

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



OPERATING SYSTEM LAB (CO2018)

ASSIGNMENT

Simple Operating System

Advisor: GV Nguyễn Mạnh Thìn
Students: Lê Ngọc Vinh - 2213964
Nguyễn Ngọc Duy - 2210522
Đỗ Hoàng Quân - 2212779

Class: TN02

HO CHI MINH CITY, MAY 2024



Contents

| | | |
|----------|--|-----------|
| 1 | Introduction of the Assignment | 4 |
| 1.1 | An overview | 4 |
| 2 | Scheduler | 5 |
| 2.1 | Introduction | 5 |
| 2.1.1 | Theory of Scheduling | 5 |
| 2.1.2 | Description of Multilevel Queue Policy | 5 |
| 2.1.3 | Requirements of assignment | 7 |
| 2.2 | Implementation | 7 |
| 2.3 | Result | 12 |
| 2.3.1 | Input | 12 |
| 2.3.2 | Output | 13 |
| 2.3.3 | Gantt Diagram | 13 |
| 2.4 | Question | 14 |
| 3 | Paging-based Memory Management | 16 |
| 3.1 | Introduction | 16 |
| 3.1.1 | Virtual memory mapping in each process | 16 |
| 3.1.2 | Physical Memory | 17 |
| 3.1.3 | Paging-based address translation scheme | 18 |
| 3.2 | Implementation | 19 |
| 3.2.1 | mm-vm.c | 19 |
| 3.2.2 | mm-memphy.c | 28 |
| 3.2.3 | mm.c | 29 |
| 3.3 | Result | 30 |
| 3.3.1 | Input | 30 |
| 3.3.2 | Output | 31 |
| 3.4 | Status of the mapped page and the index page involving TLB | 32 |
| 3.5 | Questions | 32 |
| 4 | TLB Memory Management | 34 |
| 4.1 | Introduction of TLB | 34 |
| 4.2 | TLB Operations in the Assignment | 34 |
| 4.3 | Implementation | 37 |
| 4.3.1 | TLB operations | 37 |
| 4.3.2 | TLB Cache operations | 44 |



| | | |
|----------|---|-----------|
| 4.4 | Result | 52 |
| 4.4.1 | Input | 52 |
| 4.4.2 | Output of Direct Mapped TLB | 53 |
| 4.4.3 | Output of Fully Associative TLB | 55 |
| 4.5 | Question | 57 |
| 5 | Synchronization | 60 |
| 5.1 | Overview | 60 |
| 5.2 | Implementation | 60 |
| 5.3 | Question | 62 |

1 Introduction of the Assignment

1.1 An overview

The assignment is about simulating a simple operating system to help student understand the fundamental concepts of scheduling, synchronization and memory management. Figure 1 shows the overall architecture of the *operating system* we are going to implement. Generally, the OS has to manage two *virtual* resources: CPU(s) and RAM using two core components:

- Scheduler (and Dispatcher): determines which process is allowed to run on which CPU.
- Virtual memory engine (VME): isolates the memory space of each process from other. The physical RAM is shared by multiple processes but each process do not know the existence of other. This is done by letting each process has its own virtual memory space and the Virtual memory engine will map and translate the virtual addresses provided by processes to corresponding physical addresses.

