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

Operating Systems (CO2018)

Assignment

Simple Operating System

Advisor:	Mrs. Lê Thanh Vân
Students:	Cao Chánh Trí - 2153917
	Lê Trọng Đức - 2152523
	Trần Hoàng Khôi Tuấn - 2012359

HO CHI MINH CITY, MAY 2023

Contents

Member list & Workload

No.	Fullname	Student ID	Workloads	Percentage of work
1	Cao Chánh Trí	2153917	- Memory Management - Implementation	100%
2	Lê Trọng Đức	2152523	- Memory Management - Result Interpretation	100%
3	Trần Hoàng Khôi Tuấn	2012359	- Scheduler - Implementation	100%

1 Scheduler

1.1 Overview

Basic concept

Scheduler is a fundamental component of an operating system responsible for manages process execution, allocates system resources, and ensures efficient and fair utilization of the CPU. It determines which processes should run based on factors like priority, resource availability, and scheduling algorithms, optimizing system performance and maintaining a responsive user experience.

Our scheduler

In this simple OS, the scheduler employs multiple ready queues, each associated with a fixed priority value, to determine which process should execute when a CPU becomes available. When a new program is loaded, a new process is created, and a Process Control Block (PCB) is assigned to it. The program's content is copied to the text segment of the process, as indicated by the code pointer in the PCB. The PCB is then placed in the ready queue corresponding to the process's priority. The CPU executes processes in a round-robin style, allowing each process a time slice to run. Once the time slice expires, the CPU enqueues the process back into its associated priority ready queue and selects the next process to continue execution.

Multi-Level Queue policy

The system follows the Multi-Level Queue (MLQ) policy, where the number of traversed steps in the ready queue list is determined by a fixed formula based on the priority, known as "slot" (slot = MAX PRIO - prio). Each queue has a fixed number of slots for CPU usage, and when all slots are occupied, the system switches to the next queue, leaving the remaining work for future slots, even if it requires completing a full round of the ready queue. The scheduler's design simplifies the process of adding a process to the appropriate ready queue using priority matching. The MLQ policy is implemented through the "get proc" operation, which fetches a process and dispatches it to the CPU for execution.

1.2 Implementation

1.2.1 queue.c

In this file we need to implement the following functions:

- `enqueue()` : put a new process to a specific queue, sort the queue in ascending order of priority (higher priority process near the end of the queue).

```c
void enqueue(struct queue_t *q, struct pcb_t *proc)
{
  /* TODO: put a new process to queue [q] */
  if (q->size == MAX_QUEUE_SIZE)
    return;

  if (q->size == 0)
  {
    q->proc[0] = proc;
  }
  else
  {
    int i;
    for (i=q->size; i>0 && proc->priority >= q->proc[i-1]->priority; i--)
    {
    // In one queue, if 2 processes have the same priority number,
    // prioritize the ready one than the recently run one
    q->proc[i] = q->proc[i - 1];
    }
    q->proc[i] = proc;
  }
  q->size++;
}
```

Listing 1: enqueue()

- `dequeue()` : pop the highest priority process at the end of the queue, update the queue's `size` and `remaining_slots` .

```
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 (empty(q))
    return NULL;

  // get the highest priority element at the end of the queue
  struct pcb_t *proc = q->proc[q->size-1];
  q->proc[q->size-1] = NULL;
  q->size--;
  q->remaining_slots--;
  return proc;
}
```

Listing 2: dequeue()

The `reamining_slots` variable you see above is because we initialized it in the struct `queue_t` (in queue.h) to represent that queue's remaining slots for CPU usage.

```
struct queue_t {
  struct pcb_t * proc[MAX_QUEUE_SIZE];
  int size;
  int remaining_slots;
};
```

Listing 3: queue_t

1.2.2 sched.c

Firstly, we initialized the `HIGHEST_QUEUE` variable as index of the highest prio unempty queue in the `mlq_ready_queue` to make it easier to reset the `remaining_slot` of each queue when the scheduler completed a round of `mlq_ready_queue` .

```
#ifdef MLQ_SCHED
static struct queue_t mlq_ready_queue[MAX_PRIO];
unsigned long HIGHEST_QUEUE; //highest prio unempty queue in the current mlq
#endif
```

Listing 4: Initialize HIGHEST_QUEUE

In this file we implement the function:

- `get_mlq_proc()` : Get a process from the highest prio unempty queue in the `mlq_ready_queue` , reset the `remaining_slot` of each queue when scheduler complete a round of `mlq_ready_queue` .

```c
struct pcb_t * get_mlq_proc(void) {
  /*TODO: get a process from PRIORITY [ready_queue].
   * Remember to use lock to protect the queue.
   * */
  struct pcb_t * proc = NULL;
  pthread_mutex_lock(&queue_lock);
  // get process from the highest prio unempty queue
  for (HIGHEST_QUEUE; HIGHEST_QUEUE<MAX_PRIO; HIGHEST_QUEUE++)
    if (!empty(&mlq_ready_queue[HIGHEST_QUEUE]) && mlq_ready_queue[
    HIGHEST_QUEUE].remaining_slots >0)
    {
      proc = dequeue(&mlq_ready_queue[HIGHEST_QUEUE]);
      break;
    }
  // when the scheduler complete a round of mlq_ready_queue
  if (HIGHEST_QUEUE == MAX_PRIO)
  {
    // HIGHEST_QUEUE has passed through all the queues,
    // remaining_slot of each queue should be re-initialized
    for (int prio = MAX_PRIO - 1; prio >=0; prio--)
    {
      mlq_ready_queue[prio].remaining_slots = MAX_PRIO - prio;
      // reset HIGHEST_QUEUE, indexing the highest prio unempty queue
      if (!empty(&mlq_ready_queue[prio]))
        HIGHEST_QUEUE = prio;
    }
    // After re-initialize all conditions, pop the highest priority pcb if
    exists
    if (HIGHEST_QUEUE < MAX_PRIO)
      proc = dequeue(&mlq_ready_queue[HIGHEST_QUEUE]);
  }
  pthread_mutex_unlock(&queue_lock);
  return proc;
}
```

Listing 5: get_mlq_proc()

