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

OPERATING SYSTEMS (CO2018)

Assignment

Simple Operating System Project

Instructor(s):
Nguyễn Phương Duy , CSE - HCMUT

Team name:
Class CC03 - Group 05 - Semester 232

Student(s):
Cao Ngọc Lâm - 2252419
Trịnh Anh Minh - 2252493
Hồ Khánh Nam - 2252500
Nguyễn Châu Hoàng Long - 2252444
Lê Nguyễn Gia Bảo - 2210216

HO CHI MINH CITY, MAY 2024

Contents

1 Member list & Workload

No.	Fullname	Student ID	Works	Percentage of work
1	Cao Ngoc Lam	2252419	Implement cpu-tlb.c, cpu-tlbcache.c, write Report	20%
2	Nguyen Chau Hoang Long	2252444	Implement mm.c, mm-vm.c and mm-memphy.c, write Report	20%
3	Ho Khanh Nam	2252500	Answering Questions, show result and Write Report	20%
4	Trinh Anh Minh	2252493	Answering Questions, explain result and Write Report	20%
5	Le Nguyen Gia Bao	2210216	Implement sched.c and queue.c. Write Report	20%

2 Introduction

In today's technological advancement, the Operating System plays a crucial role in acting as an intermediary between the user and the computer hardware. As each day unfolds, the technology evolves, empowering machines to handle increasingly complex tasks with precision. Therefore, the operating system solves the problems of computer efficiency by the ability of multitasking, resource allocation and overall system management. The purpose of an operating system is to provide an environment in which a user can execute programs conveniently and efficiently.

In this assignment, we aim to simulate a simple Operating System. Our focus converges on elucidating the fundamental concepts of the scheduler, synchronization, and the mechanism allocation from virtual-to-physical memory. Particularly, we attempt to develop the operation system by implementing two main components that are responsible for managing virtual resources (the CPU(s) and RAM), which are the scheduler/dispatcher and the virtual memory engine (VME). The implementation and evaluation also depends on the instruction and requirement of the assignment.

Moreover, this assignment offers a comprehensive exploration of essential operating system concepts through practical implementation. Through evaluation and analysis, we aim to deepen our understanding of operating system design principles and enhance our ability to develop efficient and reliable computing system.

3 Scheduler

3.1 Answering Questions

Question: What is the advantage of the scheduling strategy used in this assignment in comparison with other scheduling algorithms you have learned?

Answer:

The scheduling strategy our team uses in this assignment is the **priority queue**. In here, there are many advantages of **priority queue** that we can use to apply in this assignment, that include:

- It enables the process with the higher priority to be completed first.
- A system with a priority queue is able to respond faster since higher priority processes are carried out first.
- Assist in making sure that resources are allocated to much more important processes.

However, it also has some disadvantages that is the queue may cause lower priority processes to starve.

Compared to alternative scheduling algorithms:

- **First come first serve (FCFS)**: **FCFS** is a CPU scheduling algorithm that easy to implement. However, because each process may only run in the order it comes, higher-priority processes may have to wait for a long period, which is crucial in real-time systems. It may experience the convoy effect when longer processes arrive before shorter ones.
- **Shortest job first (SJF)**: Using a non-premptive scheduling technique, **SJF** permits the shortest work to execute first. Like **FCFS**, though, the process that gets to execute may not always have the greatest priority. Additionally, determining each process's burst time can be challenging, particularly in real-time systems, and processes with long burst times run the risk of starvation.
- **Shortest Remaining Time First (SRTF**: **SRTF** is the preemptive version of shortest job first. It also have similar **advantage** and problem like **SJF**.
- **Round Robin (RR)**: Each process in the **RR** preemptive scheduling method is given a set time quantum, or time slice. Processes are carried out in a round fashion. Round robin ensures timeliness and fairness, which are crucial in real-time systems, and keeps no process from starving, in contrast to priority queue. On the other hand, a higher priority activity could have to wait for several time slices to complete, which could result in a delay.

In this assignment, we need to implement the multi-level queue which enables us to divide the scheduling process into numerous queues, each of which may have its own scheduling process and priority.

Some **advantages** of multi-level queue being:

- It enables the scheduler to have different scheduling for each queue, which is crucial for real-time systems (even though our basic operating system just utilizes round robin scheduling for every queue).
- Low scheduling overhead, as each process does not move between queues.

However, it also has some **disadvantages**:

- The inflexibility of queues prevents processes with large wait times from moving up, unlike multi-level feedback queues that allow for such movement.

3.2 Implementation

3.2.1 Queue data structure

In this program, a queue `queue_t` is a linked list of processes, specifically process control blocks (PCB). The data section of each node is a `pcb_t` struct which represents a PCB.

```
struct queue_node
{
  struct pcb_t * data;
  struct queue_node *next;
};

struct queue_t
{
  struct queue_node * head;
  struct queue_node * tail;
  int size;
};
```

However, in order to implement the desired behavior of the multi-level queue structure (which will be discussed after this subsection), we must add another property to `queue_t`. Hence, the struct's definition is extended:

```
struct queue_t
{
  struct queue_node * head;
  struct queue_node * tail;
#ifdef MLQ_SCHED
    int slot_left;
#endif
  int size;
};
```

We also make available three methods for a queue, these are: `enqueue` to append a process to the queue; `dequeue` to remove the process at front from the queue and get that process; and `empty` to check whether or not the queue currently contains any item. We won't discuss the details of the implementations of these methods as they do not deviate from widely-known implementations.

3.2.2 Scheduler data structure

The data structure changes based on the desired type of run queue.