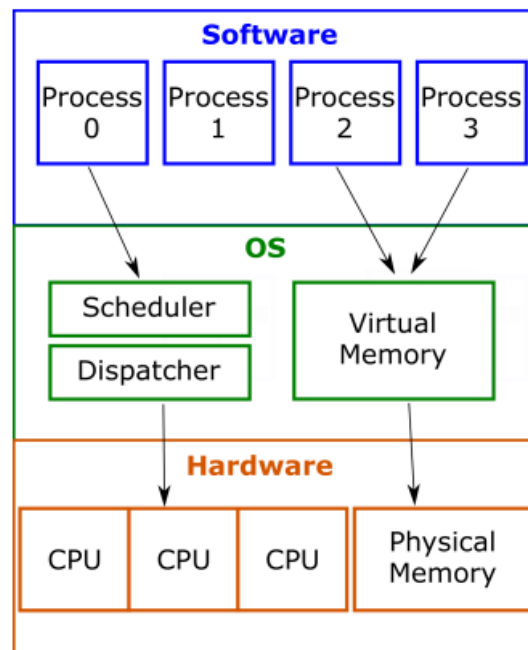


Figure 1: The general view of key modules in this assignment

2 Scheduler

2.1 Introduction

2.1.1 Theory of Scheduling

Let's dive into a background where the development and changed in technology lead to the condition that increased the number of processes. The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. Therefore put more pressure on developer to improve by using resources utilization in general. In the heart of a multitasking operating system lies process scheduling, the mastermind behind efficiently dividing the CPU's time among competing programs. This crucial role involves striking a balance between various performance metrics, such as how much work gets done, how long processes wait, and how quickly they react. To achieve this, operating systems utilize a diverse toolbox of scheduling algorithms. Each algorithm has its own approach, from prioritizing tasks that finish quickly to ensuring all processes get a fair shot. By carefully selecting the right algorithm for the workload, process scheduling significantly boosts system performance, keeps resources well-utilized, and ultimately, keeps users happy.

Many contemporary computer systems support multiple processors and allow each processor to schedule itself independently. Typically, each processor maintains its own private queue of processes (or threads), all of which are available to run. Additional issues related to multi-processor scheduling include processor affinity, load balancing, and multicore processing.

2.1.2 Description of Multilevel Queue Policy

The scheduler works as given figure. For each new program, the loader will create a new process and assign a new pcb to it. The loader then reads and copies the content of the program to the text segment of the new process. The PCB of the process is pushed to the associated ready queue having the same priority with the value prio of this process. Then, it waits for the CPU. The CPU runs processes in round-robin style. Each process is allowed to run in time slice. After that, the CPU is forced to enqueue the process back to it associated priority ready queue. The CPU then picks up another process from ready queue and continue running.

In this system, we implement the Multi-Level Queue (MLQ) policy. The system contains MAX PRIO priority levels. Each priority is held by one ready queue. We simplify the add queue and put proc as putting the process proc to appropriated ready queue by priority matching. The main design is belong to the MLQ policy deployed by get proc to fetch a proc and then dispatch CPU.

The description of MLQ policy: the traversed step of eady queue list is a fixed formulated number based on the priority, i.e. $\text{slot} = (\text{MAX PRIO} - \text{prio})$, each queue have only fixed slot to

use the CPU and when it is used up, the system must change the resource to the other process in the next queue and left the remaining work for future slot eventhough it needs a completed round of ready queue.

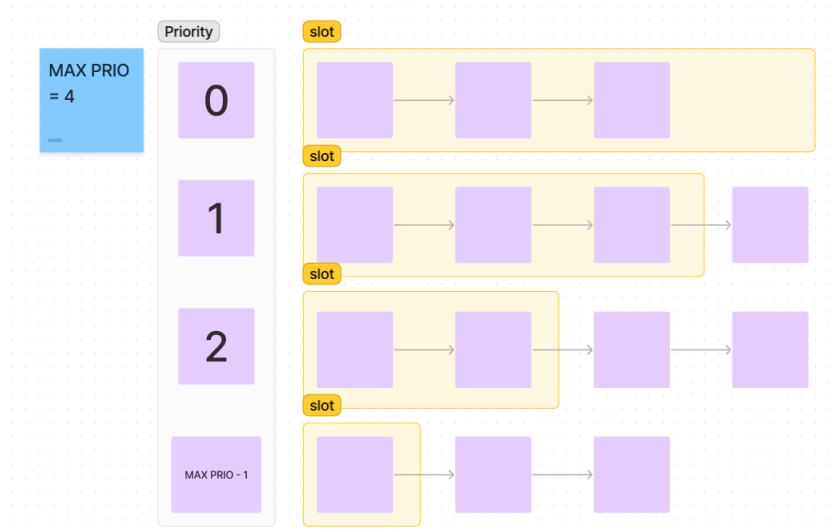


Figure 2: Multi-level Queue example

Here we represent the MLQ of my scheduling as the upper figure. The slot for each queue with match priority will reduced by 1 whenever the priority increase. For instance with the queue with priority[0] is the number of process if atomically execute would finish there job, the range of usable slot after that would be smaller. Another the situation is the number of process exceed the number of limit slots, the left over outside the queue would not be picked for running. Additionally, if the process is not finish right after being picked up, would be placed back at the end of matching priority queue, and do the previous step with reduced number of slot left.