We also initialize a function `min` to get the smaller value between two index. We add this function to `put_mlp_proc()` so the scheduler can update the `HIGHEST_QUEUE` every time it put back a process to `mlq_ready_queue` .

```c
void put_mlq_proc(struct pcb_t * proc) {
  pthread_mutex_lock(&queue_lock);
  HIGHEST_QUEUE = min(HIGHEST_QUEUE, proc->prio); // update HIGHEST_QUEUE
    everytime put a process
  enqueue(&mlq_ready_queue[proc->prio], proc);
  pthread_mutex_unlock(&queue_lock);
}
```

Listing 6: Update the HIGHEST_QUEUE

1.3 Gantt diagram and result interpretation

1.3.1 sched_0

Input:

```
2 1 2
0 s0 1
4 s1 0
```

Listing 7: sched_0 input

Input summary table:

Process	Arrival time	Burst time	Priority queue	Priority
s0	0	15	1	12
s1	4	7	0	20

Number of CPUs: 1

Output:

```
Time slot   0
ld_routine
        Loaded a process at input/proc/s0, PID: 1 PRIO: 1
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
        Loaded a process at input/proc/s1, PID: 2 PRIO: 0
Time slot   4
Time slot   5
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
Time slot   6
Time slot   7
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot   8
Time slot   9
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  10
Time slot  11
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  12
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  1
Time slot  13
Time slot  14
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  15
Time slot  16
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  17
Time slot  18
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
```

```
Time slot  19
Time slot  20
        CPU 0: Put process  1 to run queue
        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
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Listing 8: sched_0 output

Gantt diagram:

Time slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
CPU 0		s0	s0	s0	s0	s1	s1	s1	s1	s1	s1	s1	s0	s0	s0	s0	s0	s0	s0	s0	s0	s0	s0

Explaination:

In this case only one CPU is used so only one process is executing at a time. Process s0 has lower priority queue than process s1 but it comes first. Therefore, the CPU executes s0 first, and when s1 appears, s0 has to give up the slot for s1 to be executed until it completed, then s0 can be executed the rest.

1.3.2 sched_1

Input:

```
2 2 4
0 s0 2
4 s1 0
6 s2 1
7 s3 2
```

Listing 9: sched_1 input

Input summary table:

Process	Arrival time	Burst time	Priority queue	Priority
s0	0	15	2	12
s1	4	7	0	20
s2	6	12	1	20
s3	7	11	2	7

Number of CPUs: 2

Output:

```
Time slot   0
ld_routine
        Loaded a process at input/proc/s0, PID: 1 PRIO: 3
Time slot   1
        CPU 1: Dispatched process  1
Time slot   2
Time slot   3
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
        Loaded a process at input/proc/s1, PID: 2 PRIO: 0
Time slot   4
Time slot   5
```

