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

OPERATING SYSTEMS (CO2017)

Assignment

Simple Operating System

Advisor: Le Thanh Van
Students: Nguyen Quang Minh Tien - 2150029.
Nguyen Hoang An - 2152406.

HO CHI MINH CITY, MAY 2023

Contents

1 Member list & Workload

No.	Fullname	Student ID	Problems	Percentage of work
1	Nguyen Quang Minh Tien	2150029	Synchronization, Sceduling	50%
2	Nguyen Hoang An	2152406	Memory Management	50%

2 Theory

2.1 Introduction

The purpose of this assignments is to simulate a simple operating system.
The scheduler, synchronization, and technique for assigning virtual memory to physical memory are the three key components that students will practice.
On completing this assigments, Students will learn the fundamentals of how a basic operating system functions as well as comprehend the functionnal components of the primary operating system modules. In general, this assignments will endow students with a firm foundation to further investigate and contribute to the field of operating systems.

2.2 Scheduler

Processes are split up into several queues in (Multi-Level Queue) MLQ scheduling depending on their priority, with each queue having a distinct priority level. Processes with greater priorities are put in queues with higher priority levels, while those with lower priorities are put in queues with lower priority levels..
Processes with equal priority are scheduled in FCFS (First Come First Search) order. In this assignment, the greatest priority is zero, while the lowest priority is indicated by a larger number. In accordance with the concept mentioned in assignment instruction, we define the **time slot** property for a queue. The number of times a queue may be chosen to dequeue a process within is indicated by its time slot. This implies that even when a queue contains ready processes when its time slot hits 0, the scheduler ignores it. In the case that every time slot in the queue has already been taken, we will restore it to its initial condition and repeat the process.

2.3 Memory Operations

There are 5 types of memory operations:

- `CALC`: stands for common algorithmic calculations in computers.
- `ALLOC`: allocates a section of memory from the memory device to our process.
- `FREE`: free the area assigned for certain region.
- `READ`: Access a specific area that the process has already allocated by using the command.
- `WRITE`: Access a specific area that the process has already allocated by using the command.

Since the `CALC` operation has no memory accessing part, we mainly focus on the other four.

2.3.1 ALLOC

The OS first determine if the macro MM_PAGING is defined before performing `get_free_vmrg_area`. If the macro MM_PAGING is specified, it will call method `get_free_vmrg_area` to get the free space for memory allocation.

- If a free area is discovered, it changes the caller process's memory management structures (symrgtbl) with the start and end addresses of the allocated region. Additionally, it gives the alloc_addr pointer the start address and returns 0 to signify a successful allocation. The current VMA is initially retrieved using `get_vma_by_num`

- If a free area of the requested size is not immediately available. Then it attempts to raise the virtual memory area's upper limit and runs `get_free_vmrg_area` once again to look for a free memory area of the required size. The memory management structures are updated with the proper start and end addresses, with the start address set to old_sbrk and the end address set to old_sbrk + size, if a free area is identified (the returned value is 0). The function returns 1 to indicate an allocation failure if both attempts to allocate memory fail.

2.3.2 FREE

The system tracks for the requested area in the virtual memory and cleans up memory. The detailed process is demonstrated in Section 3.2.2.

2.3.3 READ/WRITE

By method `pg_getpage`, which is particularly described in Section 3.2.3, the system accesses the destined page and either `READ` or `WRITE`.

3 Implementation

3.1 Scheduling & Synchronization

3.1.1 Questions & Answers

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

The CPU time required to allocate to processes and threads is estimated using a scheduling algorithm. The primary objective of any CPU scheduling algorithm is to maximize CPU utilization by keeping the CPU as occupied as feasible. Some scheduling algorithms we commonly use are:

- First Come First Serve (FCFS): The first process in the CPU's available sequence can access the CPU first.
- Shortest Job First (SJF): A batch-system-friendly non-preemptive method that minimizes waiting time. It executes the fastest process next.
- Longest Job First Scheduling (LJF): This method uses process burst time. The process with the longest burst duration is first in the ready queue.
- Longest Remaining Time First Scheduling (LRTF): Process the process with the most CPU time first. After some time, it will check whether a process with greater Burst Time arrives. Any process with higher burst time will preempt the current process.
- Shortest Remaining Time First (SRTF): This algorithm is preemptive SJF. This scheduling technique executes the process with the shortest remaining burst time first and may preempt it with a new task with a shorter execution time.
- Round Robin (RR): Each process is allotted quantum time to operate in this preemptive scheduling technique. One process executes for a quantum before preempting and starting another. Thus, context switching preserves preempted process states.