For a linear queue, the data structured used is a linked-list queue as defined above.

```
static struct queue_t ready_queue;
```

For the multi-level queue, the data structure used is an array of linked-list queues. Each array index corresponds to a priority label which corresponds to a queue. Each queue has an amount of "health point"-like values called slot, or slot_left as it is internally called.

```
static struct queue_t mlq_ready_queue[MAX_PRIO];
```

What queue type is desired is detected by manually defining a build flag before the build process, specifically the flag MLQ_SCHED.

```
#ifdef MLQ_SCHED
static struct queue_t mlq_ready_queue[MAX_PRIO];
#else
static struct queue_t ready_queue;
#endif
```

3.2.3 Initialization

Initialize the queue. This involves manually setting the length of ready_queue to 0.

```
ready_queue.size = 0;
```

If a multi-level queue is used, we must also set each priority slot's queue length to 0 and its slot capacity to a fixed amount. We must modify and extend the procedure.

```
#ifdef MLQ_SCHED
int i ;

for (i = 0; i < MAX_PRIO; i ++) {
    struct queue_t *current_queue = &mlq_ready_queue[i];
    current_queue->size = 0;
    current_queue->slot_left = MAX_PRIO - i;
}
#endif
ready_queue.size = 0;
```

We must also initialize the mutex lock which is from the POSIX thread libraries (pthread). We assume that we've defined a queue_lock object of type pthread_mutex_t in the global program space.

```
pthread_mutex_init(&queue_lock, NULL);
```

Put it all together, we get a procedure function that will be called by the program on start-up.

```
void init_scheduler(void) {
#ifdef MLQ_SCHED
    int i ;

    for (i = 0; i < MAX_PRIO; i ++) {
        struct queue_t *current_queue = &mlq_ready_queue[i];
        current_queue->size = 0;
        current_queue->slot_left = MAX_PRIO - i;
    }
#endif
    ready_queue.size = 0;
    run_queue.size = 0;
    pthread_mutex_init(&queue_lock, NULL);
}
```

3.2.4 Getter

The function that will be exported is get_proc. There are two versions of this function, one for linear queue, and one for multi-level queue. What version is used is decided by the build flag MLQ_SCHED.

In both versions, the function are split into two sections. The first section is about retrieving a process queue item from a queue. The second section is about performing side effects on the process and queue(s).

First, the process item is retrieved from the run queue ready_queue. This will modify the queue as the item is removed from the queue.

```
struct pcb_t *retrieved_proc = dequeue(&ready_queue);
```

Then, the function determines the amount of time slots the process will run by modifying the time_slot_allow property of the process struct. It is also through this determination procedure that we implement the round-robin algorithm. The maximum amount of time the process can run is time_slot which, in the context of this program, is the quantum value in the parlance of the round-robin algorithm.

```
retrieved_proc->time_slot_allow = time_slot;
```

This time_slot_allow value will be used by our CPU routine loop to set the maximum amount of time the process will be handled by the CPU, in case the time it takes to finish a process's job may exceed this.

Because our program is multi-threaded and the CPU routines are run concurrently, this function is a critical section, so we must use a mutex lock to protect it against unintended memory access. We will lock the mutex at the start of the procedure and unlock the mutex right before the procedure ends. Put it all together, this is how a version of the get_proc function looks like.

```
struct pcb_t * get_proc(void) {
    pthread_mutex_lock(&queue_lock);
    struct pcb_t *retrieved_proc = dequeue(&ready_queue);
    retrieved_proc->time_slot_allow = time_slot;
    pthread_mutex_unlock(&queue_lock);
    return retrieved_proc;
}
```

In a multi-level queue, this procedure is more complex. In order to implement the behavior of this data structure, the procedure goes through several control blocks.

First, we identify the queue from which we retrieve a process. In our specification, the order of index corresponds to the order of priority, and lower priority value means earlier consideration, so we iterate from left to right (index ascending) and find the first queue that fits the criteria, which are:

- The queue has some item (not empty);
- The queue must has a positive slot amount.

The second criterion exists so that low-priority process collections (rightmost, high-valued indexed) are not starved.

```
struct pcb_t *proc;
struct queue_t *target_queue;
struct queue_t *iter_queue_ptr;
const struct queue_t *iter_queue_ptr_end;

proc = NULL;
target_queue = NULL;
iter_queue_ptr = mlq_ready_queue;
iter_queue_ptr_end = mlq_ready_queue + MAX_PRIO;

while (iter_queue_ptr < iter_queue_ptr_end && (empty(iter_queue_ptr) || (!empty(
    iter_queue_ptr) && iter_queue_ptr->slot_left <= 0))) {
    iter_queue_ptr++;
}

target_queue = iter_queue_ptr;
```

We then proceed to perform side effects based on the found process.