```
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
        CPU 0: Dispatched process  2
        Loaded a process at input/proc/s2, PID: 3 PRIO: 1
Time slot   6
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  3
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot   7
        Loaded a process at input/proc/s3, PID: 4 PRIO: 2
Time slot   8
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  3
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot   9
Time slot  10
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  3
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  11
Time slot  12
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  4
Time slot  13
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  3
Time slot  14
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  4
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  3
Time slot  15
Time slot  16
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  4
Time slot  17
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  3
Time slot  18
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  4
        CPU 1: Processed  3 has finished
        CPU 1: Dispatched process  1
Time slot  19
Time slot  20
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  4
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
Time slot  21
Time slot  22
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  4
Time slot  23
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
        CPU 0: Processed  4 has finished
        CPU 0 stopped
Time slot  24
Time slot  25
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
Time slot  26
Time slot  27
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
Time slot  28
        CPU 1: Processed  1 has finished
        CPU 1 stopped
```

Listing 10: sched_1 output

Gantt diagram:

Time slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
CPU 0						s1							s3															
CPU 1		s0					s2												s0									

Explaination:
In this case, we use 2 CPUs running concurrently. Consider CPU 0, process s1 appears at time slice 4 and has a higher priority than process s3, so when s3 appears at time slice 7, s1 continues to run until it completes, then s3 can be executed. Besides, s3 cannot replace s2 and s0 running in CPU 1 because s2 and s0 have higher priority queue and priority respectively. In CPU 1, s0 has a priority queue of 2 and s2 has a priority queue of 1, so when s2 appears at time slice 6, s0 has to give up the slot for s2 to be executed until it completes, then s0 can be executed the rest.

1.4 Answering Question

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

Some outstanding advantages of priority queue:

- Prioritization: Tasks are scheduled based on their priority, allowing critical or high-priority tasks to be executed promptly.
- Flexibility: Priorities can be dynamically adjusted during runtime, accommodating changes in task importance or urgency.
- Versatility: Priority criteria can be customized to match the specific requirements of the system, such as deadlines, resource needs, or other factors.
- Real-time support: Priority scheduling is well-suited for real-time systems with strict timing constraints.

The other scheduling algorithms have some disadvantages compared to the priority queue:

- First Come First Serve: The process requests the CPU first is allocated CPU first, other processes have to wait for it to finish before being executed. This result in longer average waiting time and Convoy effect.
- Round Robin: RR allocates a fixed time slice to each task in a cyclic manner, allowing fair sharing of CPU time among tasks. Choosing the appropriate quantum time can be challenging: too short increases context switching overhead, too long reduced responsiveness and become FCFS.

- Shortest Job First: The process with the least burst time is processed first, meaning that a long process may never get executed and the system keep executing the short processes. This result in starvation.
- Multilevel Queue: Processes are executed depending on priority of that particular level of queue to which process belongs. It leads to starvation of processes at lower levels of the multi-level queue.

In addition, priority queue scheduling can prevent starvation by using aging technique.

2 Memory Management

2.1 Overview of Memory Management

Basic concept

Virtual memory is an essential component of modern operating systems, providing the ability to run large programs with limited physical memory. It is a memory management technique that uses a combination of hardware and software to enable a process to use more memory than is physically available .Further, virtual memory abstracts main memory to an extremely large, uniform array of storage, separating logical memory as viewed by the programmer from physical memory. This technique frees programmer from physical memory.

Memory mapping

Memory mapping mechanism is a crucial aspect of virtual memory that allows the operating system to manage the virtual memory space efficiently. Memory mapping mechanism works by dividing the virtual memory space into smaller pages that can be mapped to physical memory as needed. Each page can be assigned to a physical memory location, and the mapping information is stored in a page table that the operating system maintains. When a process accesses a virtual memory address, the operating system uses the page table to map the address to the corresponding physical memory location. This process is transparent to the process and allows it to access more memory than is physically available. The use of virtual memory and memory mapping mechanism has revolutionized the way operating systems manage memory, allowing for more efficient use of available memory and enabling the execution of large programs that would not be possible otherwise.

Demand paging - Lazy Swapping

Lazy swapping, also known as demand paging, is a memory management technique used to allow programs to access more memory (RAM) than is physically available in the system. Increasing the degree of multi-programming of the system mean increasing the number of process loaded into the RAM.
Lazy swapping technique means the whole process program is not fully loaded into RAM but only the portion of a program is loaded. One part of the program is loaded into RAM, the other part is loaded into SWAP. When that portion in SWAP is needed, we will perform page swapping to load the needed pages into RAM.

Thrashing

A process is thrashing if it is spending more time paging than executing. Consider what occurs if a process does not have "enough" frames— that is, it does not have the minimum number of frames it needs to support pages in the working set. The process will quickly page-fault. At this point, it must replace some pages. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately. The higher the degree of multi-programming is, the higher the rate of thrashing will occur.

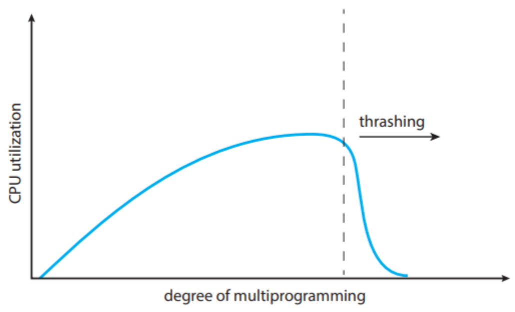

Figure 1. Thrashing

Thrashing handling with Working set Model

The working set model is a dynamic page replacement algorithm that allocates frames to a particular process assuming that the nearest future of pages will be used is a close approximation of the recent past pages in memory. This model uses a parameter Δ to define the working-set window.
If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set Δ time units after its last reference. Thus, the working set is an approximation of the program's locality.
Once Δ has been selected, the operating system monitors the working set of each process and allocates to that working set enough frames to provide it with its working-set size. If there are enough extra frames, another process can be initiated. If the sum of the working-set sizes increases, exceeding the total number of available frames, the operating system selects a process to suspend. The process's pages are swapped, and its frames are reallocated to other processes. The suspended process can be restarted later.
This working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible. Thus, it optimizes CPU utilization.

2.2 Implementation

2.2.1 Memory Allocation Procedure

ALLOC in most case, it fits into available region. If there is no such a suitable space, we need lift up the barrier sbrk and since it have never been touched, it may needs provide some physical frames and then map them using Page Table Entry.

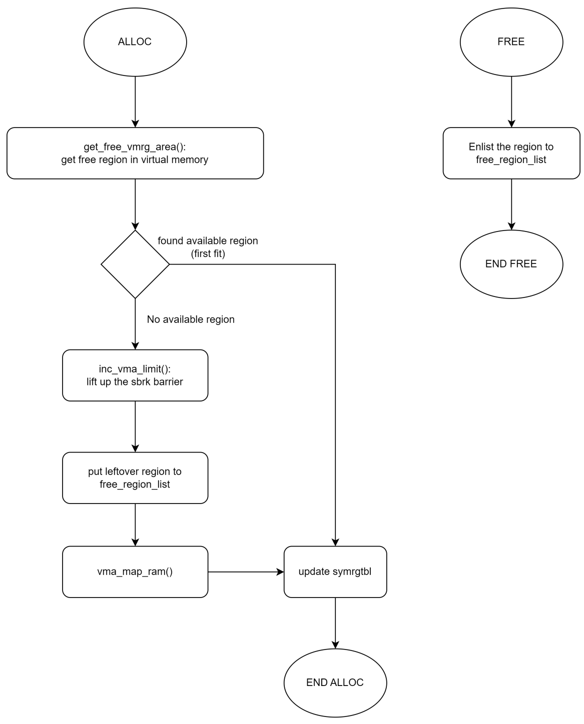

Figure 2. Memory Allocation Procedure

The approaches for lifting up the barrier sbrk can be listed: pages alignment, requested size alignment. When align to pages, the advantage of this approach is unrestrained implementation, since the one-to-one mapping to memory is easier, but can cause internal fragmentation heavily. When align to the requested size, it avoid internal fragmentation completely while user keeps allocating, but costly to manage the mapping procedure.

In out simple OS, we combined the approaches above, the barrier sbrk is lifted up aligned to pages, but do not abandon the leftover virtual memory region, instead, that region should be enlist to the `vm_free_rg_list` for later allocating instruction. As a result, it reduces internal fragmentation - better memory utilization, and keeps the simplicity of the management and mapping procedure, since it is mapped together with the whole page.

```
/*inc_vma_limit - increase vm area limits to reserve space for new variable
 *@caller: caller
 *@vmaid: ID vm area to alloc memory region
 *@inc_sz: increment size
 */
int inc_vma_limit(struct pcb_t *caller, int vmaid, int inc_sz)
{
  struct vm_rg_struct *newrg = malloc(sizeof(struct vm_rg_struct));
  int inc_amt = PAGING_PAGE_ALIGNSZ(inc_sz);
  int incnumpage = inc_amt / PAGING_PAGESZ;
  struct vm_rg_struct *area;
  struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);

  int old_end = cur_vma->vm_end;

  // leftover region starts at sbrk + requested size, end at new vm_end (aligned
```

```c
    to page)
  cur_vma->sbrk = cur_vma->vm_end + inc_sz;
  cur_vma->vm_end += inc_amt;

  area = get_vm_area_node_at_brk(caller, vmaid, inc_sz, inc_amt); // set the limit
  enlist_vm_rg_node(&caller->mm->mmap->vm_freerg_list, area); // enlist new region

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

  return 0;
}
```

Listing 11: Lift the barrier and enlist the leftover region

2.2.2 Memory Mapping

In the normal configuration where the RAM size is big enough, when a process requests for some free frames in the main memory, the struct MEMPHY will provides the according number of frames from the `free_frame_list`. However, it is not always guaranteed that there are enough frames for each process, then the Lazy Swapper mechanism will be utilized, allocating frames directly from the secondary memory, and those frames will be swapped to RAM only when they are needed.
Also, we maintain an array of flags, indicating the frames' location `frm_loc`, value 0 for RAM, 1 for SWAP.