Priority scheduling is one of the most prevalent scheduling algorithms. Priority is assigned to each procedure. The process with the highest priority will be executed first, followed by the next highest priority, and so on. Processes with the same priority are carried out in order of arrival. Priority can be determined based on requirements for memory, time, or any other resource. Additionally, the ratio of average I/O to average CPU surge time can be used to determine priority. Yet, some common benefits includes:

- Effectiveness: The strategy allows for effective prioritization of activities without requiring a complete reclassification of the task record every time a new work is added or completed. As a result, tasks may be scheduled more quickly and efficiently, which is particularly important in real-time systems.
- Adaptability: Since priorities may be shifted in response to changes in the system's needs or standards, priority queues can aid with the efficient and adaptable scheduling of jobs. This improves the flexibility and responsiveness of work scheduling, which is particularly important in setups where priorities or duties are constantly shifting.
- Enhanced performance: A priority queue has the potential to improve system performance by allowing for more efficient use of hardware resources like CPU and RAM. This allows the system to more efficiently allocate resources, which speeds up the process of completing jobs.
- Better regulation: Since priorities may be set according to predetermined standards, this method allows for more precise work allocation. In systems with complex requirements or dependencies, this allows for more granular permission over task planning.

In conclusion, the use of a priority queue in scheduling algorithms has several advantages, including improved efficiency, scalability, performance, and control. However, the proper implementation of a priority queue will depend on the specific needs and constraints of the system under development.
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.

In a multi-threaded program, the execution of a given set of instructions may occur in a completely unexpected sequence. Having numerous strands of activity interacting with one another can produce unexpected outcomes. For instance, in our minimal operating system, the `ALLOC` command is invoked to allocate memory for two processes (which are each running in their own thread). Each procedure involves scanning the main memory for available frames and marking them as used. However, in the worst-case scenario, both processes would see the same frame as available at the same time, and both attempt to map it to their respective logical memory space.
A *race condition* happens when many threads attempt to read from or write to the same section of shared memory at the same time. Reading and writing to a picture frame is yet another illustration. Reads and writes are not necessarily atomic on all platforms and architectures. This means that it is feasible to observe values that have been written in part by one thread and in part by another. We deal with this by using mutual exclusion or mutex (`pthread_mutex_t` in C) to synchronize access to shared memory.
Liveness is a problem that arises when synchronization is not managed effectively. A liveness failure occurs when an activity reaches a state in which it is perpetually incapable of advancing. Starvation and deadlock are examples of common errors. This can contribute to poor resource utilization and stalling in an operating system. When implementing our operating system, these

issues must be addressed, particularly since most concurrency flaws do not always manifest during development and testing.
Overall, the use of threads results in a performance improvement. However, excessive use of synchronization can result in performance that is inferior to that of a comparable single-threaded program. This is due to the overhead associated with securing and unlocking, which can be detrimental during periods of high demand. In our operating system, only a subset of duties utilize shared memory and require synchronization, while others operate exclusively with private state. Due to our scheduling algorithm, for instance, only one thread at a moment executes a process (`pcb_t`). This indicates that operations on the logical memory are not susceptible to concurrency issues (since they are unique to each process). However, physical memory access must be synchronized because it is a common resource and is shared among all processes. To maintain thread safety while maintaining acceptable performance, we attempt to lock only the essential portions - mostly for `READ` and `WRITE` operations.

3.1.2 Synchronizing the process queue

3.1.3 Avoiding race conditions in vm_map_ram

```
    pthread_mutex_lock(caller->pageLock);

  struct framephy_struct *frm_lst = NULL;
  int ret_alloc;

  /*@bksysnet: author provides a feasible solution of getting frames
   *FATAL logic in here, wrong behaviour if we have not enough page
   *i.e. we request 1000 frames meanwhile our RAM has size of 3 frames
   *Don't try to perform that case in this simple work, it will result
   *in endless procedure of swap-off to get frame and we have not provide
   *duplicate control mechanism, keep it simple
   */
  ret_alloc = alloc_pages_range(caller, incpgnum, &frm_lst);

  if (ret_alloc < 0 && ret_alloc != -3000) {
    pthread_mutex_unlock(caller->pageLock);
    return -1;
  }

  /* Out of memory */
  if (ret_alloc == -3000)
  {
    pthread_mutex_unlock(caller->pageLock);
     return -1;
  }

  /* it leaves the case of memory is enough but half in ram, half in swap
   * do the swaping all to swapper to get the all in ram */
  vmap_page_range(caller, mapstart, incpgnum, frm_lst, ret_rg);
  pthread_mutex_unlock(caller->pageLock);
  return 0;
```

When `pthread_mutex_lock` is called, it blocks the current thread until the lock on the `pageLock` mutex is acquired. This ensures that only one thread can access the shared resource at a time, avoiding race conditions and data inconsistencies.

