normally.
- At time slot 4 and 6, we have 2 coming processes. However, these 2 processes' default priority are 20 and they are lower than the priority of P1 (12). Therefore, the process P1 continue being executed.
- At time slot 7, process P4 appears and its priority is 7, which is higher than process P1's priority. Now, the process P1 has to give the turn back to process P4 and wait until process P4 executed completely. However, process p1 is executing/dispatching at this time slot, P4 is actually executed at 2 time slots later (time slot 9)
- At time slot 20, process P4 is now completely executed so it will give back the turn to other process, specifically here is the process P1 (it's priority is the highest one now comparing with other 2 processes).
- At time slot 27, process P1 finishes its execution and give back turn for last 2 processes. Now, these last 2 processes have the same priority, so we use queue First-In-First-Out meaning that which process comes to the queue firstly will be executed before the other. Here because process 2 comes first at time slot 4, it will be executed.
- From now til the end, 2 processes having same priority will be executed alternately.

1.6.2 Explain Gannt Chart

- We can easily observe that the time for executing each process is limited to 2 time slots. For example, if you look at the period of time from time slot 1 to 8 in the Gannt Chart, we do not merge all block from 1 to 8 into 1 block and name it P1. Because the configure file assign a value of 2 to the time slice (the amount of time (in seconds) for which a process is allowed to run). Therefore, each process will be executed 2 time slots and then we will check if there is another coming process that has higher priority than the current executed one. If yes, we will replace the current with the new one, otherwise the current process will be continued executing.
- At time slot 19, there is only one block for P4 because now this process has finished in the first time slot, we do not need to use 2 time slots for it.

2 Memory management

2.1 Background

Memory is a term defining a collection of data in a particular format. It is used to store instructions and process data. This also plays an important role in the Operating System because we can not do anything without place to store data. When we have memory for our system, the next question is how to manage them effectively? Obviously, to manage the memory is not an easy work, this needs a perfect combination of many functions.

Usually, when we talk about memory, we do not care about the real idea behind this definition. We only use it with the meaning as the place to store data, but in fact, memory is separated into 2 types: **virtual memory** and **physical memory**. Each process has its own virtual memory section, containing single or multiple virtual memory areas (**VMA**). Each of these areas ranges continuously in [vm_start, vm_end]. However, the actual usable area is limited by the top pointing at ***sbrk***. Within VM, several memory regions exist as a representation of space for variables, and are managed through a simple symbol table as an array, with each cell storing the start and end of a region.

Meanwhile, the physical memory is the real memory hardware (RAM). We can not always evaluate the amount of memory we can use to run the process precisely so there is always a fact that RAM can be used up and if we do not have any solutions for this problem, the system will be crashed leading to data leakage, which is very dangerous. Therefore, people use SWAP to solve this (in this assignment, we use 1 RAM and 4 SWAPS). Generally, we can consider SWAP as a "virtual" RAM, which will act like a real RAM but its access speed will be much lower than actual RAM. This technique should only be used when RAM is used up and not so frequently. Using SWAP is just a back-up plan for the case of fully used RAM.

Both virtual and physical memory are separated into several fixed-size memory blocks, which are called: ***page*** in VM and ***frame*** in Physical Memory. When a process want to write to or read from the memory, it must be in RAM or it will be loaded from SWAP in case not in RAM. However, what if RAM is now full? We have to find a victim page using LRU technique (Least-recently-used, means finding the process that is least recently used before). Then, this victim page will be swapped out to SWAP and a new page will be swapped in from SWAP.

Page table is a table used for mapping between Virtual Memory and Physical Memory. This is actually an array storing page table entry (pte), which is a 32-bit unsigned integer with:

- **PAGING_PAGE_PRESENT** extracts bit number 31 representing if the page has been mapped already or not (in RAM).
- **PAGING_PAGE_SWAPPED** extracts bit number 30 representing if the page is in RAM or in SWAP.
- **PAGING_FPN** extracts bits from 0 to 12 representing the frame number in RAM the page is currently in.
- **PAGING_SWP** extracts bits from 5 to 25 representing the offset in SWAP where the page is located.

2.2 Implementation

In this section, we will be developing several crucial functions within three files: mm.c, mm-vm.c, and mm-memphy.c.

2.2.1 mm-vm.c

In the file mm-vm.c, there are five functions that need to be addressed: __alloc(), __free(), pg_getpage(), validate_overlap_vm_area(), find_victim_page().

- ***__alloc():***

  The fundamental concept behind constructing the **__alloc()** function involves designating a new memory region within a process's virtual memory area and returning the address of the newly allocated memory region.

  The procedure for implementation isn't as straightforward as simply navigating and pinpointing an appropriate space for allocation. Due to the limitations of a memory area, when we reach the VMA boundary, it's crucial to increase the memory area size using the inc_vma_limit() function. Consequently, there are two situations that we must address.