```c
/**
 * @brief Request MEMPHY for some frames
 *
 * @param caller pcb_t
 * @param req_pgnum number of requested pages
 * @param frm_lst the return frame list
 * @param frm_loc frame location, 0 for RAM, 1 for SWAP
 * @return int
 */
int alloc_pages_range(struct pcb_t *caller, int req_pgnum, struct framephy_struct
    **frm_lst, int *frm_loc)
{
  int pgit, fpn, swpfpn;
  for (pgit = 0; pgit < req_pgnum; pgit++)
  {
    struct framephy_struct *newfp = malloc(sizeof(struct framephy_struct));
    if (MEMPHY_get_freefp(caller->mram, &fpn) == 0)
    {
      // Successfully alloc some frame in RAM
      enlist_frm_lst(caller, frm_lst, newfp, fpn);
      frm_loc[req_pgnum - pgit - 1] = 0;
    }
    else
    {
      // There is no more free frame in RAM
      // Lazy Swapper: alloc in SWAP, never swaps a page into memory unless page
    will be needed
      if (MEMPHY_get_freefp(caller->active_mswp, &swpfpn) == 0)
      {
        enlist_frm_lst(caller, frm_lst, newfp, swpfpn);
        frm_loc[req_pgnum - pgit - 1] = 1;
      }
      else
        return -3000; // No free frame in SWAP
    }
  }
  return 0;
}
```

Listing 12: Alloc pages range

From the obtained frame list, the OS now can update each page table entry to the corresponding physical frame number

```
int vmap_page_range (struct pcb_t *caller,
                     int addr,
                     int pgnum,
                     struct framephy_struct *frames,
                     struct vm_rg_struct *ret_rg,
                     int *frm_loc
                     )
{
  uint32_t *pte;
  struct framephy_struct *fpit, *temp;

  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;
  for (pgit = 0; pgit < pgnum; pgit++)
  {
    temp = fpit;
    pte = malloc(sizeof(uint32_t));
    *pte = 0;

    if (frm_loc[pgit] == 0)
    {
      pte_set_fpn(pte, fpit->fpn);
      MEMPHY_put_usefp(caller->mram, temp->fpn);
    }
    else
    {
      pte_set_swap(pte, 0, fpit->fpn);
      MEMPHY_put_usefp(caller->active_mswp, temp->fpn);
    }

    caller->mm->pgd[pgn + pgit] = *pte;
    fpit = fpit->fp_next;

    enlist_pgn_node(&caller->mm->fifo_pgn, pgn + pgit); // put in the loop to
    track all the pages

    free(temp);
    free(pte);
  }
  free(frm_loc);
  return 0;
}
```

Listing 13: Alloc pages range

2.2.3 Page Replacement

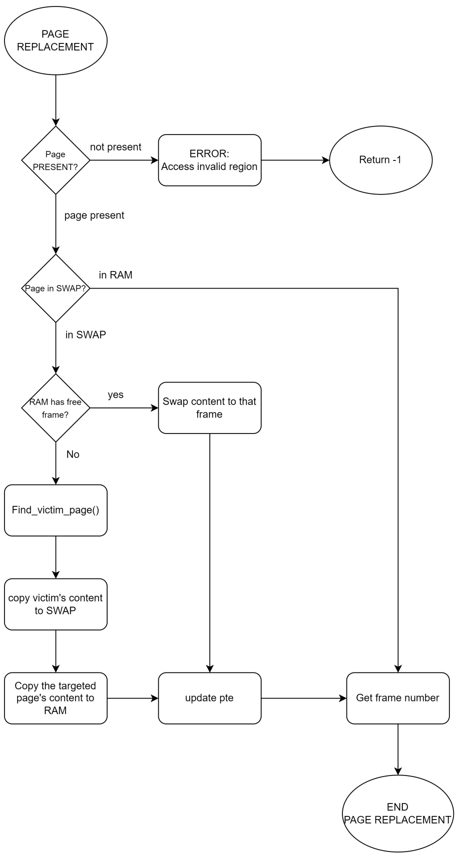

Figure 3. Page Replacement

When a Read/Write instruction is executed, the OS always have to ensure the memory location is in the main memory. If the target page is in secondary memory (i.e. present bit 1, swap bit 1), then it must be swapped into RAM to perform Read/Write instruction. Firstly, the OS requests a free frame in RAM if available, so it can copy the data to the returned frame, free the old frame in SWAP, and update the `pgd` accordingly.

If all the available RAM is consumed by other processes, we need to give back some pages to MEMSWAP device (swapping out) to make more rooms via `find_victim_page`.

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

  if (!PAGING_PAGE_PRESENT(pte))
  {
    printf("ACCESS INVALID MEMORY REGION!\n");
    return -1;
  }
  else if (PAGING_PAGE_IN_SWAP(pte))
  { /* Page is not online, make it actively living */
```

```
    int tgtfpn = PAGING_SWP(pte);
    int tmpfpn;
    if (MEMPHY_get_freefp(caller->mram, &tmpfpn) == 0)
    {
      __swap_cp_page(caller->active_mswp, tgtfpn, caller->mram, tmpfpn);
      MEMPHY_put_freefp(caller->active_mswp, tgtfpn);
      pte_set_fpn(&caller->mm->pgd[pgn], tmpfpn);
    }
    else
    {
      int vicpgn, vicfpn;
      int swpfpn;

      /* Find victim page */
      if (find_victim_page(caller->mm, &vicpgn) == -1)
        printf("Process %d: No victim page found\n", caller->pid);

      vicfpn = GETVAL(caller->mm->pgd[vicpgn], PAGING_PTE_FPN_MASK, 0);
      /* Get free frame in MEMSWP */

      if (MEMPHY_get_freefp(caller->active_mswp, &swpfpn) == -1)
        printf("Process %d: No free frame in MEMSWP\n", caller->pid);

      /* Do swap frame from MEMRAM to MEMSWP and vice versa*/
      /* Copy victim frame to swap */
      __swap_cp_page(caller->mram, vicfpn, caller->active_mswp, swpfpn);
      __swap_cp_page(caller->active_mswp, tgtfpn, caller->mram, vicfpn);

      /* Update page table */
      pte_set_swap(&caller->mm->pgd[vicpgn], 0, swpfpn);
      pte_set_fpn(&caller->mm->pgd[pgn], vicfpn);
    }
    enlist_pgn_node(&caller->mm->fifo_pgn, pgn);
  }

  *fpn = GETVAL(mm->pgd[pgn], PAGING_PTE_FPN_MASK, 0);

  put_workingset(caller->working_set, pgn);
  return 0;
}
```

Listing 14: Read/Write get page

In this simple OS, only FIFO finding victim page is applied. This algorithm has several drawbacks: high page fault rate, the computational cost increases when the queue data structure is not applied for `fifo_pgn` list, or in the worst case, the program will crash since it might return null page if a process does not have any online page initially.