The first subprocedure tries to retrieve a process from the queue. Notice how we don't quit the procedure immediately when a not-found status is retrieved (when proc is not NULL). This is because we try to make the procedure's control flow predictable i.e. doesn't finish before the end of function definition. To compensate, most of these subprocedures only run when a process is found; we use the conditional control blocks to check if a process is found.

```
if (iter_queue_ptr >= iter_queue_ptr_end) {
    proc = NULL;
} else {
    struct pcb_t *retrieved_proc = dequeue(target_queue);
    proc = retrieved_proc;
}
```

The second subprocedure tries to set the maximum amount of time a process can run. This respects the rule where each queue has an amount of slot capacity, and each process retrieval depletes it by a time_slot amount.

```
if (proc != NULL) {
    proc->time_slot_allow = MIN(target_queue->slot_left, time_slot);
    target_queue->slot_left -= time_slot;
}
```

The third subprocedure tries to reset the slot amounts of the queues when all queues have been drained of their slots. This reset is necessary so that the anti-starvation measure mentioned above doesn't inadvertently disable the data structure permanently.

```
iter_queue_ptr = mlq_ready_queue;
iter_queue_ptr_end = mlq_ready_queue + MAX_PRIO;

while (iter_queue_ptr < iter_queue_ptr_end && iter_queue_ptr->slot_left <= 0) {
    iter_queue_ptr++;
}

if (iter_queue_ptr >= iter_queue_ptr_end) {
    int i;

    for (i = 0; i < MAX_PRIO; i++) {
        struct queue_t *current_queue = &mlq_ready_queue[i];
        current_queue->slot_left = MAX_PRIO - i;
    }
}
```

Put it all together, we get a second version of get_proc:

```
struct pcb_t * get_proc(void) {
    pthread_mutex_lock(&queue_lock);

    struct pcb_t *proc;
    struct queue_t *target_queue;
    struct queue_t *iter_queue_ptr;
    const struct queue_t *iter_queue_ptr_end;

    proc = NULL;
    target_queue = NULL;
    iter_queue_ptr = mlq_ready_queue;
    iter_queue_ptr_end = mlq_ready_queue + MAX_PRIO;

    while (iter_queue_ptr < iter_queue_ptr_end && (empty(iter_queue_ptr) || (!empty(
    iter_queue_ptr) && iter_queue_ptr->slot_left <= 0))) {
        iter_queue_ptr++;
    }

    target_queue = iter_queue_ptr;

    if (iter_queue_ptr >= iter_queue_ptr_end) {
        proc = NULL;
    } else {
        struct pcb_t *retrieved_proc = dequeue(target_queue);
        proc = retrieved_proc;
    }

    if (proc != NULL) {
        proc->time_slot_allow = MIN(target_queue->slot_left, time_slot);
        target_queue->slot_left -= time_slot;
    }

    iter_queue_ptr = mlq_ready_queue;
    iter_queue_ptr_end = mlq_ready_queue + MAX_PRIO;

    while (iter_queue_ptr < iter_queue_ptr_end && iter_queue_ptr->slot_left <= 0) {
        iter_queue_ptr++;
    }

    if (iter_queue_ptr >= iter_queue_ptr_end) {
        int i ;

        for (i = 0; i < MAX_PRIO; i ++) {
            struct queue_t *current_queue = &mlq_ready_queue[i];
            current_queue->slot_left = MAX_PRIO - i;
        }
    }

    pthread_mutex_unlock(&queue_lock);
    return proc;
}
```

3.3 Result

3.3.1 Single thread

In this test, the parameters are:

- Quantum (time_slot) equals 3;
- There are three processes, each process starts without offset and lasts for 4 time slots.

This test demonstrates how the basic queue methods and the round-robin algorithm work as intended.

- All processes are supposed to be run for 4 time slots, but each is put off when three time slots of work have been done. This is due to the quantum value being 3;
- Once put off, the processes are resumed running in the same order as when they were sequentially run i.e. $p1$ then $p2$ then $p3$. This is due to the FIFO nature of the queue.

File name is output/sched_simple_0.output

Time slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13
CPU 0		*p1*			*p2*			*p3*			*p1*	*p2*	*p3*	

3.3.2 Multi-thread

The same input parameters as above, but run on 2 threads of execution. The scheduling algorithms work independently on each thread, but it is still a shared data structure across threads.

File name is output/sched_nonsimple_0.output

Time slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13
CPU 0	*p1*			*p3*			*p3*							
CPU 1		*p2*			*p1*	*p2*								

3.3.3 Processes with priorities

The same input parameters as above and some additional parameters:

- Priority, multi-level queue mode is enabled (MLQ_SCHED flag is defined);
- The processes 1, 2, and 3 have the priority value of 2, 0, and 2, respectively.

The introduction of priorities changes the underlying scheduling algorithm, the details of which have been explained in the above "Implementation" section. These changes are evident in gantt chart below. Notice how after $p1$ is forced put to the queue, the next process to be run was $p3$ instead of $p2$ even though, according to the output log, $p3$ was loaded after $p2$. $p3$ was handled first because it has the priority value of 0 which is higher than that of $p2$ which was 2 which meant that it received first attention when the scheduling algorithm seeks a process to return.

File name is output/sched_priority_0.output

Time slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13
CPU 0		*p1*			*p3*			*p3*	*p2*			*p1*	*p2*	

4 Memory Management and Translation lookaside buffer

4.1 Answering Questions

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:

- Improves memory management by splitting available memory into smaller parts that may be allocated and de-allocated individually.
- Allows you to isolate distinct forms of memory consumption, such as code, data, and stack, for easier organization and optimization.
- Makes it possible to use memory more effectively by avoiding the requirement to allocate a single, big block of memory and instead enabling smaller memory segments to be allocated as needed.
- Makes memory protection and access control easier by enabling varying permissions and access levels for distinct memory portions.
- Enables greater memory use and performance by facilitating memory mapping, which allows various segments to be mapped to different regions of the virtual address space.

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

Answer:

- Greater flexibility in managing memory space is made possible by the paging memory management system's usage of more than two levels.
- Improved memory allocation: Allocating memory resources more correctly helps decrease waste.
- Increased complexity: As the number of levels increase, the complexity of the memory management system also increases. This could lead to a decrease in performance or an increase in the likelihood of errors.
- Increased overhead: As the number of levels increases, the system's overhead for monitoring and controlling the various paging memory management system levels also increases.
- Increased memory access time: As the number of levels rises, so does the time it takes to reach a specific memory location, potentially resulting in performance loss.