  Case 1: A suitable region is identified within the free region list using the get_free_vmrg_area() function. In this scenario, the discovered region is allocated to rgnode. Subsequently, we can update the memory region at the rgid position within the symrgtbl array.

```
struct vm_rg_struct rgnode;
  if (get_free_vmrg_area(caller, vmaid, size, &rgnode) == 0)
  {
    caller->mm->symrgtbl[rgid].rg_start = rgnode.rg_start;
    caller->mm->symrgtbl[rgid].rg_end = rgnode.rg_end;

    *alloc_addr = rgnode.rg_start;
    return 0;
    }
```

  Case 2: In the event that no appropriate region is found, it becomes essential to expand the limit of the current VMA (cur_vma) using the inc_vma_limit() function. Once the VMA limit has been successfully increased, we can proceed to update the memory region in the same manner as in Case 1, and reset the VMA's sbrk barrier.

```
  struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);
  int inc_sz = PAGING_PAGE_ALIGNSZ(size)
  int old_sbrk ;
  old_sbrk = cur_vma->sbrk;

  if( inc_vma_limit(caller, vmaid, inc_sz) == 0 ) {
    caller->mm->symrgtbl[rgid].rg_start = old_sbrk;
    caller->mm->symrgtbl[rgid].rg_end = old_sbrk + size;
    *alloc_addr = old_sbrk;

    cur_vma->sbrk = old_sbrk + inc_sz;
  }
```

- ***__free():***

  Contrary to the __alloc() function, which removes an existing memory region from the free region list, the __free() function adds a new free region at the beginning of the list. This function utilizes two key variables:

- rgnode: a new node for the free region.

- currg: the current region that is to be removed.

Once we have successfully established a new and empty memory region 'rgnode', we assign the values of rg_start and rg_end to match those of currg. We then employ the enlist_vm_freeeg_list() function to incorporate this new region into the VMA's free region list. Additionally, we liberate currg by setting its rg_start and rg_end values to 0.

```
struct vm_rg_struct *rgnode  = malloc(sizeof(struct vm_rg_struct));
  rgnode->rg_start = currg->rg_start;
  rgnode->rg_end = PAGING_PAGE_ALIGNSZ(currg->rg_end);

  enlist_vm_freerg_list(caller->mm, rgnode);

  currg->rg_end = -1;
  currg->rg_start = -1;

  return 0;
```

- ***pg_ getpage()***

  The ***pg_ getpage()*** function is a crucial component of the memory management system in an operating system that utilizes paging with swapping capabilities. This function is specifically designed to address page faults and manage the transition of pages between physical memory and swap space.

```
int pg_getpage(struct mm_struct *mm, int pgn, int *fpn, struct pcb_t *caller)
{
  uint32_t pte = mm->pgd[pgn];
  if (!PAGING_PAGE_PRESENT(pte))
  {
    fpn = NULL;
    return -1;
  }
  else {
    if(!PAGING_PAGE_IN_SWAP(pte)) {
      *fpn = PAGING_FPN(pte);
      set_page_hit_cur_to_zero(mm, pgn);
      return 0;
    }
    else {
      int tgtfpn = PAGING_SWP(pte);
      int vicfpn;
        int vicpgn, swpfpn;
        uint32_t vicpte;

        uint32_t * ret_ptbl = NULL;
        find_victim_page(caller->mm, &vicpgn, &ret_ptbl);

        vicpte = ret_ptbl[vicpgn]; //caller->mm->pgd[vicpgn];
```

```
25          if(!PAGING_PAGE_IN_SWAP(vicpte))
26            vicfpn = PAGING_FPN(vicpte);
27          else{
28            while(PAGING_PAGE_IN_SWAP(vicpte)){
29              find_victim_page(caller->mm, &vicpgn, &ret_ptbl);
30              vicpte = ret_ptbl[vicpgn];
31            }
32            vicfpn = PAGING_FPN(vicpte);
33          }
34
35          /* Get free frame in MEMSWP */
36          MEMPHY_get_freefp(caller->active_mswp, &swpfpn);
37          MEMPHY_dump(caller->mram);
38          MEMPHY_dump(caller->active_mswp);
39          __swap_cp_page(caller->mram, vicfpn, caller->active_mswp, swpfpn);
40          MEMPHY_dump(caller->mram);
41          MEMPHY_dump(caller->active_mswp);
42
43          int exist = 1;
44          if(!PAGING_PAGE_PRESENT(ret_ptbl[vicpgn]))
45            exist = 0;
46          pte_set_swap(ret_ptbl + vicpgn, 0, swpfpn);
47          if(!exist) CLRBIT(*(ret_ptbl + vicpgn), PAGING_PTE_PRESENT_MASK);
48          MEMPHY_dump(caller->active_mswp);
49          MEMPHY_dump(caller->mram);
50          MEMPHY_dump(caller->active_mswp);
51          __swap_cp_page(caller->active_mswp, tgtfpn, caller->mram, vicfpn);
52          MEMPHY_dump(caller->mram);
53          MEMPHY_dump(caller->active_mswp);
54          *fpn = vicfpn;
55
56
57          pte_set_fpn(mm->pgd + pgn, vicfpn);
58          MEMPHY_put_freefp(caller->active_mswp, tgtfpn);
59
60          struct pgn_t* tmp = global_lru;
61          while(tmp!=NULL){
62            tmp->cur=tmp->cur+1;
63            tmp=tmp->pg_next;
64          }
65          enlist_pgn_node(&global_lru, pgn, caller->mm);
66
67          caller->mm->lru_pgn = global_lru;
68
69        }
70      }
71      MEMPHY_dump(caller->mram);
```

```
72    return 0;
73  }
```

The analysis below details the operational steps and logic of the function:

- **Page Table Entry Retrieval:**
  The function begins by accessing the page table entry (PTE) for the given page number (pgn) from the process's page directory (***mm->pgd***).
- **Handling Non-Present Pages:**
  If a page is not present in memory (as determined by ***PAGING_PAGE_PRESENT***), the function sets the frame pointer **fpn** to NULL and returns -1. This indicates a page fault that requires further handling.
- **Present Page Processing:**
  For pages currently in memory (not in the swap area), the function directly retrieves and assigns the frame number to ***fpn**. If a page is in the swap space, the function initiates a sequence to bring it back into physical memory.
- **Swap Management:**
  A key operation involves calling ***find_victim_page*** to identify a suitable page for swapping out, making room in the physical memory. The function performs a series of swap operations, involving moving the identified victim page to the swap space and retrieving the needed page back into the physical memory. This process includes frequent memory dumps for state verification.
- **Page Table Update Mechanism:**
  The function updates the PTE of the victim page to reflect its new location in the swap space. It also updates the PTE of the requested page to point to its new frame in physical memory.
- **Memory Management List Enhancement:**
  The global Least Recently Used (LRU) list is updated to reflect the recent access, crucial for optimizing memory management. All pages in the LRU list have their cur value incremented, symbolizing either time passage or access frequency.

- ***validate_overlap_vm_area()***
- ***find_victim_page()***

  The ***find_victim_page*** function is an integral part of the paging system in an operating system, particularly for the implementation of the page replacement mechanism. This function employs a Least Recently Used (LRU) algorithm to manage memory efficiently.

```
1   int find_victim_page(struct mm_struct *mm, int *retpgn, uint32_t** ret_ptbl)
2   {
3     struct pgn_t *pg = global_lru;
4
5     /* TODO: Implement the theorical mechanism to find the victim page */
6     if(pg == NULL) {
7       *retpgn = -1;
8       *ret_ptbl = NULL;
9       return -1;
10    }
11    else if(pg->pg_next == NULL) {
```

```
    *retpgn = pg->pgn;
    global_lru = NULL;
  }
  else {
    int max=0;
    struct pgn_t *temp = global_lru;
    while(temp != NULL) {
      if( temp->cur > max){
        max=temp->cur;
      }
      temp = temp->pg_next;
    }

    struct pgn_t *tmp;
    while(pg!= NULL) {
      if(pg->cur == max){
        if(pg->pg_next == NULL)
          break;
        else{
          tmp = pg->pg_next;
          break;
        }
      }
      pg = pg->pg_next;
    }

    struct pgn_t *t = global_lru;
    while (t->pg_next!=pg){
      t=t->pg_next;
    }
    if (pg->pg_next == NULL)
    {
      t->pg_next=NULL;
    }
    else
    {
      t->pg_next=tmp;
    }
    *retpgn = pg->pgn;
  }
  (*ret_ptbl) = pg->owner->pgd;
  if((*ret_ptbl) == NULL) printf("ret_ptbl == NULL\n");
  else printf("ret_ptbl != NULL, victim page %d\n", (*retpgn));
  free(pg);
  mm->lru_pgn = global_lru;
  print_list_pgn(global_lru);
  return 0;
```

```
59 }
```

The following sections detail the steps involved in this function:

- **Empty LRU List:**
  If the LRU list is empty (***global_ lru*** is NULL), the function indicates failure by setting ****retpgn*** and ****ret_ ptbl*** to -1 and NULL, respectively, and returns -1.
- **Single Page Scenario:**
  In cases where the LRU list contains only one page, this page is identified as the victim, and ***global_ lru*** is set to NULL.
- **Max cur Value Identification:**
  The function traverses the LRU list to locate the page with the highest cur value, which likely indicates the least recently used page.
- **Victim Page Identification:**
  A subsequent iteration over the list locates the page with a cur value matching the previously identified maximum. This page is earmarked as the victim page.
- **Link Adjustment:**
  If the victim page is not the last in the list, the function re-links the preceding page directly to the subsequent page, bypassing the victim page.
- **Handling the Last Page:**
  If the victim page is at the end of the list, the preceding page's ***pg_ next*** is set to NULL. Output Variable Updates
- **Victim Page Number:**
  The function stores the page number of the victim page in ****retpgn***.
- **Page Table Entry Update:**
  The page table entry of the victim page's owner (pg->owner->pgd) is stored in ****ret_ ptbl***, essential for the upcoming page table modifications.
- **Freeing Memory:**
  The memory allocated for the victim page structure is freed.
- **LRU List Update:**
  The ***lru_ pgn*** field within the ***mm_ struct*** is updated to reflect the new head of the global LRU list. Post-Operation Debugging: After the removal of the victim page, the function prints the updated LRU list for verification purposes.