For example: When many process has taken all the available RAM, a new process with low priority is now executed, and allocate some memory for the first time. The Lazy Swapping occurs, all the new allocated pages are in MEMSWAP device. Now, these pages must be swapped to the main memory for serving a Write instruction. However, since this process' `fifo_pgn` list has zero element, it cannot find any pages to swapped. This will cause a fatal error and crash the program. If the simulation scales up with more processes, Thrashing will occur, decade the CPU utilization.

```
/*find_victim_page - find victim page
 *@caller: caller
 *@pgn: return page number
 *
 */
```

```c
int find_victim_page(struct mm_struct *mm, int *retpgn)
{
  struct pgn_t *pg = mm->fifo_pgn;
  if (pg == NULL)
  {
    printf("CANNOT FIND ANY PAGE\n");
    return -1;
  }

  // There is only 1 page online
  if (pg->pg_next == NULL)
  {
    *retpgn = pg->pgn;
    mm->fifo_pgn = NULL;
    free(pg);
    return 0;
  }

  // Traverse the list
  while (pg->pg_next && pg->pg_next->pg_next)
    pg = pg->pg_next;

  *retpgn = pg->pg_next->pgn;
  free(pg->pg_next);
  pg->pg_next = NULL;

  return 0;
}
```

Listing 15: Find victim page

2.2.4 Working Set (Additional)

To address the aforementioned issue in this simple OS, a global page replacement could be considered. Although, it is a powerful solution for this problem, and can be simply implemented with `condition variable`, many followed-up questions arise: which process should be chose to be the target and why, which frame will be replaced and why...

Therefore, our alternative solution for this problem is using the Working Set model. Normally, it can be referenced by the `USER_DEFINE_BITS` in each `pte`, but we implemented this mechanism independently to be more manageable.

We use the same configuration control using constant definition, to make less effort on dealing with the interference among feature-oriented program modules.

```c
/**
 * For small sized RAM with many processes run concurrently,
 * use Working_set to avoid thrashing
 */
#define WORKING_SET 1
...
struct WorkingSet {
    int* working_set_arr;
    int max_size;
    int cursor;
};
...
void init_workingset(struct pcb_t *proc);
void increment_cursor(struct WorkingSet *working_set);
void put_workingset(struct WorkingSet *working_set, int pgn);
int is_in_working_set(struct WorkingSet *working_set, int pgn);
```

```c
void remove_from_pgn_node_list(struct pgn_t **plist, int pgn);
void put_sleep(struct pcb_t *proc);
void print_working_set(struct WorkingSet *working_set);
```

Listing 16: Working Set Interface

Each process holds a pointer to its working set, which is initialized along with that process. The window size `max_size` Δ is a fixed number depends on the RAM size and PAGE size, calculated as `Δ = RAM_SIZE / (PAGING_PAGESZ * 2)`, e.g. RAM size is 1KB, PAGE size is 256 byte, then window size $\Delta = 4$. The time interval is one cycle of CPU time. Initially, all slots are set to -1, to indicate free slot - working set is not full when value -1 still presents, and will be updated at the cursor position to be the page number on which Read/Write instruction executed.

```c
void init_workingset(struct pcb_t *proc)
{
    proc->working_set = malloc(sizeof(struct WorkingSet));
    proc->working_set->max_size = proc->mram->maxsz / (PAGING_PAGESZ * 2);
    proc->working_set->working_set_arr = malloc(sizeof(int) * proc->working_set->
    max_size);

    for (int i = 0; i < proc->working_set->max_size; i++)
        proc->working_set->working_set_arr[i] = -1;

    proc->working_set->cursor = 0;
}
```

Listing 17: Initialize Working Set

After one time slice of CPU, prior to enqueueing back to the `mlq_queue`, the process is forced `put_sleep`, to swap out the pages remaining in RAM but not present in the working set.

When the working set is not full, we do not need to swap out any pages, otherwise, loop through the `pgd` and check if the page is in RAM and not in the working set, then we copy the content of that page to a free frame in MEMSWAP, and put the RAM frame number to the free list for later usage.

```c
void put_sleep(struct pcb_t *proc)
{
    /* Working Set is not full - Return */
    if (proc->working_set->working_set_arr[proc->working_set->cursor] == -1)
        return;

    int pgn_start, pgn_end;
    int pgit;

    struct vm_area_struct *cur_vma = get_vma_by_num(proc->mm, 0);
    int end = cur_vma->vm_end;

    pgn_start = PAGING_PGN(0);
    pgn_end = PAGING_PGN(end);

    for (pgit = pgn_start; pgit < pgn_end; pgit++)
    {
        uint32_t pte = proc->mm->pgd[pgit];
        if (PAGING_PAGE_PRESENT(pte) &&
            !PAGING_PAGE_IN_SWAP(pte) &&
            is_in_working_set(proc->working_set, pgit) == -1)
        {
            printf("\t\tPut pgn %d to swap\n", pgit);

            int swpfpn;
            int ramfpn;

            ramfpn = GETVAL(pte, PAGING_PTE_FPN_MASK, 0);
            remove_from_pgn_node_list(&proc->mm->fifo_pgn, pgit);
            MEMPHY_get_freefp(proc->active_mswp, &swpfpn);
            __swap_cp_page(proc->mram, ramfpn, proc->active_mswp, swpfpn);
            pte_set_swap(&proc->mm->pgd[pgit], 0, swpfpn);
            MEMPHY_put_freefp(proc->mram, ramfpn);
        }
    }
}
```