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

Answer:

Segmentation with paging is the combination of the 2 techniques: Segmentation and Paging. There are 2 most popular non-contiguous memory allocation techniques: Segmented Paging and Paged Segmentation

1. Segmented Paging:

Segmented paging is a memory management technique that the physical memory is divided into pages, and then maps each logical address used by a process to a physical page. Segments are used to map virtual memory addresses to physical memory addresses, rather than dividing the virtual memory into pages.

Advantages:

- Memory usage reduction: by dividing the memory into fixed size of blocks, it can avoid external fragmentation and makes the memory allocation become more simpler.
- Improving memory utilization: since the size of each segment can be varied with the number of pages it's mapped, a program can have multiple segment with varying size but still be based on paging mechanism for allocating each segment.

- Protection and security: the virtual space of each process can be isolated, which makes more secured for the program
- Allow the process to use more memory than the amount of physical memory space by utilizing the mechanism of swapping, which is to swap the unused page to the hard disk

Disadvantages:

- Internal fragmentation: Internal fragmentation can still occur within a page if the data doesn't fit perfectly
- Slower speed: the translation becomes more sequential, increasing the memory access time for the page content and page table to look up
- Large memory overhead: the segmented page requires to maintain both segment tables and page table, which can consume more memory compared to a single table used in only segmentation or paging

2. Paged Segmentation:

Paged Segmentation is a memory management technique that the process's address space is divided into segments and each segment is divided into pages. This allows for a flexible allocation of memory, where each segment can have a different size, and each page can have a different size within a segment.

Advantages:

- Eliminating external fragmentation and reducing the page table size
- the paged segmentation uses the swapping mechanism to utilize more memory, which is similar to segmented paging

Disadvantages:

- The problem of internal fragmentation can still occur and is not be solved completely. However, the probability is less
- Hardware is more complex than the segmented paging scheme
- The paged segmentation also increases the memory overhead, which is the same as the segmented paging
- The delays can occur when accessing the memory, since the extra level of paging at the first stage adds to the delay in memory access

Question: What will happen if the multi-core system has each CPU core can be run in a different context, and each core has its own MMU and its part of the core (the TLB)? In modern CPU, 2-level TLBs are common now, what is the impact of these new memory hardware configurations to our translation schemes?

Answer:

1. There are some advantages in running process if the multi-core system has each CPU core can be run in a different context with its own MMU and TLB:

- Improving efficiency and security in memory access: Since the MMU of each core is isolated from the others, the errors that may occur in core's address space will not affect the other core. Therefore, the memory access can be independently handled and protected
- Increasing the memory access speed: Each core with its own TLB will makes the translation of virtual addresses to physical addresses more faster, instead of having only one TLB for all cores in the memory management. It also reduces latency and increases memory access speed.
- Enhancing parallel processing: When each CPU core is run in different context, it enables to run processes and threads concurrently without causing interference, which boosts the system's ability for parallel processing.

To conclude, by letting each core run with its own MMU and TLB, this scheme will improve the memory management with better memory access efficiency and security, and enhancing parallel processing capacity.

2. The 2-level TLB is a hierarchical structure that divides the TLB into 2 level: the smaller and faster L1 TLB, and the larger but slower L2 TLB. If the an address is not found in L1 TLB (also known as the TLB miss), the L2 TLB can be checked before getting access to the main memory page table. By using this TLB structure scheme, it will improve the address translation efficiency by increasing hit rates in the TLB structure. Moreover, the 2-level TLB can decrease the memory access time by using the concept of locality, which is to use the L1 TLB as the location keeping the address that recently accessed by the CPU, and the L2 TLB to hold a larger amount of address. However, due to the fact that this is a multiple level TLB, it comes with trade-offs, i.e. increasing the complexity in memory hierarchy design.

4.2 Implementation

To build the memory management component of this minimal operating system, certain critical functions must be implemented in specific files. In the `mm-vm.c` file, the following functions are required: `__alloc()`, `__free()`, `pg_getpage()`, and `find_victim_page()`. In accordance with the first-in-first-out (FIFO) algorithm, these functions are in charge of memory allocation, clearing memory regions, retrieving pages from RAM, and identifying victim pages.

In `mm.c` file the following functions need to be implemented: `vmap_page_range()` and `alloc_pages_range()`. These routines are necessary for allocating the necessary number of frames in RAM and mapping a range of pages at an aligned address, respectively. The operating system's memory management module will be able to carry out crucial tasks like memory allocation, de-allocation, page retrieval, and victim page selection by putting these functions into the relevant files. This will enable effective memory management and utilization within the system.