* **Caution:** The meaning of slot we currently using is the quantity of remaining process that can be used not the left-over time for the corresponding queue_t. Due to the reason that multi level queue treated many queue based on there priority, therefore if we use number of time per queue would defeated the purpose of MLQ.

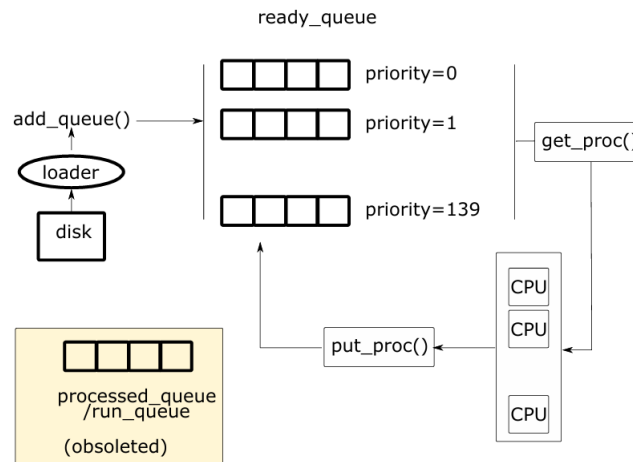


Figure 3: Multi-level Queue Policy

2.1.3 Requirements of assignment

MLQ policy only goes through the fixed step to traverse all the queue in the priority ready queue list. Your job in this part is to implement this algorithm by completing the following functions:

- `enqueue()` and `dequeue()` (in `queue.c`): We have defined a struct (`queue_t`) for a priority queue at `queue.h`. Your task is to implement those functions to put a new PCB to the queue and get the next 'in turn' PCB out of the queue.
- `get_mlq_proc()` (in `sched.c`): gets PCB of a process waiting from the ready queue system. The selected ready queue 'in turn' has been described in the above policy.

2.2 Implementation

1. `queue.h`: Changing the properties of struct `queue_t`

```
1 struct queue_t {
2     struct pcb_t * proc[MAX_QUEUE_SIZE];
3     int size;
4     int slot;
5 };
```

We add the attribute `slot` variable type `int`, which store the number of process left that can used of corresponding `queue_t`.

2. `queue.c`: Implementing the `enqueue()` and `dequeue()`

With basic queue, when you **Enqueue (Insert)** an element into a queue, that means you adds an element to the rear of the queue. In this situation, `enqueue()` method receive a `queue_t` and `pcb_t`. representing the process to be added. We just simply add it in the end of the given queue and increase the size of that queue by 1. Also each queue come up with a fixed slots of `pcb_t`, therefore we have to precheck whether it's enough space to add `pcb_t` corresponding with that process in.

```
1 void enqueue(struct queue_t *q, struct pcb_t *proc)
2 {
3     /* TODO: put a new process to queue [q] */
4     /*
5      * Increase the number of Processe(s) in queue by 1
6      * Put in the queue the new process
7      */
8     if (q->size < MAX_QUEUE_SIZE)
9     {
10         q->proc[q->size] = proc;
11         q->size++;
12     }
13     else
14     {
15         printf("Try to add a proces to a full queue_t\n");
16         exit(1);
17     }
18 }
```

As in the basic concept of queue, our approach to this function is:

- Precheck if the given queue is empty or not for next step.
 - After that, take the first `pcb_t` (meaning the very first process has come “in turn”).
 - Move and copy the remaining in not chosen `pcb_t` to the front.
 - Completely delete data and changed in number of `pcb_t` (size) and slot use left.
-

```
1 struct pcb_t *dequeue(struct queue_t *q)
2 {
3     /* TODO: return a pcb whose priority is the highest
4      * in the queue [q] and remember to remove it from q
5      * */
6
7     // Can only process if only it is not empty
```

```
8         if (empty(q))
9             return NULL;
10
11         // res hold the pcb with the highest priority
12         struct pcb_t *res = q->proc[0];
13
14         /* Remove the process from the queue
15          * shift the right to left 1 time by removing a pcb took place */
16         for (int i = 0; i < q->size - 1; i++)
17         {
18             q->proc[i] = q->proc[i + 1];
19         }
20
21         // Set NULL to last element
22         q->proc[q->size - 1] = NULL;
23
24         // Size change by 1 cause we take 1 pcb
25         q->size--;
26
27         // Decrease the number of proces in this queue type by one,
28         // the usable slot after this call is slot--
29         q->slot--;
30
31         return (res);
32     }
```

3. sched.c: Implement the get_mlq_proc() and get_proc()

As the assignment told us that this simulation os run on dual mechanism, below code we show would represent single and multi level queue setting. The following show the get_proc() from single mechanism.

```
1 struct pcb_t *get_proc(void)
2 {
3     struct pcb_t *proc = NULL;
4     /*TODO: get a process from [ready_queue].
5      * Remember to use lock to protect the queue.
6      * */
7
8     pthread_mutex_lock(&queue_lock);
```

```
9   proc = dequeue(&ready_queue);  
10  pthread_mutex_unlock(&queue_lock);  
11  
12  return proc;  
13 }
```

The operating system run and store all the on-going process in one single queue. Therefore, this function would take the very first process control block of the queue and run by round robin method by the CPU. Another notice of this decision of First Come First Serve scheduling would solved the starvation problem of `low_prio` process (with high number `prio`). The approach of round robin just for the `high_prio` process would raise the situation that `low_prio` process would never be executed (starvation), therefore we would not choose this technique to solve this problem.

To move on the function implementing without single queue, we shift to multi level queue policy would be shown by:

- Loop through all `queue_t` and find any `pcb_t` (process) that satisfy the condition that the queue has remaining available slot for that queue to run (Proportion to the priority, which `low_prio` has more attempt to be picked up and vice versa to the `high_prio`) and match the queue priority.
- Here we use the trick of C, which if the iterator reach out the end of loop, without the return (interrupt by “in turn” process found), the iterator would be equal to the `MAX_PRIO`, which also mean no process that valid the condition above. After that we perform the reset in slot attribute for all `queue_t` in the `mlq` and perform the search for a process needed.
- If both step above failed in finding the `pcb_t`, `NULL` would be assign to the place of returned process control block.

```
1  struct pcb_t *get_mlq_proc(void)  
2  {  
3      struct pcb_t *proc = NULL;  
4      /*TODO: get a process from PRIORITY [ready_queue].  
5       * Remember to use lock to protect the queue.  
6       */  
7  
8      // New 2 mutex lock and unlock  
9      pthread_mutex_lock(&queue_lock);  
10     int cur_prio;
```

```
11  for (cur_prio = 0; cur_prio < MAX_PRIO; ++cur_prio)
12  {
13      if (!empty(&mlq_ready_queue[cur_prio]) && mlq_ready_queue[cur_prio].slot
14          > 0)
15      {
16          proc = dequeue(&mlq_ready_queue[cur_prio]);
17          break;
18      }
19  }
20  // This might run into a problem, thus the above code --slot but
21  // There might be a process somewhere in the mlq but we have use all slot
22  // Due to that reason, we need to "refill" slot for all queues
23  // This section only occurs few time, after CPU use once in << <slot(slot -
24      1)/2> times
25  if (cur_prio == MAX_PRIO)
26  {
27      // The mlq has run a cycle (all queue for its related slot)
28      // Therefore, set the slot for all queues again
29      for (int i = 0; i < MAX_PRIO; i++)
30      {
31          mlq_ready_queue[i].slot = MAX_PRIO - i;
32      }
33
34      // Take out the process
35      for (cur_prio = 0; cur_prio < MAX_PRIO; ++cur_prio)
36      {
37          // in this loop we noo need to check the slot, cause we just pre
38              giveslot for all queue
39          // This is only true in this part cause the number of slot run from
40              [1 -> MAX_PRIO - 1]
41          // but for clear code, we would also check it
42          if (!empty(&mlq_ready_queue[cur_prio]) &&
43              mlq_ready_queue[cur_prio].slot > 0)
44          {
45              proc = dequeue(&mlq_ready_queue[cur_prio]);
46              break;
47          }
48      }
49  }
50  pthread_mutex_unlock(&queue_lock);
```

```
46     return proc;
47 }
```