3.2 Memory Management Implementaion

3.2.1 ALLOC

3.2.1.a __alloc

The `__alloc` method will first check whether there is any free region by running function `get_free_vmrg_area`, if this function returns TRUE, indicating that the system located a suitable region, we may add appropriate attribute to that region:

```
caller->mm->symrgtbl[rgid].rg_start = rgnode.rg_start;
caller->mm->symrgtbl[rgid].rg_end = rgnode.rg_end;
*alloc_addr = rgnode.rg_start;
```

Nevertheless, if the function returns `FALSE`, means that it fails to locate any suitable space. Thus, method `inc_vma_limit` will be called to enlarge the requested area. After executing it, the new region can be update by:

```
caller->mm->symrgtbl[rgid].rg_start = old_sbrk;
caller->mm->symrgtbl[rgid].rg_end = old_sbrk + inc_sz;
*alloc_addr = old_sbrk;
```

3.2.1.b get_vma_by_num

The purpose of this function is to traverse the linked list of virtual memory areas (VMAs) for a given process and return a pointer to the VMA (pvma) that has the same ID as the provided vmaid.

```
struct vm_area_struct *get_vma_by_num(struct mm_struct *mm, int vmaid)
{
  struct vm_area_struct *pvma= mm->mmap;
  if(mm->mmap == NULL)
    return NULL;
  int vmait = 0;
  while (vmait < vmaid)
  {
    if(pvma == NULL)
    return NULL;
    pvma = pvma->vm_next;
  }
  return pvma;
}
```

3.2.1.c inc_vma_limit

When the available space in the present area is insufficient to accommodate the re-quested region, this function is called to enlarge the area. The function must first run the method validate_overlap_vm_area to see whether there is an exception for overlapping other regions. After that, the system increases the area region and performs mapping process. Method `vm_map_ram` will be used to achieve such task.

3.2.1.d vm_map_ram

This function is executed inside `inc_vma_limit` to map the frames to unmapped space in virtual area. The code is a function called vm_map_ram that maps the given address space incrementally to physical memory frames. Here's how it works:

The first step initializes two variables: frm_lst, which is a list of physical memory frames, and ret_alloc, which is an integer representing the result of allocating enough physical memory frames for mapping the virtual memory region.

Next, the `alloc_pages_range` function is called with three arguments: caller, which is a pointer to the pcb_t structure representing the process calling this function, incpgnum, which is the number of pages to be mapped, and &frm_lst, which is the address of the physical memory frame list. The `alloc_pages_range` function attempts to allocate enough physical memory frames to map the given virtual memory range incrementally. If successful, the physical frames are stored in frm_lst, and the function returns 0. If not enough memory is available, the function returns -3000 to indicate that the request failed because of insufficient memory. If any other error occurs, the function returns -1.

If ret_alloc is negative (i.e., there was an allocation error), then the function returns -1 immediately. If ret_alloc is -3000 (i.e., out of memory error), then the function returns -1 after printing an error message.

3.2.1.e alloc_pages_range

The function is responsible for allocating physical memory frames for a virtual memory region. If enough free memory frames are not available in physical memory, it performs paging out (swapping) of some of the physical memory frames to swap space in order to make enough free memory frames available in physical memory.

```
while(pgit < req_pgnum) { // collect required number of frames
      swpfpn = vicfpn = -1;
      vicpte = NULL;
      // get victim page from global list
      find_victim_page(caller->mm, &vicpte);
      vicfpn = GETVAL(*vicpte, PAGING_PTE_DIRTY_MASK, PAGING_PTE_FPN_LOBIT);
      // get free frame from active_mswp
      MEMPHY_get_freefp(caller->active_mswp, &swpfpn);
      // and move victim page to swap
      __swap_cp_page(caller->mram, vicfpn, caller->active_mswp, swpfpn);
      pte_set_swap(vicpte, 0, swpfpn);
      // add the new frame to the front of frm_lst
      fp_new = (struct framephy_struct *)malloc(sizeof(struct framephy_struct));
      fp_new->fpn = vicfpn;
      fp_new->fp_next = *frm_lst;
      *frm_lst = fp_new;
      pgit++;
    }
```

A while loop iterates over each page in the requested memory range. For each page, the `MEMPHY_get_freefp` is called. If a free frame is available in `caller->mram`, then its frame number is assigned to `*fpn`.