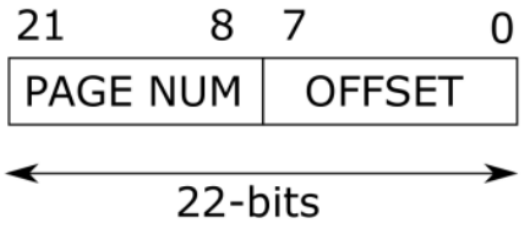
4.2.1 Analysis of Support and Macro Functions

4.2.1.a Suport Functions

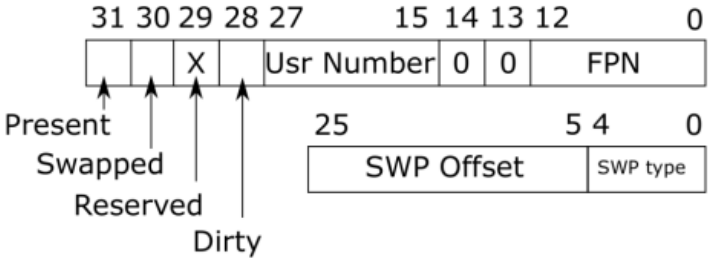
- `enlist_pgn_node(struct pgn_t **plist, int pgn)`: primarily used to expand the FIFO page list with more pages.
- `enlist_framephy_node(struct framephy_struct **framephylist, int fpn)`: mostly used for adding more pages into the used or free page list of RAM or Active Swap.
- `enlist_vm_rg_node(struct vm_rg_struct **rglist, struct vm_rg_struct* rgnode)`: mostly used to expand the virtual memory's free region list with more regions.
- `MEMPHY_get_freefp(struct memphy_struct *mp, int *retfpn)`: retrieves a free frame from the memory physical structure by its number from variable `retfpn`.
- `pte_set_fpn(unit32_t *pte, int fpn)`: We set the page table entry (`pte`) to the specified frame number if the `pte` indicates that the page is in RAM.
- `pte_set_swap(unit32_t *pte, int swptyp, int swpoff)`: if the page table entry (pte) is in `Swap`, sets it with the inputted swap type and swap offset.
- `swap_cp_page(struct memphy_struct *mpsrc, int srcfpn, struct memphy_struct *mpdst, int dstfpn)`: swaps and copies the content page from the source frame to the destination frame.

4.2.1.b Used Marco

- `GETVAL(a,b,c)`: apply bitwise - and between a and b, then take the result and shift it `c` bits to the right: `(a & b) >>c`



- `PAGING_MAX_SYMTBL_SZ`: the maximum size of symbol table (30)
- `PAGING_PAGESZ`: size of a page (256 bytes)
- `PAGING_PGN(addr)`: is defined as `GETVAL(x,PAGING_PGN_MASK,PAGING_ADDR_PGN_LOBIT)`. On the CPU address with `PAGING_PGN_MASK: 11...1100...00` (composed of 14 high-bits 1 and 8 low-bits 0), this macro employs bitwise-and (&) operation. The page number is obtained by shifting the result of `PAGING_ADDR_PGN_LOBIT` (8) bits to the left after performing a bitwise-and operation on the CPU address using this binary number.
- `PAGING_FPN(pte)`: This macro probably has a mistake in the original code. As a result, we used the `GETVAL` macro to build this macro based on its original version. `PAGING_FPN = GETVAL(pte, PAGING_PTR_FPN_MASK, 0)` is its definition, while `PAGING_PTE_FPN_MASK` is a binary integer 000...011...11 (composed of 13 low bits 1 and 29 high bits 0). Without changing any bits, we execute a bitwise-and operation between the `pte` and `PAGING_PTR_FPN_MASK` in order to retrieve the frame number `(FPN)`, which is equivalent to the 13 low-bits.



- `PAGING_PAGE_PRESENT(pte)`: This macro conducts a bitwise-and operation between `pte` and `PAGING_PTE_PRESENT_MASK`. The binary value 10...00, with the first high bit being 1 and the next 31 low bits being 0, represents the `PAGING_PTE_PRESENT_MASK`. The received value will be 0 if there isn't a page table entry (`pte`) in RAM, and 1 otherwise.

- `PAGING_SWP(pte)`: in the source code, this macro is likely to be incorrect. As a result, we used the `GETVAL` macro to construct this macro based on its original design. `GETVAL(pte, PAGING_PTE_SWPOFF_MASK, PAGING_SWPFPN_OFFSET)` is its definition. `PAGING_PTE_SWPOFF_MASK` is represented by the binary number 11..110000, which consists of 21 high-bits (1 and 5 low-bits 0), while `PAGING_SWPFPN_OFFSET` is represented by the decimal value 5. We do a bitwise-and operation between `pte` and `PAGING_PTE_SWPOFF_MASK`, then shift 5 bits to the right, to extract `SWP` Offset, which translates to 21 high-bits.

4.2.2 TLB (Translation Lookaside buffer)

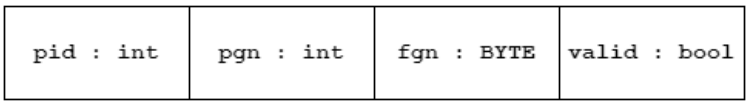
TLB (Translation Lookaside Buffer) is a memory cache that stores the recent translations of virtual memory to physical memory. It is used to reduce the time accessing to the physical memory. In our assignment, TLB is belong to `struct memphy_struct`, since it is proposed to utilize the power of hardware to leverage its caches capabilities.

4.2.2.a Structure of TLB

Since the initial code of `memphy_struct` doesn't support the mechanism of caches, we add new attributes into this struct in `os-mm.h`:

```
typedef struct {
   int pid;
   int pgn;
   BYTE fpn;
   bool valid;
} tlb_entry;

struct memphy_struct {
   ...
#ifdef CPU_TLB
   tlb_entry * tlb_table;
   int tlb_count;
#endif
   pthread_mutex_t mtx;
};
```



Since the TLB is used among the processes, there's a `pid` factor to determine which process own that entry. When CPU generates the virtual address, `tlb_table` will searched through its entries to find the `pgn` associated with that address and find the extract `fgn`, thereby generating physical address quickly without the need of accessing PTE in `mm_struct`. The `valid` factor will be False in case the entry is deleted from the table, otherwise it will always be true.