It's worth to remind that we would add mutex lock for this part, due to the number of CPU can work at a same time represented as multi thread call function `cpu_routine()`, those CPU in the simulation work independently at needed a process to work on and all these call the `get_proc()` function. The process "in turn" should be given by the operating system should be concurrently and correct. Without the protection, the taken process would come up with unexpected behaviour.

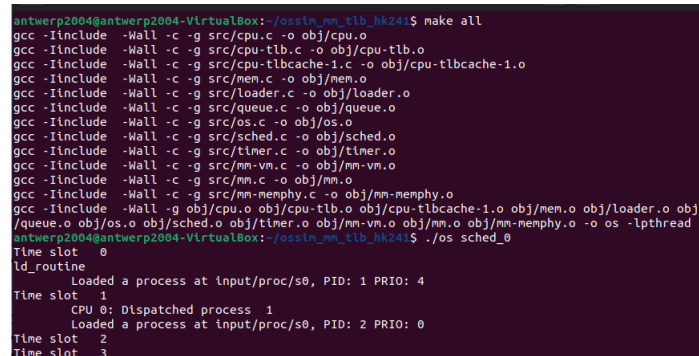
2.3 Result

2.3.1 Input

Input file: sched_0.txt

```
1 2 1 2
2 0 s0 0
3 4 s1 0
4
5 // Below this line is not the input default
6 // This is just the descriptive for each process above
7 s0 = 15
8 s1 = 7
```

To compile, run `make all`, then run `./os sched_0` in terminal.



```
antwerp2004@antwerp2004-VirtualBox:~/os$ make all
gcc -Iinclude -Wall -c -g src/cpu.c -o obj/cpu.o
gcc -Iinclude -Wall -c -g src/cpu-tlb.c -o obj/cpu-tlb.o
gcc -Iinclude -Wall -c -g src/cpu-tlb-cache-1.c -o obj/cpu-tlb-cache-1.o
gcc -Iinclude -Wall -c -g src/mem.c -o obj/mem.o
gcc -Iinclude -Wall -c -g src/loader.c -o obj/loader.o
gcc -Iinclude -Wall -c -g src/queue.c -o obj/queue.o
gcc -Iinclude -Wall -c -g src/os.c -o obj/os.o
gcc -Iinclude -Wall -c -g src/sched.c -o obj/sched.o
gcc -Iinclude -Wall -c -g src/timer.c -o obj/timer.o
gcc -Iinclude -Wall -c -g src/mm-vn.c -o obj/mm-vn.o
gcc -Iinclude -Wall -c -g src/mm.c -o obj/mm.o
gcc -Iinclude -Wall -c -g src/mm-memphy.c -o obj/mm-memphy.o
gcc -Iinclude -Wall -g obj/cpu.o obj/cpu-tlb.o obj/cpu-tlb-cache-1.o obj/mem.o obj/loader.o obj/queue.o obj/os.o obj/sched.o obj/timer.o obj/mm-vn.o obj/mm-memphy.o -o os -lpthread
antwerp2004@antwerp2004-VirtualBox:~/os$ ./os sched_0
Time slot 0
ld_routine
    Loaded a process at input/proc/s0, PID: 1 PRI0: 4
Time slot 1
    CPU 0: Dispatched process 1
    Loaded a process at input/proc/s0, PID: 2 PRI0: 0
Time slot 2
Time slot 3
```