2.2.2 mm.c

```
1  int vmap_page_range(struct pcb_t *caller, // process call
2                                  int addr, // start address which is aligned to
                                   pagesz
3                                 int pgnum, // num of mapping page
4             struct framephy_struct *frames,// list of the mapped frames
5                struct vm_rg_struct *ret_rg)// return mapped region, the real mapped
                  fp
6  {                                          // no guarantee all given pages are
   mapped
7    //uint32_t * pte = malloc(sizeof(uint32_t));
```

```
8     //struct framephy_struct *fpit = malloc(sizeof(struct framephy_struct));
9     //int  fpn;
10    int pgit = 0;
11    int pgn = PAGING_PGN(addr);
12
13    ret_rg->rg_end = ret_rg->rg_start = addr;
14
15
16    /* TODO map range of frame to address space (virtual address) */
17
18    struct framephy_struct *fpit2 = frames;
19    for( ; pgit < pgnum; pgit++) {
20      if(fpit2 == NULL) {
21        printf("Not enough frame in alloc\n");
22        return -1;
23      }
24      if((pgn+pgit) >= PAGING_MAX_PGN) {
25        printf("Out_of_range pgd of alloc\n");
26        return -1;
27      }
28      pte_set_fpn(caller->mm->pgd + (pgn+pgit), fpit2->fpn);
29
30     /* Tracking for later page replacement activities (if needed)
31      * Enqueue new usage page */
32
33      struct pgn_t* tmp = global_lru;
34      while(tmp!=NULL)
35      {
36        tmp->cur=tmp->cur+1;
37        tmp=tmp->pg_next;
38      }
39
40      enlist_pgn_node(&global_lru, pgn+pgit, caller->mm);
41      caller->mm->lru_pgn = global_lru;
42      ret_rg->rg_end = ret_rg->rg_end + PAGING_PAGESZ;
43      fpit2 = fpit2->fp_next;
44      }
```

Operational Logic

- **Initialization:**

  The function starts by setting the initial page number (***pgn***) based on the provided starting address (***addr***). The return region (***ret_rg***) is initialized with this address.

- **Mapping Logic:**

  Iterates through the specified number of pages (***pgnum***), mapping each to a physical frame from the provided list (***frames***). If the frames list is exhausted before completing the mapping or if the page number exceeds the maximum allowed (***PAGING_MAX_PGN***), the function prints an error and returns -1.

- **Page Table Entry Update:**

  For each mapping, the function sets the frame page number (***fpn***) in the process's page directory (***caller->mm->pgd***).

- **Memory Management List Update:**

  Updates the global Least Recently Used (LRU) list to reflect the new mappings, essential for future page replacement decisions. Increments the cur value for all pages in the LRU list, indicating the passage of time or access.

- **Region End Update:**

  Updates the end address of the mapped region (***ret_rg->rg_end***) for each mapped page.

2.2.3 mm-memphy.c

For **mm-memphy.c**, there is only one function that requires us to handle which is **MEMPHY_dump()**. This function shows us the content in the physical memory.

This leads us to the final part of executing memory management section, so once handling them all we can obtain more and more concepts about how a Simple Operating System operates.

```
int MEMPHY_dump(struct memphy_struct * mp)
{
    /*TODO dump memphy content mp->storage
     *     for tracing the memory content
     */

    for(int i = 0; i < mp->maxsz; i++) {
      //if(mp->storage[i] != 0) printf("At byte %d: %c\n", i, mp->storage[i]);
    }
    return 0;
}
```