4.2.2.b Support functions

In the assignment, there are two functions support the direct mapping mechanism for the TLB:

1. `int tlb_cache_read(struct memphy_struct * mp, int pid, int pgnum, BYTE * value)`
   This function is used for finding the entry have the `pid` and `pgnum`, in order to read the `fpn` value stored in that entry and assign it to the `value` variable. The implementation for this function:

```
int tlb_cache_read(struct memphy_struct * mp, int pid, int pgnum, BYTE * value)
{
   /* TODO: the identify info is mapped to
    *       cache line by employing:
    *       direct mapped, associated mapping etc.
    */

   if (mp == NULL || mp->tlb_count == 0)
      return -1;
```

```
10
11      int i;
12      for (i = 0; i < mp->tlb_count; i++) {
13          if (mp->tlb_table[i].pid == pid && mp->tlb_table[i].pgn == pgnum) {
14              *value = mp->tlb_table[i].fpn;
15              return 0;
16          }
17      }
18
19      return -1;
20  }
```

2. `int tlb_cache_write(struct memphy_struct * mp, int pid, int pgnum, BYTE value)`
   This function is used for finding the entry have the `pid` and `pgnum`, then assigned the `value` into the `fgn` variable of that entry. There would be three cases for this function:

   (a) Cannot found the entry associated with, `pid` and `pgnum` and have `valid = True`, we add new entry into this table.

   (b) We found the entry associated with `pid` and `pgnum`, then update `value` into `fgn`.

   (c) Found the free entry (`valid = False`) then overwrite the value into this entry.

   The implementation for this function:

```
1   int tlb_cache_write(struct memphy_struct *mp, int pid, int pgnum, BYTE value)
2   {
3       /* TODO: the identify info is mapped to
4        *       cache line by employing:
5        *       direct mapped, associated mapping etc.
6        */
7       if (mp == NULL)
8           return -1;
9
10      int i, free_idx = -1;
11      for(i = 0; i < mp->tlb_count; i++) {
12          if (mp->tlb_table[i].pid == pid && mp->tlb_table[i].pgn == pgnum && mp->
        tlb_table[i].valid) {
13              mp->tlb_table[i].fpn = value;
14              return 0;
15          } else if (!mp->tlb_table[i].valid) {
16              free_idx = i;
17          }
18      }
19
20      if (free_idx != -1) {
21          mp->tlb_table[free_idx].pid = pid;
22          mp->tlb_table[free_idx].pgn = pgnum;
23          mp->tlb_table[free_idx].fpn = value;
24          mp->tlb_table[free_idx].valid = true;
25      } else {
26          mp->tlb_table[mp->tlb_count].pid = pid;
27          mp->tlb_table[mp->tlb_count].pgn = pgnum;
28          mp->tlb_table[mp->tlb_count].fpn = value;
29          mp->tlb_table[mp->tlb_count].valid = true;
30          mp->tlb_count++;
31      }
32      return 0;
33  }
```

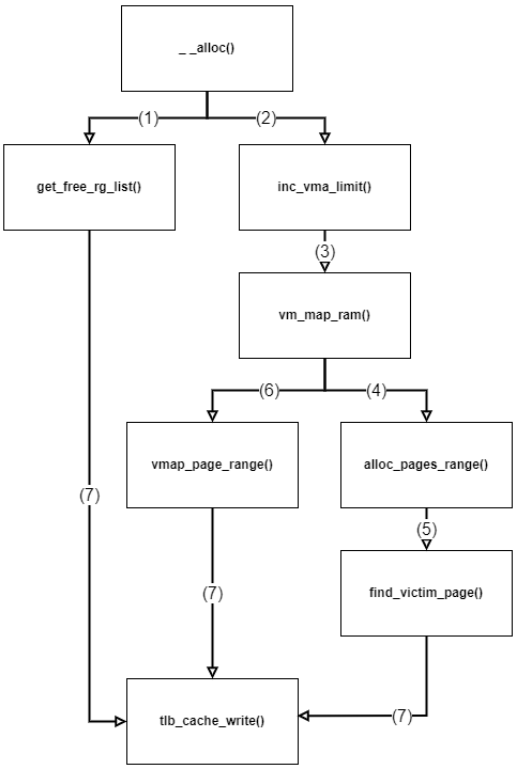
4.2.3 Code Breakdown

4.2.3.a List of implemented Functions:

- In **mm.c** source file (related to mapping pages to frames):
  - `vmap_page_range()`: To map a range of pages at an aligned address-list
  - `alloc_pages_range()`: To allocate the required number of frames in RAM
- In **mm-vm.c** source file (related to virtual memory):
  - `__free()`: To remove a region from virtual memory
  - `pg_getpage()`: To retrieve a page from RAM
  - `find_victim_page()`: To find a victim page using the FIFO algorithm.
- In **mm-memphy.c** source file (related to physical RAM):
  - `MEMPHY_dump()`: To dump the information about RAM.
- In **cpu-tlb.c** source file (related to utilize TLB):
  - `tlballoc()` : To alloc the memory region like `__alloc()`, combining with TLB cache mechanisms.
  - `tlbfree_data()`: To free the memory region like __free(), combining with TLB cache mechanism.
  - `tlb_free_entry_of()`: To set the `valid` factor of an entry in TLB table to `false`, indicating that the entry is deleted, letting future `tlb_cache_write()` can utilize this entry.
  - `tlbread()`: To read the memory region like `__read()`, combining with TLB cache mechanism.
  - `tlbwrite()`: To write to the memory region like `__write()`, combining with TLB cache mechanism.