If there are not enough free frames in `caller->mram`, then the else part of the conditional is executed. This causes the function to swap out pages from physical memory (`caller->mram`) to swap space (`caller->active_mswp`) until enough free frames become available in `caller->mram`. Once a free frame is obtained from `caller->mram`, it is added to `frm_lst`, which is a pointer to a `framephy_struct` list of allocated frames.

3.2.1.f vmap_page_range

The function is used to map a range of virtual memory pages for a given process to a list of physical frames:

- Set the page table entry to match the current frame number.
- Enqueue the page number of a virtual memory page usage into a process's page replacement algorithm (`find_victim_page`).

```
// Allocate memory for an iterator frame:
struct framephy_struct *fpit = malloc(sizeof(struct framephy_struct));
// Initialize page iterator and calculate the corresponding page number of the
    start address.
int pgit = 0;
int pgn = PAGING_PGN(addr);
// Set the start and end regions of the returned structure to the starting address
    .
ret_rg->rg_end = ret_rg->rg_start = addr;
// Assign frame iterator to the first frame in mapped frame list
fpit = frames;
// Iterate over the mapped frame list and map each frame to a virtual memory page.
while (fpit != NULL && pgit < pgnum) {
    // Map the frame's fpn to the corresponding virtual memory page table entry.
    pte_set_fpn(&caller->mm->pgd[pgn + pgit], fpit->fpn);
    /* Tracking the page usage of the process for later replacement activities (if
     needed).
    * Enqueue new usage page. */
    enlist_pgn_node(caller->mm, caller->mm->gfifo_pgn, pgn + pgit, &caller->mm->
    pgd[pgn + pgit]);
    // Updating the iterator variables.
    fpit = fpit->fp_next;
    pgit++;
    }
// Update the end of the returned range structure.
ret_rg->rg_end = addr + pgit * PAGING_PAGESZ;
return 0;
```

3.2.2 FREE (method _free)

Frames that are mapped into a process's virtual memory will not be collected after FREE, since we do not consider garbage collection mechanism for this semester's project. However, the frames are still retained in a global FIFO list, which prevents frame loss by using a page replacement mechanism.

```
// Allocate dynamic memory for a new vm_rg_struct (memory region struct) pointer
struct vm_rg_struct * rgnode = malloc(sizeof(struct vm_rg_struct));

// If rgid is an invalid id, return -1
if(rgid < 0 || rgid > PAGING_MAX_SYMTBL_SZ)
    return -1;
// Get start and end addresses of the memory region specified by rgid
// and store it in our newly allocated vm_rg_struct pointer
rgnode->rg_start = get_symrg_byid(caller->mm, rgid)->rg_start;
rgnode->rg_end = get_symrg_byid(caller->mm, rgid)->rg_end;
// Add obsoleted memory region from rgnode to the memory free list of caller's
    virtual memory system
enlist_vm_freerg_list(caller->mm, *rgnode);
// Update vma table of caller to mark that the memory region has been freed
caller->mm->symrgtbl[rgid].rg_start = -1;
caller->mm->symrgtbl[rgid].rg_end = -1;
caller->mm->symrgtbl[rgid].rg_next = NULL;
```

It frees the memory region represented by `rgid` in the process' memory map. It checks if `rgid` is valid, retrieves the starting and ending addresses of the memory region corresponding to `rgid`,

enlists the newly freed memory region, and clears the information about the freed memory region from the `rgid` entry in the symbol table of the process' memory map. The function returns 0 upon successful operation.

3.2.3 READ/WRITE (method pg_getpage)

The function is getting the page table entry for the given `pgn` from the page directory (stored in `mm->pgd`). If the page isn't currently loaded into physical memory, then it requires some swapping. If the page is not present, the function checks if there is any free frame in RAM. If there is, it swaps the target page with the free frame using `__swap_cp_page`. The frame that previously held the target page is added to a free frame list to reuse it later.

```
1    if (MEMPHY_get_freefp(caller->mram, &tmpfpn) == 0) {
2      // ! When there's a free fp in RAM:
3      __swap_cp_page(caller->active_mswp, tgtfpn, caller->mram, tmpfpn);
4      // Put the frame previously hold the target to the free_fp_lst:
5      /* Since the tgtfp has been swapped into the RAM, we should update the frame
6      of it in SWP to free for future allocation*/
7      struct framephy_struct *fpnew = malloc(sizeof(struct framephy_struct));
8      fpnew->fpn = tgtfpn;
9      fpnew->fp_next = caller->active_mswp->free_fp_list;
10     /* Update its online status of the target page*/
11     // pte_set_fpn() & mm->pgd[pgn];
12     pte_set_fpn(&mm->pgd[pgn], tmpfpn);
13   }
```