2.3 Output results - File os_0_mlq_paging

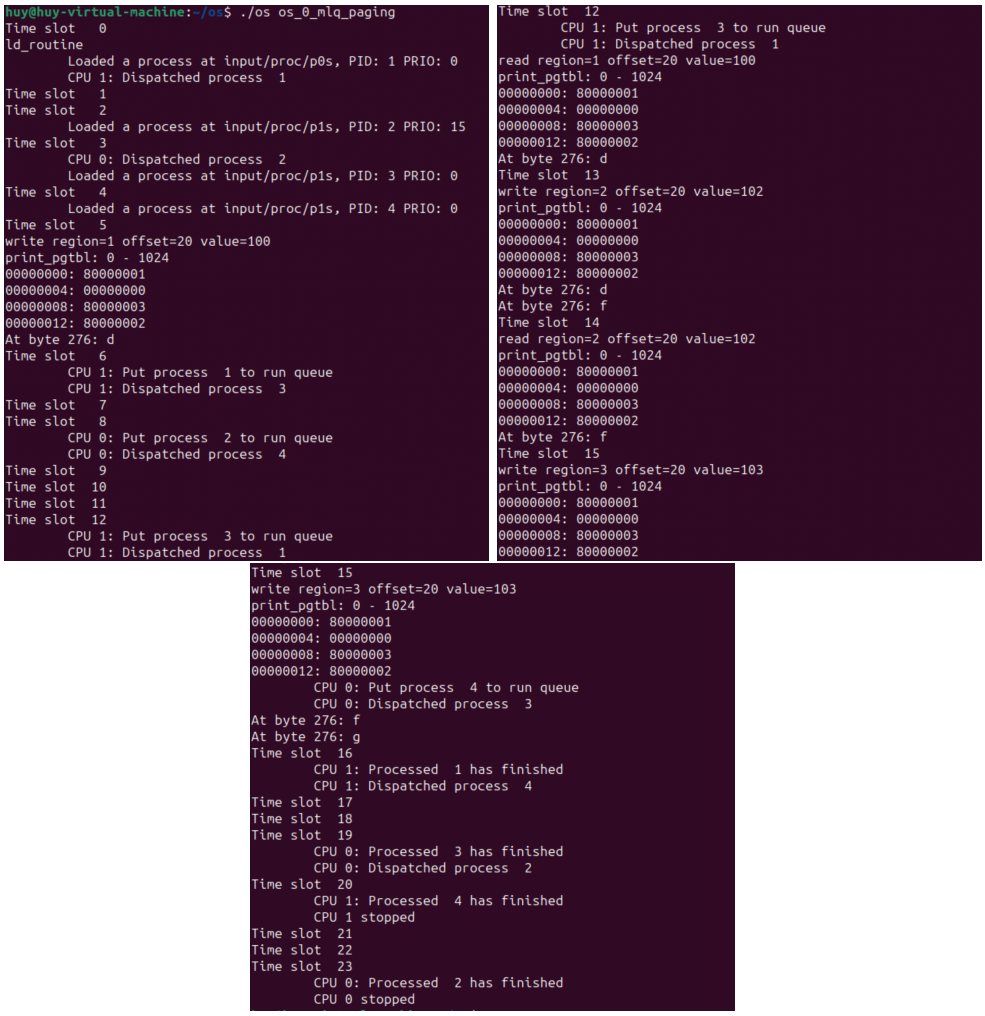


Figure 7: Result for running os_0_mlq_paging input file

This is list of instructions **p0s**:

```
1 14
calc
alloc 300 0
alloc 300 4
free 0
alloc 100 1
write 100 1 20
read 1 20 20
write 102 2 20
read 2 20 20
write 103 3 20
read 3 20 20
calc
free 4
calc
```

Now, we are about to explain instructions related to memory such as alloc, free, write, read. Since we have 4 processes; however, we just have 1 among them including those instructions mentioned above,

which is **p0s**.

- The first instruction is **alloc 300 0** that means we get to allocate the memory having the size 300 and save this region to the *symrgtbl* at the index 0. Get the virtual region in a reasonable region (We can use the **inc_vma_limit** function if needed in order to extend the virtual memory). After all, we can allocate 300-size region starting from 0 to 300 and save it to symrgtbl[0]. We also map page 0, 1 to frame number 1, 0 in RAM respectively.
- Following that is **alloc 300 4**, it operates in a similar way where we can allocate 300-size region starting from 512 to 812 and save it to symrgtbl[4]. We also map page 2, 3 to frame number 3, 2 in RAM respectively.
- For **free 0**, it means that we have to free the memory region at symrgtbl[0], so the region starting from 0 to 300 now becomes a free region.
- Then, we encounter **alloc 100 1**. Due to the fact that we have just freed the region from 0 to 300, this will allocate the 100-size region from 0 to 100 and save it to symrgtbl[1].
- **write 100 1 20** means we get to write the value 100 to the address equivalent to the starting value region at symrgtbl[1] plus offset(20), so that's why we write that value 100 to the memory at the address 20.
- After that, we use **read 1 20 20** to read 1-byte memory at the address which equals to the starting value region at symrgtbl[1] plus offset(20), so we read 1-byte memory at the address 20.
- For the next 4 lines of read and write commands, which are **write 102 2 20**, **read 2 20 20**, **write 103 3 20**, **read 3 20 20**, contain the same types of error that showed **undefined** since the region is not yet allocated.
- Finally, **free 4** determines that we need to free the memory region at symrgtbl[4], so the region starting from 512 to 812 currently becomes a free region.