Listing 18: Initialize Working Set

2.3 Result interpretation

2.3.1 os_0_mlq_paging

Input

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

Listing 19: os_0_mlq_paging input

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
```

Listing 20: p0s

p1s

```
1 10
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc


.
```

Listing 21: p1s

Output

```
Time slot   0
ld_routine
  Loaded a process at input/proc/p0s, PID: 1 PRIO: 0
  CPU 1: Dispatched process  1
Time slot   1
Time slot   2
  Loaded a process at input/proc/p1s, PID: 2 PRIO: 15
  CPU 0: Dispatched process  2
Time slot   3
Time slot   4
Time slot   5
write region=1 offset=20 value=100
Print frames list in RAM:
Frame num[2]  PID:1  Status:1
Frame num[3]  PID:1  Status:1
Frame num[0]  PID:1  Status:0
Frame num[1]  PID:1  Status:1

Time slot   6
  CPU 1: Put process  1 to run queue
  CPU 1: Dispatched process  1
read region=1 offset=20 value=100
Print frames list in RAM:
Frame num[2]  PID:1  Status:1
Frame num[3]  PID:1  Status:1
Frame num[0]  PID:1  Status:0
Frame num[1]  PID:1  Status:1

Time slot   7
write region=2 offset=20 value=102
Print frames list in RAM:
Frame num[2]  PID:1  Status:1
Frame num[3]  PID:1  Status:1
Frame num[0]  PID:1  Status:0
Frame num[1]  PID:1  Status:1

Write in invalid region !
Time slot   8
read region=2 offset=20 value=0
Print frames list in RAM:
Frame num[2]  PID:1  Status:1
Frame num[3]  PID:1  Status:1
Frame num[0]  PID:1  Status:0
Frame num[1]  PID:1  Status:1

Read in invalid region !
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  2
Time slot   9
write region=3 offset=20 value=103
Print frames list in RAM:
Frame num[2]  PID:1  Status:1
Frame num[3]  PID:1  Status:1
Frame num[0]  PID:1  Status:0
Frame num[1]  PID:1  Status:1

Write in invalid region !
Time slot  10
read region=3 offset=20 value=0
Print frames list in RAM:
Frame num[2]  PID:1  Status:1
Frame num[3]  PID:1  Status:1
Frame num[0]  PID:1  Status:0
Frame num[1]  PID:1  Status:1

Read in invalid region !
Time slot  11
Time slot  12
  CPU 1: Put process  1 to run queue
  CPU 1: Dispatched process  1
  CPU 0: Processed  2 has finished
  CPU 0 stopped
Time slot  13
Time slot  14
  CPU 1: Processed  1 has finished
  CPU 1 stopped
```

Listing 22: os_0_mlq_paging output

For our output, we mainly focus on interpret the RAM status. In this test case, there are two processes `p0s` and `p1s` run using 2 CPUs. The process `p1s` mainly has the instruction `calc`, therefore it doesn't have any impact to the RAM status. Mostly, we concern about the `read` and `write` instructions in process `p0s`.

- At the beginning, the two first instructions `alloc 300 0` and `alloc 300 4` create a total number of 4 pages, which is mapped to 4 frames 0, 1, 2, 3.
- Next, the `free` instruction free the the first two pages (mapped to the frames 0 and 1).
- Then, the `alloc 100` instruction comes in, this instruction doesn't allocate new pages, but it reuse the virtual region that the `alloc 300 0` had allocated before. At here it reuse page 1 (mapped to frame 1).
- Due to that, the RAM status at time slot 5 show that only frame 0 is free, other frame is still in used (value 1 is in used, value 0 is frame not used).
- The `read 1 20 20` doesn't require swapping or has any effects to the page table, hence the RAM status stay the same.
- The next four instruction `write 102 2 20` `read 2 20 20` `write 103 3 20` and `read 3 20 20`, they access the region which has not been allocated, therefore the system print the error. Moreover, the RAM status isn't affected.
- And in the rest time slots, the CPU does it work with `calc` and `free`.

2.3.2 Find Victim test

So far, we have not got any situation where `find_victim_page()` in page replacement occurs. In this test case, we focus on the behavior of the memory mapping when local page replacement happens.

Input: A small RAM with 1024 BYTE (4 frames) will be used for demonstration (PAGESZ is 256).

```
2 1 1
1024 16777216 0 0 0
0 t1 0
```

Listing 23: find victim test input

Process t1: keeps on allocating new region, forcing page fault occurs.

```
1 17
alloc 300 0
write 300 0 270
alloc 300 1
write 300 1 270
```

```
alloc 300 2
write 300 2 270
alloc 300 3
write 300 3 270
alloc 300 4
write 300 4 270
free 0
free 3
free 2
free 1
```

Listing 24: Process t1

Output

```
Time slot   0
ld_routine
  Loaded a process at input/proc/t1, PID: 1 PRIO: 0
  CPU 0: Dispatched process  1
Time slot   1
write region=0 offset=270 value=44
print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000000
Time slot   2
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot   3
write region=1 offset=270 value=44
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Time slot   4
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
NO FREE FRAME IN MEMPHY
NO FREE FRAME IN MEMPHY
Time slot   5
write region=2 offset=270 value=44
print_pgtbl: 0 - 1536
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
00000016: c0000020
00000020: c0000000
NO FREE FRAME IN MEMPHY
Time slot   6
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
NO FREE FRAME IN MEMPHY
NO FREE FRAME IN MEMPHY
Time slot   7
write region=3 offset=270 value=44
print_pgtbl: 0 - 2048
00000000: c0000040
00000004: 80000000
00000008: 80000003
00000012: 80000002
00000016: c0000020
00000020: 80000001
```

```
00000024: c0000060
00000028: c0000000
NO FREE FRAME IN MEMPHY
Time slot   8
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
NO FREE FRAME IN MEMPHY
NO FREE FRAME IN MEMPHY
Time slot   9
write region=4 offset=270 value=44
print_pgtbl: 0 - 2560
00000000: c0000040
00000004: c0000080
00000008: 80000003
00000012: 80000002
00000016: c0000020
00000020: 80000001
00000024: c0000060
00000028: 80000000
00000032: c00000a0
00000036: c0000000
NO FREE FRAME IN MEMPHY
Time slot  10
...
...
```

Listing 25: find victim test output

- At time sot 4, this process has already spanned the whole main memory, though, it keeps allocating for new region. Lazy swapper mechanism involves, the new pages will be mapped to MEMSWAP.
- Then, at time slot 5, a Write instruction is executed on page number 5, but there is no more free frame in RAM, the `find_victim_page()` activates, with fifo algorithm, the first page is targeted. Firstly, it requests a free frame in MEMSWAP, in order to copy the content of that page to, here, the next free slot has `fpn` 2, so it updates the `pgd` to be 0xC0000040. Then it can copy the content of the target page from MEMSWAP to RAM for further operation, also updates the `pgd` to be 0x8000001 (the victim it found earlier).
- The same procedures applied at time slot 7 and 9, sequentially find victim page and update the according page table entry.

2.3.3 Working set test (Additional)

We only use a simple case to analyze the working set configuration output. It is worth to note, this solution can be applied to all the input files without changing the overall behavior of the OS.

Input: The RAM size is 1024 BYTE, i.e. there are only 4 available frames (PAGESZ is 256)

```
1 2 3
1024 16777216 0 0 0
1 d1  130
2 d2  39
4 d3  15
```

Listing 26: working set test input

Process d2. (The processes d1 and d3 only have `calc` instruction.)