If there are no free frames in RAM, the function needs to swap some pages out to bring in the desired page. It gets a victim page using the `find_victim_page()` function. The victim page's frame is copied to the swap space and the target page is swapped in its place. The victim page's frame is added to the free frame list:

```
1  else {
2      // ! When there is no free fp in RAM:
3      uint32_t * vicpte = NULL;
4      int swpfpn;
5      // find_victim_page(caller->mm, &vicpgn);
6      find_victim_page(caller->mm, &vicpte);
7      int vicfpn = GETVAL(*vicpte, PAGING_PTE_FPN_MASK, PAGING_PTE_FPN_LOBIT);
8      // int vicfpn = PAGING_FPN(vicpte);
9      /* Get free frame in MEMSWP */
10     MEMPHY_get_freefp(caller->active_mswp, &swpfpn);
11     /* Do swap frame from MEMRAM to MEMSWP and vice versa*/
12     /* Copy victim frame to swap */
13     __swap_cp_page(caller->mram, vicfpn, caller->active_mswp, swpfpn);
14     /* Copy target frame from swap to mem */
15     __swap_cp_page(caller->active_mswp, tgtfpn, caller->mram, vicfpn);
16     // Put the frame previously hold the target to the free_fp_lst:
17     MEMPHY_put_freefp(caller->active_mswp, tgtfpn);
18     /* Update page table */
19     pte_set_swap(vicpte, 0, swpfpn);
20     /* Update its online status of the target page */
21     // pte_set_fpn() & mm->pgd[pgn];
22     pte_set_fpn(&mm->pgd[pgn], vicfpn);
23   }
```

When the frame storing the contents of a page is currently in the swap space, it means the page is not currently in physical memory, and hence a page fault occurs. In order to access the contents on the target page, the system needs to bring the page back into physical memory. This involves performing a page replacement algorithm, which typically replaces the least recently used pages

with the desired page as part of swapping pages between physical memory (RAM) and secondary storage

3.2.4 Replacement Algorithm

3.2.4.a _find_victim_page

The method set the head of the global free page list to a variable pg of type struct pgn_t which points to the first page to be examined. A variable prev of same type as pg has been initialized with NULL.
Next, the function checks if the pg is empty or not. If it is empty, then an error message is printed and the function returns -1. Otherwise, a while loop will traverse through all the pages in the free page list until it reaches the last page. This is done by assigning the pg pointer to its own pg_next pointer and updating the prev pointer with the previous value of pg. The last page condition is checked using the last_page boolean flag, which is set to 1 by default and gets updated to 0 when the pg pointer is pointing to a page that is not the last page.
After finding the last page, the victim page number is assigned to the return pointer passed as a parameter to this function. If the last victim page is found, then the global free page list is set to NULL. Finally, the pointer to the previously attached page is also set to NULL.

```
struct pgn_t * pg = mm->global_fifo_pgn->head;
  struct pgn_t *prev = NULL;
  /* TODO: Implement the theorical mechanism to find the victim page */
  if (pg == NULL) {
    printf("[ERROR] Empty free_pg_list!\n");
    return -1;
  }
  // Collect last page from pg
  int last_page = 1;
  while (pg->pg_next != NULL) {
    prev = pg;
    pg = pg->pg_next;
    last_page = 0;
  }
  // Assign victim page number to return pointer

  // Check for last victim page condition
  if (last_page) {
    // Set page list to NULL
    mm->global_fifo_pgn->head = NULL;
    // *(mm->global_fifo_pgn) = NULL;
  }
  if(prev != NULL)
    prev->pg_next = NULL;
  // free(pg);
  *retpte = pg->pte;
```

3.2.4.b _enlist_pgn_node

This method creates a new node with the given details (page number, page table entry, caller and next pointer) and adds it to the front of the linked list (represented by `gplist`) containing all the pages kept in memory.

```
struct pgn_t *new_node = (struct pgn_t *)malloc(sizeof(struct pgn_t));

  // Assign the values of 'pgn', 'pte', 'caller' to the corresponding fields in
    the 'new_node'
```

```
4   new_node->pgn = pgn;
5   new_node->pte = pte;
6   new_node->caller = caller;
7
8   // Set the next page of the 'new_node' to the head of the global page list '
      gplist'
9   new_node->pg_next = gplist->head;
10
11  // If the 'gplist->head' is NULL, then the 'new_node' will be the first node, so
      its 'pg_next' field is set to NULL.
12  if(gplist->head == NULL)
13    new_node->pg_next = NULL;
14
15  // Set the 'gplist->head' to the newly created 'new_node', thereby adding the
      new node to the front of the global page list.
16  gplist->head = new_node;
17
18  // return 0 indicating successful enlistment of the page node to the global page
      list.
19  return 0;
```