2.4 Output results - File os_1_mlq_paging_small_4K

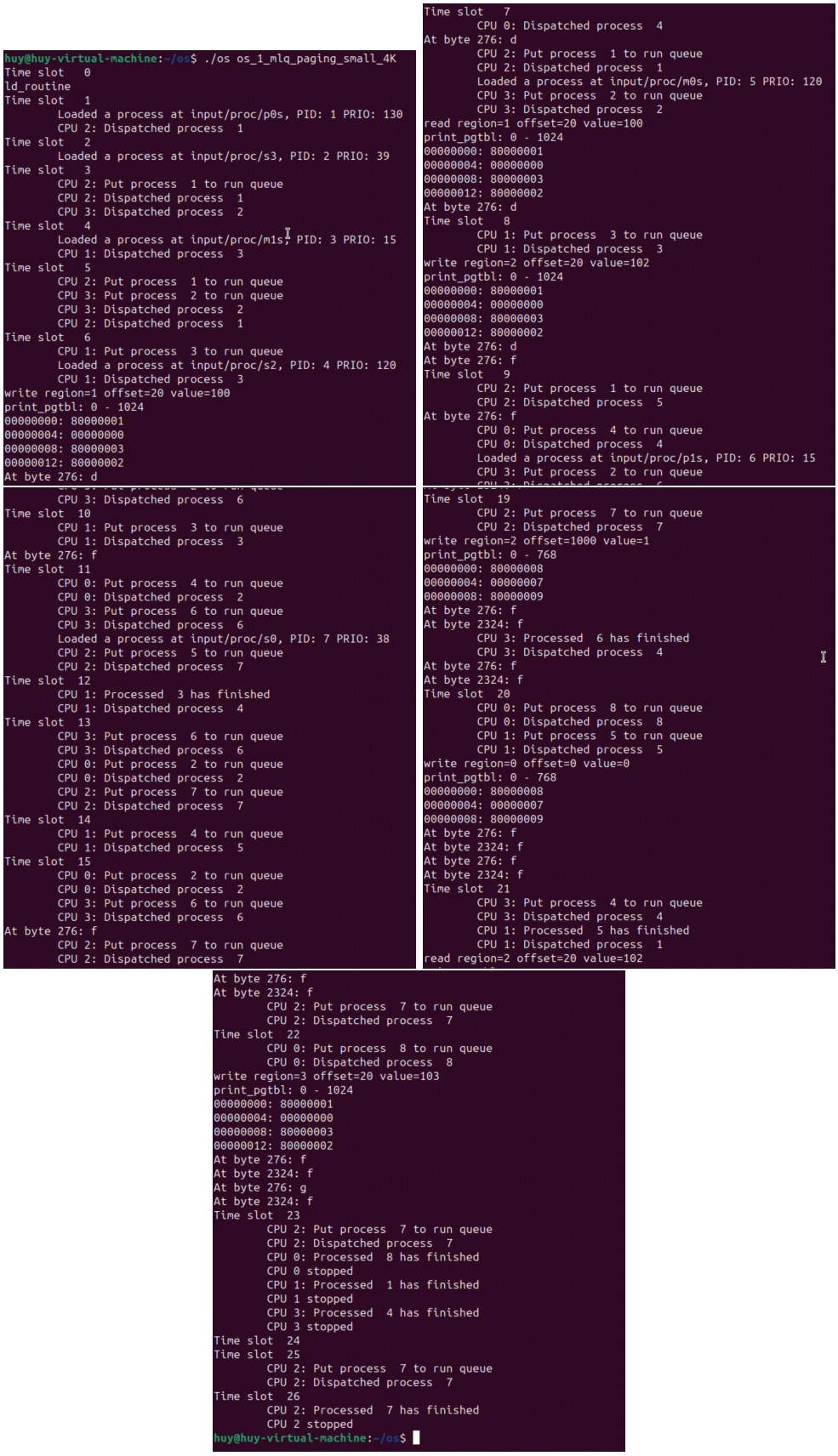


Figure 8: Result for running os_1_mlq_paging_small_4K input file

This is list of instructions **p0s**:

```
1 14
calc
alloc 300 0
alloc 300 4
free 0
alloc 100 1
write 100 1 20
read 1 20 20
write 102 2 20
read 2 20 20
write 103 3 20
read 3 20 20
calc
free 4
calc
```

This is list of instructions **m0s**:

```
1 6
alloc 300 0
alloc 100 1
free 0
alloc 100 2
write 102 1 20
write 1 2 1000
```

This is list of instructions **m1s**:

```
1 6
alloc 300 0
alloc 100 1
free 0
alloc 100 2
free 2
free 1
```

This input is similar to the previous, except it has more processes. For better understanding, we enable the VMDBG and MMDBG modules. The process **p0s** is already explained so we will discuss **m0s** and **m1s** here.

- Process 3 (**m1s**) Process 3 starts at time slot 4.
  - First instruction is **alloc 300 0**. At first, the VMA 0 has not been allocated, therefore *get_free_vmrg_area()* failed to get free region. The VMA is expanded by 2 pages, from 0 to 512B, and then mapped to frame 4 and 5 in RAM. We are then able to allocate 300B at first region to rgid 0, and update its boundary to (300 - 512).
  - Second instruction is **alloc 100 1**. We have no problem because the first region still has over 100B space left, so we take 100B from it and save to rgid 1.
  - Third instruction is **free 0**, we simply deallocate symrgtbl[0] (0 - 300).
  - Fourth instruction is **alloc 100 2**. Since we previously freed 0 - 300, we then search on this free region and get enough space for rgid 2.

  - The two final instrutions are **free 2** and **free 1**, we regain the regions 0 - 100 and 300 - 400 of rgid 2 and 1 respectively.

- Process 5 (**m0s**) Process 5 starts at time slot 7.
  - The first 4 instructions are the same as **m1s**. The same process happens for those instructions.
  - Fifth instruction is **write 102 1 20**. Write 102 to rgid 1 at offset 20. Since rgid 1 starts at 300, the real address is 320. We then extracts the pgn and offset, then find the page from RAM. The fpn is combined with offset to get the physical address. We then write data.
  - Final instruction is **write 1 2 1000**. After transforming, we get the page number of 3. However, our VMA only has 2 pages, so page 3 does not exist, resulting in an error situation. The instruction thus does not write any data.

3 Answer questions in each implementation section

3.1 Scheduler

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

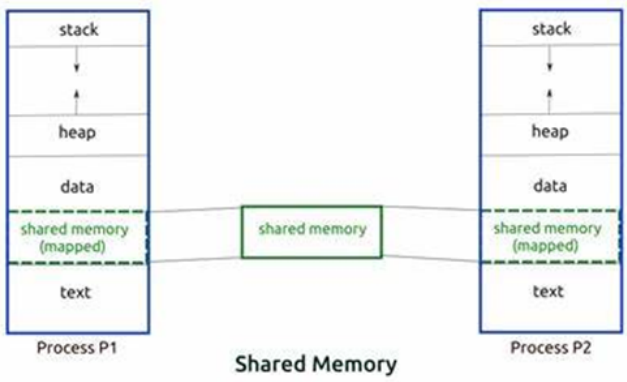
Priority feedback queue algorithms use the ideas of others algorithms such as: Priority Scheduling (every processes have a priority level to execute), Multilevel Queue (Using many queue level for processes) and Round Robin (Using quantum time for each performing process).

3.2 Memory Management

3.2.1 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?

- **Flexibility**: Segmentation provides a higher degree of flexibility than paging. Segments can be of variable size, and processes can be designed to have multiple segments, allowing for more fine-grained memory allocation.
- **Sharing**: Segmentation allows for sharing of memory segments between processes. This can be useful for inter-process communication or for sharing code libraries.