```
1 8
alloc 300 0
alloc 300 1
alloc 400 2
write 300 0 270
write 300 1 20
read 2 20 20
read 2 300 20
free 0
free 1
free 2
```

Listing 27: Process d2

Output

```
Time slot   0
ld_routine
...
...
Time slot   4
  Loaded a process at input/proc/d3, PID: 3 PRIO: 15
  CPU 1: Put process  1 to run queue
  CPU 1: Dispatched process  3
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  2
NO FREE FRAME IN MEMPHY
NO FREE FRAME IN MEMPHY
Time slot   5
  CPU 1: Put process  3 to run queue
  CPU 1: Dispatched process  3
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  2
write region=0 offset=270 value=44
print_pgtbl: 0 - 1536
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
00000016: c0000020
00000020: c0000000
Time slot   6
  CPU 1: Put process  3 to run queue
  CPU 1: Dispatched process  3
  CPU 0: Put process  2 to run queue
    Working set: [ 1 -1 ]
  CPU 0: Dispatched process  2
write region=1 offset=20 value=44
print_pgtbl: 0 - 1536
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
00000016: c0000020
00000020: c0000000
Time slot   7
  CPU 1: Put process  3 to run queue
  CPU 1: Dispatched process  3
  CPU 0: Put process  2 to run queue
    Working set: [ 1 2 ]
    Put pgn 0 to swap
    Put pgn 3 to swap
  CPU 0: Dispatched process  2
```

```
read region=2 offset=20 value=0
print_pgtbl: 0 - 1536
00000000: c0000040
00000004: 80000000
00000008: 80000003
00000012: c0000060
00000016: 80000002
00000020: c0000000
Time slot   8
  CPU 1: Put process  3 to run queue
  CPU 1: Dispatched process  3
  CPU 0: Put process  2 to run queue
    Working set: [ 4 2 ]
    Put pgn 1 to swap
  CPU 0: Dispatched process  2
read region=2 offset=300 value=0
print_pgtbl: 0 - 1536
00000000: c0000040
00000004: c0000000
00000008: 80000003
00000012: c0000060
00000016: 80000002
00000020: 80000000
Time slot   9
  CPU 1: Put process  3 to run queue
  CPU 1: Dispatched process  3
  CPU 0: Put process  2 to run queue
    Working set: [ 4 5 ]
    Put pgn 2 to swap
  CPU 0: Dispatched process  2
...
...
```

Listing 28: os_0_mlq_paging output

In this simple test, we focus on the working set behavior, and generalize to other scenarios when multiple processes particapate in.

- At Time slot 4, there is no more free frame in the main memory, the Lazy Swapper allows the process to allocate two frames in the secondary memory.
- In Time slot 5, a Write instruction is executed, writing on page number 1, thus, after CPU burst, the working set contains value 1. However, the working set is not full at this timestamp, it continues to progress.
- In Time slot 6, a Write instruction is executed on page number 2, and put in the working set. At this time, the working set is already full, so all the page numbers currently in the set (`pgn` 1 and 2) are kept presenting in the main memory, whilst the others get swapped out (`pgn` 0 and 3), to make room for other processes' allocations.
- After a Read instruction at time slot 7 and 8, page faults have occurred, and the targeted pages have been brought back to the main memory. The working set is updated accordingly, and the procedure continues.

Although this working set mechanism is not required in this simple OS, it appears to be capable of addressing the thrashing problem in some situations when RAM size is small or too many processes run concurrently.

2.4 Answering Question

Question 1

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 ?

There are some advantages when implement a design of multiple memory segments:

- **Efficient Use of Memory**: A process requires only a small amount of memory initially but will later require additional memory as it progresses. By using multiple memory segments, the process can allocate the required memory incrementally as it progresses, instead of allocating all the memory it might need upfront. Dividing a process into multiple segments allows for more efficient use of memory. Only the required memory for each segment is allocated, as a result, reducing external fragmentation.
- **Memory protection**: Multiple segmentation allows for better memory protection by providing a way to control access to different segments of memory. This can help to prevent security vulnerabilities by preventing one process from accessing the memory of another process.
- **Simplified Management**: Memory management is simplified with multiple memory segments. Each segment can be allocated and deallocated independently, making it easier to manage memory efficiently. Memory protection is also simplified, as different access rights can be assigned to different segments.
- **Flexibility**: Multiple memory segments provide flexibility in memory allocation. Suppose a process requires additional memory at runtime. By using multiple memory segments, a new segment can be allocated without affecting other segments, allowing the process to grow dynamically as required. This allows for efficient use of memory and helps to improve system performance.

Question 2

What will happen if we divide the address to more than 2-levels in the paging memory management system ? If we divide the address to more than 2-levels in the paging memory management system, it will result it will have some effects to the overall performance.

Advantages:

- Reduced memory overhead: Multilevel paging can help to reduce the memory overhead associated with the page table. This is because each level contains fewer entries, which means that less memory is required to store the page table.
- Faster page table lookup: With a smaller number of entries per level, it takes less time to perform a page table lookup. This can lead to faster system performance overall.
- Flexibility: Multilevel paging provides greater flexibility in terms of how the memory space is organized. This can be especially useful in systems with varying memory requirements, as it allows the page table to be adjusted to accommodate changing needs.

Disadvantages:

- Increased complexity: Multilevel paging adds complexity to the memory management system, which can make it more difficult to design, implement, and debug.
- Fragmentation: Multilevel paging can lead to fragmentation of the memory space, which can reduce overall system performance. This is because the page table entries may not be contiguous, which can result in additional overhead when accessing memory.
- Memory: Secondly, it increases the amount of memory that is used to store the page tables. This is because each level of the page table require a separate table, and each table can contain a large number of entries.

Question 3

What is the advantages and disadvantages of segmentation with paging ?

Advantages of segmentation with paging:

- Reduced fragmentation: Segmentation can reduce fragmentation by allowing program to be allocated memory in non-contiguous blocks. This is because segments can be of variable size, so programs can be allocated memory that is large or small as needed.
- Improve security: Segmentation can improve security by allowing program to be given access to only certain segments of memory. This can help to prevent program from accessing the data they should not have access to.
- Simplified memory management: Segmentation can simplify memory management by making it easier to track the location of data and programs in memory. This is because segments are logical units of memory, so they are easier to manage than pages.

Disadvantages of segmentation with paging:

- Increased complexity: Segmentation with paging can increase the complexity of the operating system, due to the fact that the operating system has to track the location of segments in memory, as well as the access permission of each segments. This can make it more difficult to develop and maintain the operating system.
- Reduced performance: Segmentation with paging can reduce performance by increasing the time it takes to access the data. This is caused by the time of translating the logical address to physical address. This translation process can add a significant amount of overhead, especially on systems with large amount of memory.
- Increase memory usage: Segmentation with paging can increase memory usage by requiring the operating system to maintain a segment table. This table store the location of each segment in memory, as well as the access permissions for each segments.

3 Synchronization

Locate the share resource

For this part we need to locate the share resource between processes and use the lock to protect them.
And we identify the main resource which all the processes use is the RAM and the SWAP. Thus,

we provide a mutex lock to protect all the `used_fp_list` and `free_fp_list` of the RAM and SWAP.
And we propose a mutex variable `mem_lock` for all the functions in `mm-memphy.c`.