4.2.3.b Relationship between functions:

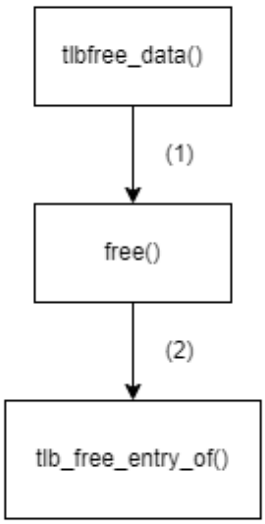
- **Alloc**



(1): To allocate a memory unit, first finding free region that can fit the allocated ones in `vm_free_rglist` by `get_free_rg_list()`.
(2): If can not found suitable free region to allocate, the virtual machine limit is increased.

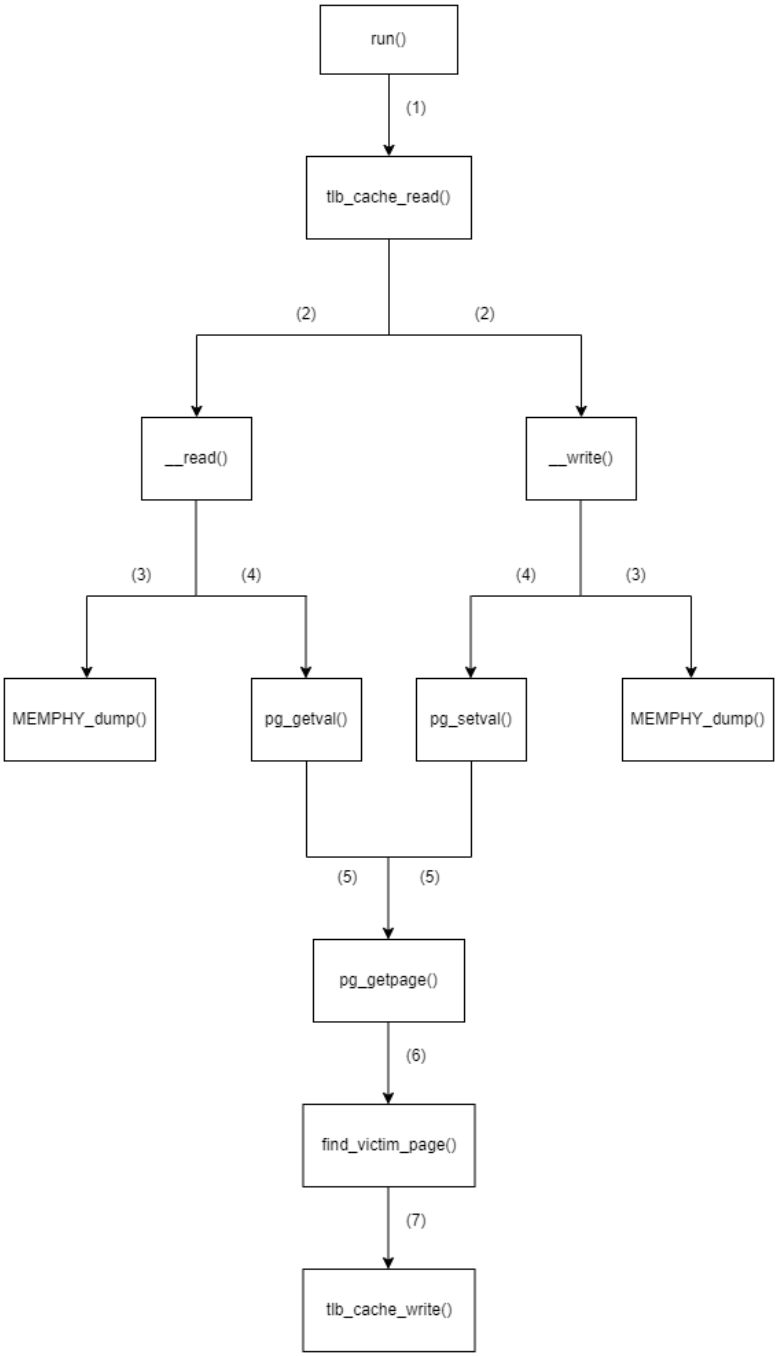
(3): If it is permissible to allocate a new virtual memory area, it needs to be mapped in RAM.
(4): Finding free frames in RAM to use for allocation process.
(5): If no free frames are available, the victim page is selected from the FIFO list of the process.
(6): The victim pages are chosen using FIFO algorithm.
(7): Whenever the status of the allocated pages is updated, if `#define CPU_TLB`, page num and frame num got from the allocated address will be added into the entries of TLB table.

- **Free**



(1): First, call `__free()` to free the target region.
(2): In order to free the unnecessary page in TLB after `__free()`, call `tlb_free_entry_of()` on target pages.