3.2.5 Questions & Answers

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

Answer:
Separating one source code into multiple memory segments or memory areas provides a host of advantages:

- Organized Memory Management: It is effectively to organize and manage various sorts of source code by separating the memory into many regions. This segmentation makes it simpler to manage the system's memory resources by enabling improved memory allocation, tracking, and protection.
- Optimized Execution Speed: Due to their decreased size after segmentation, specified segments load into main memory more quickly.
- Effective Resource Allocation: By dynamically allocating memory according to the unique requirements of each segment, segmenting the memory may increase resource efficiency. When user processes are running, for instance, the OS may give them access to greater memory while giving the kernel or other background processes less memory. This adaptability in memory allocation guarantees effective use of resources and avoids wasting.

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

Answer:
The effects of breaking the address into more than two levels in a paging memory management system are as follows:

- More levels provide a broader accessible memory range, allowing for more processes or data to be accommodated in the same amount of address space.

- More efficient and less memory-intensive: Each level's finer granularity results in smaller page table entries.
- Additional levels provide additional indirection, necessitating more memory accesses for address translation and perhaps affecting performance.
- Increased page table fragmentation is a possibility because smaller page table entries could lead to more entries and more page table fragmentation.
- More layers complicate the memory management system, requiring complex algorithms and thus raising the possibility of errors or inefficiencies.

To conclude, granted that implementing 2-levels in the paging memory management system may have some undeniable merits, its potential risk of complexity and possible fragmentation is the driving factor for not choosing this in our OS.

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

Advantages:

- Flexible Memory Management: Memory may be logically divided into segments via segmentation. These segments can represent various program components, such as code, data, stacks, etc., or distinct processes. On the other side, paging uses fixed-size pages to offer granular memory allocation.
- Sharing and Protection: Memory segmentation offers sharing and protection features. Fine-grained control over memory access rights is possible because to the ability for each segment to have its own access permissions. Multiple processes may effectively exchange code or read-only data by sharing segments, while yet preserving distinct copies of writable data for each process.
- Address Translation Simplified: The difficulty of address translation is decreased by combining segmentation and paging. A logical-to-linear address translation is possible with segmentation, but a linear-to-physical address translation is possible with paging. When opposed to intricate memory management systems, the two translations may be carried out individually, simplifying the total address translation procedure.

Disadvantages:

- Complexity: The memory management system becomes more difficult when segmentation and paging are combined. Implementing and overseeing segmentation and paging systems is necessary, as is controlling any possible interactions or conflicts between the two methods. Designing, implementing, and debugging systems may become more difficult as a result of this extra complexity.
- External fragmentation might result from segmentation because segments can change size over time. On the other hand, because of fixed-size pages, panning might lead to internal fragmentation. If not correctly handled, the combination of these two techniques may make the fragmentation issue worse and result in ineffective memory use.
- Overhead: Using segmentation and paging together results in more memory overhead. Memory is used to store segment descriptors, page tables, and page directories in order to maintain both segmentation and paging structures. Particularly when working with large address spaces or several segments, this cost might be substantial.

- Impact on Performance: Address translation in segmentation with paging includes a number of layers of indirection, which may slow down memory access and affect system speed. To convert logical addresses to physical addresses, more memory accesses and table lookups are needed, which might cause delay and lower system performance.

Overall, segmentation with paging is an effective method of managing memory that has a myriad of advantages, such as memory sharing, flexibility, and protection. However, it may result in memory fragmentation and extra overhead and is more difficult to implement.