```c
int MEMPHY_get_freefp(struct memphy_struct *mp, int *retfpn)
{
   pthread_mutex_lock(&mem_lock);

   struct framephy_struct *fp = mp->free_fp_list;

   if (fp == NULL)
   {
      printf("NO FREE FRAME IN MEMPHY\n");
      // fg = mp->used_fp_list;
      pthread_mutex_unlock(&mem_lock);
      return -1;
   }

   *retfpn = fp->fpn;
   mp->free_fp_list = fp->fp_next;

   /* MEMPHY is iteratively used up until its exhausted
    * No garbage collector acting then it not been released
    */
   free(fp);

   pthread_mutex_unlock(&mem_lock);

   return 0;
}
...
int MEMPHY_put_freefp(struct memphy_struct *mp, int fpn)
{
   pthread_mutex_lock(&mem_lock);

   struct framephy_struct *fp = mp->free_fp_list;
   struct framephy_struct *newnode = malloc(sizeof(struct framephy_struct));

   /* Create new node with value fpn */
   newnode->fpn = fpn;
   newnode->fp_next = fp;
   mp->free_fp_list = newnode;

   pthread_mutex_unlock(&mem_lock);

   return 0;
}
```

Listing 29: mutex lock in memphy

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.

The most common problem when not using synchronization is the **Race condition**, a situation which the resource is being access and modified by two or more processed at the same time. The race condition can cause lost data, deadlock or crashes.

And in our simple OS, we will illustrate with the following example test case when using and not-using mutex.

Input

```
1 2 2
512 16777216 0 0 0
0 sync1 0
0 sync2 0
```

Listing 30: synchonization test input

sync1

```
1 5
calc
calc
alloc 256 2
alloc 256 1
read 1 0 0
```

Listing 31: sync1 process

sync2

```
1 4
calc
alloc 256 0
free 0
read 0 5 0
.
```

Listing 32: sync2 process

Output

```
Time slot   0
ld_routine
...
...
Time slot   3
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  2
  CPU 1: Put process  1 to run queue
  CPU 1: Dispatched process  1
Time slot   4
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  2
read region=0 offset=5 value=0
print_pgtbl: 0 - 256
00000000: 80000000
Print frames list in RAM:
Frame num[1]  PID:1  Status:1
Frame num[0]  PID:1  Status:0

  CPU 1: Put process  1 to run queue
  CPU 1: Dispatched process  1
read region=1 offset=0 value=0
print_pgtbl: 0 - 512
00000000: 80000000
00000004: 80000001
Print frames list in RAM:
Frame num[1]  PID:1  Status:1
Frame num[0]  PID:1  Status:0
...
```

Listing 33: synctest with no Mutexlock

```
Time slot   0
ld_routine
...
...
```

```
Time slot   3
  CPU 1: Put process  2 to run queue
  CPU 1: Dispatched process  2
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
#### Alloc in swap
Time slot   4
  CPU 1: Put process  2 to run queue
  CPU 1: Dispatched process  2
read region=0 offset=5 value=0
print_pgtbl: 0 - 256
00000000: 80000001
Print frames list in RAM:
Frame num[1]  PID:2  Status:0
Frame num[0]  PID:1  Status:1

  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
read region=1 offset=0 value=0
print_pgtbl: 0 - 512
00000000: c0000020
00000004: 80000000
Print frames list in RAM:
Frame num[1]  PID:2  Status:0
Frame num[0]  PID:1  Status:1
...
```

Listing 34: synctest with Mutexlock

The difference is noticed from the timeslot 3, in this testcase, the output when using mutex show that we need to allocate the frame in SWAP, while the version when not using mutex doesn't allocate any frame in SWAP. This is cause due to two process can update the `free_fp_list` at the same time at timeslot 3 (race condition), and it leads to a wrong update result.

References

[1] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne *Operating System Concepts 10th edition*. Wiley.

[2] Hoang-himself *CO-2018 Asg - A repository on GitHub* https://github.com/hoang-himself/CO2018-Asg

[3] *Multilevel Paging in Operating System* https://www.geeksforgeeks.org/multilevel-paging-in-operating-system/

[4] *Belady's Anomaly in Page Replacement Algorithms* https://www.geeksforgeeks.org/beladys-anomaly-in-page-replacement-algorithms/

[5] *Virtual Memory in Operating System* https://www.geeksforgeeks.org/virtual-memory-in-operating-system/

[6] *GitHub Setup Tutorial* https://docs.github.com/en/get-started/quickstart/hello-world