In the context of Segmentation, the shared memory technique serves as a prime example. This method allows a specific "segment" of memory to be shared among multiple processes. The advantage of this approach is its utility in enabling inter-process communication and the sharing of code libraries.

- **Protection**: Segmentation provides a level of protection between segments, preventing one process from accessing or modifying another process's memory segment. This can help increase the security and stability of the system.
- **Efficient Memory Utilization**: As a complete module is loaded all at once, segmentation improves CPU utilization and optimizes the use of available memory resources.
- **User Perception**: The user's perception of physical memory is quite similar to segmentation. Users can divide user programs into modules via segmentation. This is because the operating system job

is to map the logical address space (segments) onto the physical memory. And this set of mapping is typically stored in a table called a segment table.

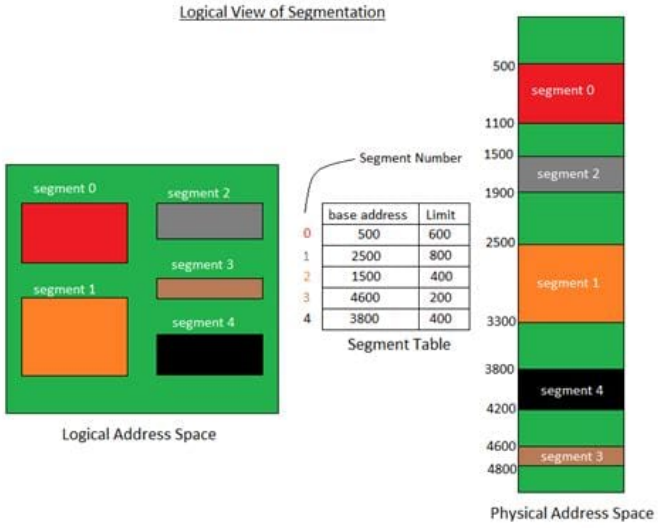


Figure 9: Logical view of Segmentation

3.2.2 The system physical memory

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

Paging is a technique that allows the physical address space of a process to be non-contiguous and divided into fixed-sized blocks called frames. The logical address space of a process is also divided into blocks of the same size called pages. A page table is used to map the pages to the frames.

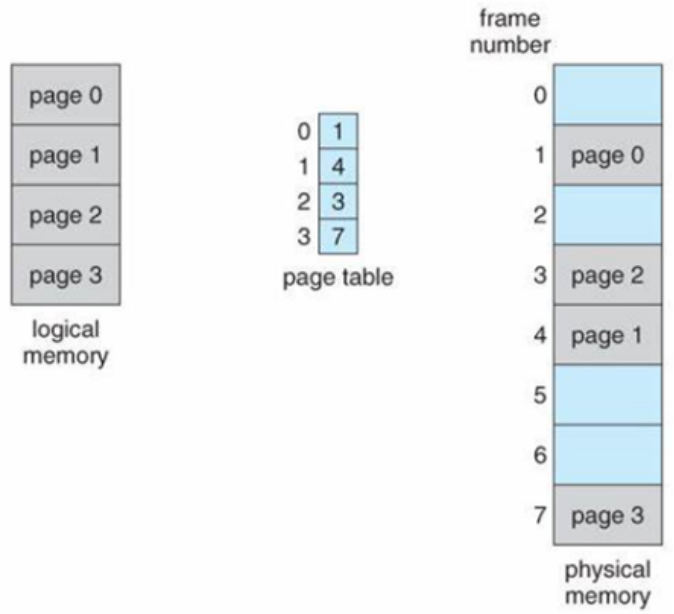
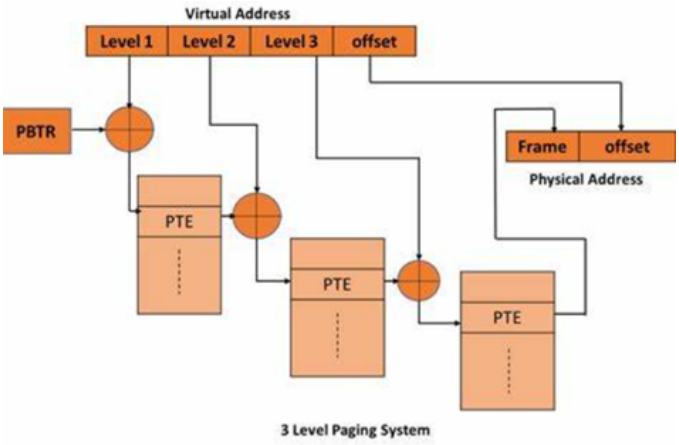


Figure 10: Page Table

If the page table is too large to fit in a single frame, it can be divided into multiple levels of page tables, such as two-level paging, three-level paging, or multilevel paging. Each level of the page table contains pointers to the next level page table, except the last level, which contains the actual frame numbers.



Dividing the address to more than two levels in the paging memory management system has some advantages and disadvantages. Some of the advantages are:

- It can reduce the size of the page table and the memory required to store it, as each level has fewer entries than a single-level page table.
- It can improve the performance and memory utilization of the system, as only the required levels of the page table need to be loaded into memory, and the rest can be swapped out to disk.
- It can support larger virtual address spaces, as more bits of the address can be used for the page number.