3.3 Demonstration

In this section, we provide a better visualization when the operating system executes by defining `DBG__` and `GANTT__` tags to get more detailed output (especially for Gantt-diagram illustration) in `os-cfg.h` :

```
#define DBG__
#define GANTT__
```

The `IO_DUMP` method allows us to inspect the value read into memory via the `READ` method, whereas the `ALLOC`, `FREE`, and `WRITE` methods return nothing. Therefore, we add some exclusive parts that prints out specific actions of a process for these actions.

```
#ifdef GANTT__
  printf("[Operation]\tPID #%d:\[_actions_]\n", proc->pid);
#endif
```

with `[_actions_]` are replaced by CALC, ALLOC, FREE, READ and WRITE whenever that operation is called.
Method `MEMPHY_dump` is implemented to observe what is written on RAM (we make the full printing conditional in case there are many things have been written on RAM):

```
#ifdef DBG__
   printf("MEMPHY_dump:\n");
   printf("|Location -- Value\n");
   for (int i = 0; i < mp->maxsz; i++) {
      if (mp->storage[i] != 0) {
         printf("|%d -- %d\n", i, mp->storage[i]);
      }
   }
   printf("-----------------------\n");
#else
    printf("MEMORY_dump.\n");
#endif
```

We use 2 bash files `gantt.sh` and `test.sh` to respectively write terminal output to `txt` file in respective folders: `gantt_output` and `os_output`.

3.3.1 Gantt Diagram

3.3.1.a Input file sched

The scheduling chart in figure 1 shows us that there are three main process operated in both CPU 0 and CPU 1. As in observed, process one starts first at time slot one due to its early arrival in the system. Following that, process operates continuously until it finishes at time slot 14, though process three arrives at time slot 12, due to its higher priority.

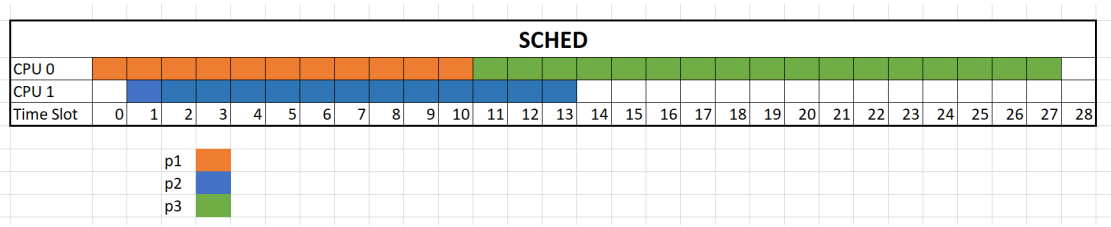

Figure 1: sched Gantt's chart

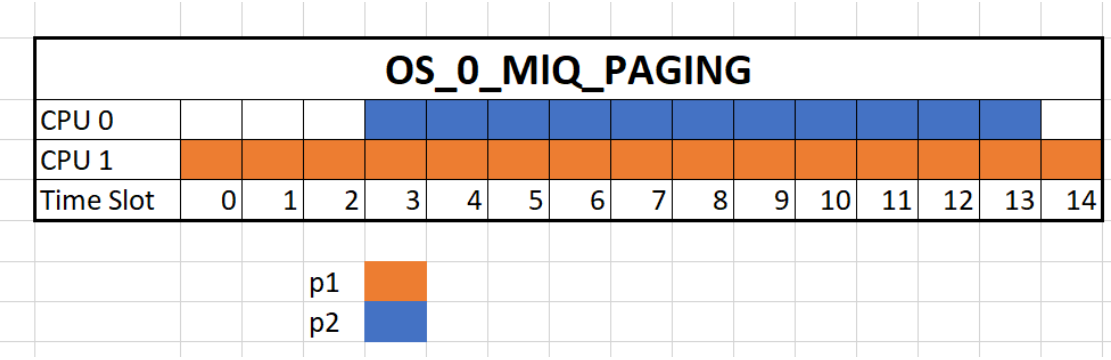

Figure 2: os_0_mlq_paging Gantt's chart

3.3.1.b Input file os_0_mlq_paging

The chart in figure 2 compares the operation of two processes running in paging. Overall, process two is run steadily till time slot 12 and finished. On the other hand, the operation of process is unstable and it finishes later although it arrives first.

3.3.1.c Input file os_0_mlq_paging

The chart in figure 3 provides us with illustration about the operation of a total number of eight processes in paging. To be general, process one is the first process to appear and run. However, process three comes and even finishes in the same CPU earlier because of its higher priority in comparison with both process one and two. Likely, process six and seven have the same way of operation as process three.