- **Read and Write**



(1): When running, the program will read the input and pass it in run to select the appropriate function to carry out the input. Then `tlb_cache_read()` will be call to retrieve the frame num.
(2): If not retrieved successfully (TLB miss), the program will carry out usual paging in RAM.
(3): When page read or page write, the source code prints out the RAM status of the previous stage, which helps us to determine the values in RAM
(4): For reading, we use the getting value function, while, for writing, we use the setting value function
(5): If this page is currently located in RAM, we return the frame number in RAM. In the other situation, not currently located in RAM, we need to swap from SWAP to RAM with the condition that

there is a free frame in RAM
(6): If not, we need to find the victim page and swap them out to get the free frame
(7): Once the required number of pages is made available, they are mapped in RAM, and the status of the allocated pages is updated. Then we will update the newest pgn and fgn to TLB through `tlb_cache_write()`

4.3 Result

a. Output

Input process: p0s file

```
1 10
calc
alloc 300 0
alloc 300 4
free 0
alloc 100 1
write 100 1 20
read 1 20 20
write 103 3 20
read 3 20 20
free 4
```

Input Testcase: test_file

```
2 1 1
4096 16777216 0 0 0
1 p0s  130
```

Code Output: in terminal

```
input/test_file
Time slot   0
ld_routine
Time slot   1
        Loaded a process at input/proc/p0s, PID: 1 PRIO: 130
        CPU 0: Dispatched process  1
Time slot   2

__alloc 300 BYTE for region 0

__alloc address: 0
Time slot   3
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1

__alloc 300 BYTE for region 4

__alloc address: 300
Time slot   4
Free region 0
Time slot   5
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1

__alloc 100 BYTE for region 1
Time slot   6
TLB hit at write region=1 offset=20 value=100
Ram content: Nothing in ram
Time slot   7
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
TLB hit at read region=1 offset=20
Ram content: 100
Time slot   8
Failed to write since this region has not been allocated yet!
Time slot   9
        CPU 0: Put process  1 to run queue
```

```
        CPU 0: Dispatched process  1
Failed to read since this region has not been allocated yet!
Time slot   10
Free region 4
Time slot   11
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

b. Explanation:

Command	Explanation
calc	Do dummy things
alloc 300 0	Allocate 300 bytes(1.17 pages), so we will allocate 2 pages for region 0. ⇒ The size of all allocated pages = 0 + 2 x 256 = 512 TLB will add the entry of page 0 since `PAGING_PGN(0) = 0`.
alloc 300 4	Allocate 300 bytes(1.17 pages), so we will allocate 2 pages for region 4. ⇒ The size of all allocated pages = 512 + 2 x 256 = 1024 TLB will add the entry of page 1 since `PAGING_PGN(300) = 1`
free 0	Enlist region 0 to free area. In TLB, entry of page 0 will be set invalid.
alloc 100 1	Allocate 100 bytes(0.39 pages), so we will allocate 1 page for region 1 . (get from free region list) ⇒ The size of all allocated pages = 1024 + 1 x 256 = 1280 TLB will add the entry of page 0 since `PAGING_PGN(0) = 0`.
write 100 1 20	Write value 100 to region 1 at offset 20. The page accessed is `PAGING_PGN(20) = 0`, so TLB will hit. After that, the code prints out the RAM status before writing the value (100). Therefore, there is no value in RAM because we have not written anything before.
read 1 20 20	Read region 1 at offset 20. The page accessed is `PAGING_PGN(20) = 0`, so TLB will hit. As a result, we get the previously written value which is 100. After reading, the code prints out the RAM status before reading, and we can alse see the value 100 in RAM
write 103 3 20	Write value 103 to region 3 at offset 20. However, this region has not been allocated before. Therefore, the code prints out the notification to show that it is failed to execute the command.
read 3 20 20	Read region 3 at offset 20. However, this region has not been allocated before. Therefore, the code prints out the notification to show that it is failed to execute the command.
free 4	Enlist region 0 to free area

5 Put it all together

Answering Questions

Question: What will happen if the synchronization is not handled in your simple OS? Illustrate the problem of your simple OS by example if you have any.

If the synchronization is not handled carefully in my simple OS, that will lead to some potential consequences:

- **Race Condition**: Since synchronization is not handled, multiple processors or threads may accessed a resource at the same time, leading to race conditions and may cause conflicts. In our simple OS, the shared resources are `mram`, `mswp` and `tlb`.
- **Deadlock**: When multiple processes or threads are waiting for each other to release resources, it can lead to a situation where all of them are stuck waiting, result in a deadlock.
- **Performance delegation**: The overhead and contention for shared resources will increase, i.e. degradation in performance, if the synchronization is not executed efficiently.
- **Inconsistent data**: When a shared data is not secured by synchronization mechanisms like semaphores or locks, it would result in making some processes or threads get incorrect data.
- **Starvation**: The starvation occurs when a process or thread acquires resources without allowing others to use them. As a result, other processes or threads cannot make progress.

Here is a situation that we do not handle synchronization in scheduling:

```
Time slot    2
        CPU 4: Dispatched process  2
        CPU 3: Dispatched process  2
        Loaded a process at input/proc/s3, PID: 1 PRIO: 130
```

In the above figure, we don't handled synchronization for enqueue and dequeue, which leads to the race condition above, where both CPU run on the same process.

6 Conclusion

In conclusion, the development and exploration of a Simple Operating System project have provided invaluable insights into the fundamental principles of operating systems design and implementation. Through this endeavor, we have gained a deeper understanding of crucial concepts such as process management, memory allocation, file systems, and inter-process communication. This project has not only sharpened our technical skills but has also fostered a spirit of collaboration and problem-solving within our team. We have encountered numerous challenges along the way, each presenting an opportunity for growth and learning. Moving forward, the lessons learned from this project will undoubtedly serve as a solid foundation for future endeavors in operating systems development. Whether pursuing further research in the field or applying our newfound knowledge in practical settings, the experience gained from building a Simple Operating System will continue to shape and inspire our journey in the world of technology.

7 References

References

[1] GeeksforGeeks. February 17, 2023. *Paged Segmentation and Segmented Paging*. Links: https://www.geeksforgeeks.org/paged-segmentation-and-segmented-paging/