Some of the disadvantages are:

- It can increase the complexity and overhead of the page table lookup process, as more memory accesses are needed to traverse the levels of the page table.
- It can increase the fragmentation and waste of memory, as each level of the page table may not be fully utilized or aligned with the frame size.
- It can increase the difficulty of implementing other memory management features, such as protection, sharing, or caching.

3.2.3 Paging-based address translation scheme

Question. What is the advantage and disadvantage of segmentation with paging?

Segmentation is a way of dividing the virtual address space of a process into logical units called segments, such as code, data, stack, heap, etc. Paging is a way of dividing the physical memory into fixed-size blocks called frames, and mapping the virtual addresses to the physical addresses using a page table. Some possible advantages of segmentation with paging are:

- **Flexibility**: Segmentation allows the programmer to specify the size and layout of the segments, and paging allows the system to allocate and deallocate frames dynamically. This gives more flexibility in memory allocation and management.
- **Protection**: Segmentation allows the system to assign different protection and access rights to different segments, such as read-only, execute-only, etc. Paging allows the system to mark some pages as invalid or swapped out, and generate exceptions if they are accessed. This improves the protection and security of the memory.

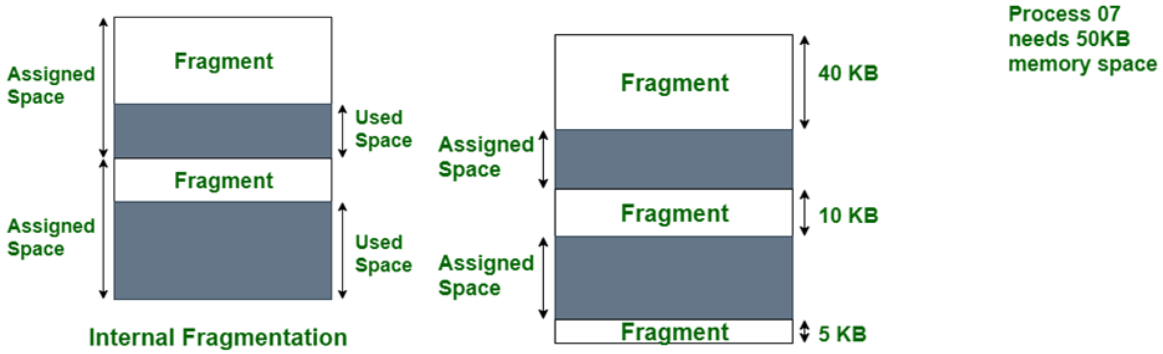


Figure 11: Internal and External Fragmentation

- **Efficiency**: Segmentation reduces the internal fragmentation caused by paging, as each segment can have a different size and fit into the available frames. Paging reduces the external fragmentation caused by segmentation, as the frames can be reused and compacted. This improves the memory utilization and performance.

Some possible disadvantages of segmentation with paging are:

- **Complexity**: Segmentation and paging require two levels of address translation, from virtual address to segment number and offset, and from segment number and offset to frame number and offset. This adds complexity and overhead to the memory management.
- **Overhead**: Segmentation and paging require additional data structures, such as segment table, page table, and page directory, to store the information about the segments and pages. These data structures consume memory space and need to be updated frequently.
- **Inconsistency**: Segmentation and paging may cause inconsistency between the logical and physical views of the memory, as the segments and pages may not be contiguous or ordered in the same way. This may affect the performance and correctness of some operations, such as memory copying, comparison, or relocation.

3.3 Put them 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.

Synchronization is important for our system, otherwise the output may be different from what we want. This is what we call "Race Conditions". Race Conditions happen when the result of a program depends on the timing and order of multiple threads or processes. Without synchronization, race conditions can cause unexpected and wrong behavior. We need to avoid unwanted interference between concurrent changes, which requires process synchronization. However, this problem can have different effects depending on the computer that runs the code. In some cases, the output is still correct even if the mutex locks are ignored, because the computer can handle synchronization to some extent, even without explicit synchronization in our code. But, in some other cases, the problem may appear and lead to wrong outcomes.

4 Conclusion

In this assignment, we have successfully executed the development of a Simple Operating System, encompassing the process scheduler, synchronization, as well as the procedure for allocating and deallocating memory from virtual to physical.

References

[1] Abraham Silberschatz, Operating System Concepts, 9th Edition

[2] Remzi H Arpaci-Dusseau - Andrea C Arpaci-Dusseau, Operating Systems - Three Easy Pieces

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

[4] GeeksforGeeks, Multilevel Feedback Queue Scheduling (MLFQ) CPU Scheduling, https://www.geeksforgeeks.org/multilevel-feedback-queue-scheduling-mlfq-cpu-scheduling/

[5] GeeksforGeeks, Introduction of Process Synchronization, https://www.geeksforgeeks.org/introduction-of-process-synchronization/

[6] GeeksforGeeks, Memory Management in Operating System, https://www.geeksforgeeks.org/memory-management-in-operating-system/

[7] GeeksforGeeks, Virtual Memory in Operating System, https://www.geeksforgeeks.org/virtual-memory-in-operating-system/

[8] GeeksforGeeks, Memory Hierarchy Design and its Characteristics, https://www.geeksforgeeks.org/memory-hierarchy-design-and-its-characteristics/

[9] GeeksforGeeks, Paged Segmentation and Segmented Paging, https://www.geeksforgeeks.org/paged-segmentation-and-segmented-paging/