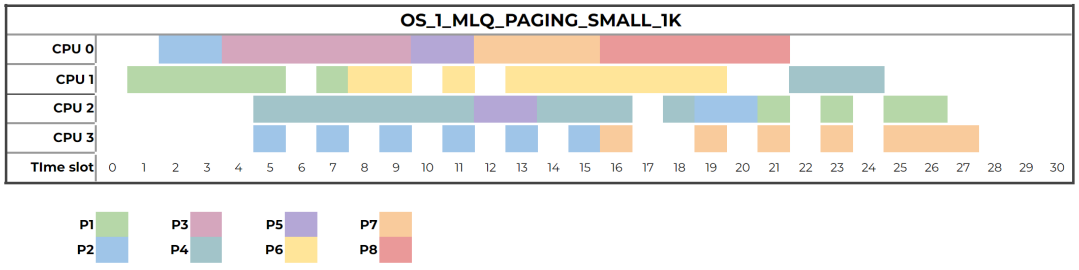

Figure 3: os_1_mlq_paging Gantt's chart

3.3.2 RAM Status

We use logging techniques to record events as they occur during the execution of memory-related procedures, providing a more complete picture of memory activity. In order to monitor the `global_fifo_pgn` after each allocation, we display the number of mapped frames and the number of free frames that remain in memory using the `RAM_dump` function in conjunction with the `print_list_mm` method.
Here we use the first 4 timeslots of `os_0_mlq_paging` to illustrate the change in `global_fifo_pgn` as well as which frame is mapped into which page number during `ALLOC`:

We also have a part for exception handling (in `WRITE` and `READ`):

```
if(currg == NULL || cur_vma == NULL )
  {
    printf("Invalid address: region not found. READ operation failed.\n");
    return -1;
  }

  if(currg->rg_start + offset >= currg->rg_end)
  {
    printf("Invalid address: out of bound. READ operation failed.\n");
    return -1;
  }
```

The first conditional statement check whether the registers or the virtual memory address are valid. Then the next conditional statement check for out of bound access. We also have this conditions printed in `os_0_mlq_paging`:

gantt_output > os_0_mlq_paging.txt

```
Time slot   0
ld_routine
    Loaded a process at input/proc/p0s, PID: 1 PRIO: 0
    CPU 1: Dispatched process  1
[Operation] PID #1: CALC
Time slot   1
[Operation] PID #1: ALLOC
[Mapping for ALLOC] PID #1 with frame mapped to 1
[Mapping for ALLOC] PID #1 with frame mapped to 0
 RAM mapping
 Number of mapped frames:   2
 Number of remaining frames:    4094
--------------------
Time slot   2
[Operation] PID #1: ALLOC
    Loaded a process at input/proc/p1s, PID: 2 PRIO: 15
[Mapping for ALLOC] PID #1 with frame mapped to 3
[Mapping for ALLOC] PID #1 with frame mapped to 2
 RAM mapping
 Number of mapped frames:   4
 Number of remaining frames:    4092
--------------------
    CPU 0: Dispatched process  2
[Operation] PID #2: CALC
Time slot   3
[Operation] PID #1: FREE
[Operation] PID #2: CALC
Time slot   4
[Operation] PID #1: ALLOC
get free vm success.
[Operation] PID #2: CALC
 RAM mapping
 Number of mapped frames:   4
 Number of remaining frames:    4092
--------------------
```

Figure 4: Demonstration of RAM and ALLOC

```
61  ----------------------
62  Time slot   7
63  [Operation] PID #1: WRITE
64  write region=2 offset=20 value=102
65  print_pgtbl: 0 - 1024
66  00000000: 80000001
67  00000004: 80000000
68  00000008: 80000003
69  00000012: 80000002
70  MEMPHY_dump:
71  |Location -- Value
72  [Operation] PID #2: CALC
73  |476 -- 100
74  ----------------------
75  Invalid address: region not found. READ operation failed.
76  Time slot   8
77  Invalid address: region not found. READ operation failed.
78  read region=2 offset=20 value=0
79  print_pgtbl: 0 - 1024
80  00000000: 80000001
81  00000004: 80000000
82  00000008: 80000003
83  00000012: 80000002
84  MEMPHY_dump:
85  |Location -- Value
86  |476 -- 100
87      CPU 0: Put process  2 to run queue
88      CPU 0: Dispatched process  2
89  [Operation] PID #2: CALC
90  ----------------------
91  Time slot   9
92  [Operation] PID #1: WRITE
93  write region=3 offset=20 value=103
94  print_pgtbl: 0 - 1024
95  00000000: 80000001
96  00000004: 80000000
97  00000008: 80000003
98  00000012: 80000002
99  MEMPHY_dump:
100 |Location -- Value
101 |476 -- 100
102 [Operation] PID #2: CALC
103 ----------------------
```

Figure 5: Exception handling

References

[1] assignment_MLQ_Paging_v20.pdf

[2] Abraham Silberschatz, Peter Baer Galvin & Greg Gagne, Operating System Concepts, 10th ed., 2018.

[3] Charles Patrick Crowley, Operating Systems: A Design Oriented Approach, 1st ed., 2006.