2.3.2 Output

```
antwerp2004@antwerp2004-VirtualBox: ~/OS-BTL-main/ossh_nmlb_hk241$ ./os sched_0
Time slot 0
ld_routine
Loaded a process at input/proc/s0, PID: 1 PRI0: 0
Time slot 1
CPU 0: Dispatched process 1
Time slot 2
CPU 0: Put process 1 to run queue
CPU 0: Dispatched process 1
Time slot 3
Loaded a process at input/proc/s1, PID: 2 PRI0: 0
Time slot 4
CPU 0: Put process 1 to run queue
CPU 0: Dispatched process 2
Time slot 5
CPU 0: Put process 2 to run queue
CPU 0: Dispatched process 1
Time slot 6
CPU 0: Put process 1 to run queue
CPU 0: Dispatched process 2
Time slot 7
CPU 0: Put process 2 to run queue
CPU 0: Dispatched process 1
Time slot 8
CPU 0: Put process 1 to run queue
CPU 0: Dispatched process 2
Time slot 9
CPU 0: Put process 2 to run queue
CPU 0: Dispatched process 1
Time slot 10
CPU 0: Put process 1 to run queue
CPU 0: Dispatched process 2
Time slot 11
CPU 0: Put process 2 to run queue
CPU 0: Dispatched process 1
Time slot 12
CPU 0: Put process 1 to run queue
CPU 0: Dispatched process 2
Time slot 13
CPU 0: Put process 2 to run queue
CPU 0: Dispatched process 1
Time slot 14
```

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

Figure 4: Output of `sched_0`

2.3.3 Gantt Diagram

With the meaning of time slice: 2 (unit of time, from now we assume 1 unit of time is 1 second). The second argument is 1 shows the number of CPU in this simulation, in this specific case, we have 1 CPU run at a time. The third argument of this input is number of process to be run which is 2 process below named: s0 and s1 which respectively has 15 and 7 instructions CALC in each program (save in text file).. 0 and 4 is the arrival time of that process, with both processes have the live priority when using MLQ_Sched is 0. Due to the unit test sched, there is no input for `tlbsz` and `paging sz`.

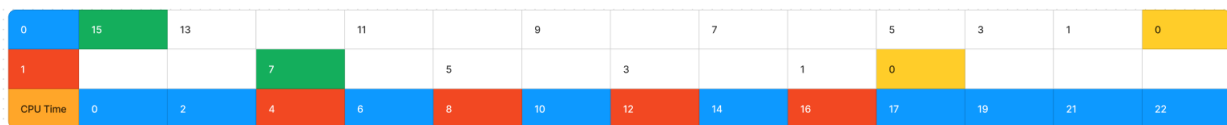


Figure 5: Gantt diagram of `sched_0` Output

The last line show the gantt chart for 1 cpu time with the red block is process named s1, and blue block box shows the time that process named s2 in the cpu. 2 head lines show the process if it represent in CPU. Green box show that process frist time loaded and run, meanwhile yellow box provide us with the information when it's finish. One notice that every counting time would normally increase by 2 due to the time slice, but if it is 1 (in this test case) which means a process has done there job, the cpu would get another process and not let 1 unit of time free due to nothing.

2.4 Question

Question 1: 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 used in this assignment has some advantages compared to other scheduling algorithms:

- **Assignment based on priority:** Multi-level queue scheduling allows tasks to be assigned priorities based on their characteristics or requirements. This means that critical or time-sensitive tasks can be given higher priority, ensuring they are processed promptly. In contrast, some other algorithms like Round Robin or First Come First Served may not inherently prioritize tasks based on their importance or urgency.
- **Better Resource Allocation:** In a multi level queue system, tasks are typically segregated based on their priority or type. This segregation enables more efficient resource allocation since tasks with higher priorities can be allocated more resources or processed with shorter waiting times. This is advantageous compared to algorithms like Round Robin, where all tasks are treated equally in terms of resource allocation, regardless of their priority or urgency.
- **Enhanced Responsiveness:** Multi-level queue scheduling can lead to improved system responsiveness, especially for interactive applications or real-time tasks. By prioritizing certain types of tasks over others, such as interactive user processes over background tasks, the system can ensure that user interactions receive prompt responses, leading to a better user experience. This responsiveness might not be as consistent in algorithms like First-Come, First-Served, where tasks are processed strictly in the order they arrive.
- **Flexibility and Customization:** Multi-level queue scheduling offers flexibility in defining and managing different queues based on specific criteria, such as task priority, type, or resource requirements. This customization allows system administrators to tailor the scheduling policy to match the characteristics and demands of their particular workload or environment, which may not be as easily achievable with simpler scheduling algorithms.
- **Avoidance of Starvation:** Multi-level queue scheduling helps prevent starvation by ensuring that lower-priority tasks are not indefinitely delayed or ignored in favor of higher-priority tasks. By allowing tasks to move between queues based on certain conditions or criteria, such as waiting time or resource availability, the system can ensure that all tasks eventually receive attention, mitigating the risk of starvation.

In summary, the advantage of using multi-level queue scheduling lies in its ability to provide more fine-grained control over task prioritization, resource allocation, responsiveness, and



starvation avoidance compared to simpler scheduling algorithms. These advantages make it well-suited for environments where tasks have varying levels of importance or urgency, and where efficient resource utilization and system responsiveness are critical considerations.

3 Paging-based Memory Management

3.1 Introduction

3.1.1 Virtual memory mapping in each process

- Virtual memory space is structured as memory mappings for each process PCB. Within the virtual address space of a process, multiple memory areas are defined, such as code segments, stack segments, and heap segments.
- These memory areas are contiguous regions delineated by their starting and ending addresses, typically referred to as `vm_start` and `vm_end`, but the actual usable area is limited by the top pointing at `sbrk`, with regions captured by `struct vm_rg_struct` and free slots tracked by `vm_freerg_list`.

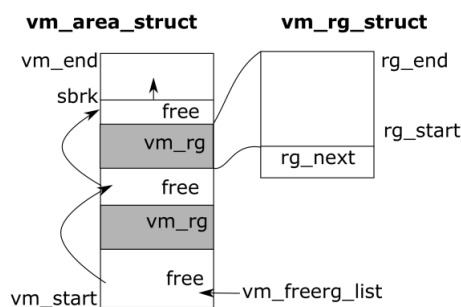


Figure 6: Structure of `vm_area` and `vm_region`.

```

1 //From include/os-mm.h
2 /*
3  * Memory region struct
4  */
5 struct vm_rg_struct {
6     unsigned long rg_start;
7     unsigned long rg_end;
8     struct vm_rg_struct *rg_next;
9 };
10 /*
11  * Memory area struct
12  */
13 struct vm_area_struct {
14     unsigned long vm_id;
15     unsigned long vm_start;
16     unsigned long vm_end;

```



```
17     unsigned long sbrk;
18     /*
19     * Derived field
20     * unsigned long vm_limit = vm_end - vm_start
21     */
22     struct mm_struct *vm_mm;
23     struct vm_rg_struct *vm_freerg_list;
24     struct vm_area_struct *vm_next;
25 };
```

- Memory regions within the virtual address space correspond to variables and data structures within the program's source code. To manage these regions, a symbol table structure (`struct vm_rg_struct symrgtbl`) is employed, which tracks the start and end points of each region. The symbol table serves as a reference for memory allocation and deallocation within the process's virtual memory space.
 - Each process's memory mapping is represented by a `struct mm_struct` data structure, which tracks memory regions. Additionally, the `pgd` field in the data structure represents the page table directory, which contains all the page table entries.
-

```
1 //From include/os-mm.h
2 /*
3 * Memory mapping struct
4 */
5 struct mm_struct {
6     uint32_t *pgd;
7     struct vm_area_struct *mmap;
8     /* Currently we support a fixed number of symbol */
9     struct vm_rg_struct symrgtbl[PAGING_MAX_SYMTBL_SZ];
10    struct pgn_t *fifo_pgn;
11 };
```

3.1.2 Physical Memory

- Physical memory hardware is installed at the system level, serving as the primary storage medium for all processes. The memory hardware includes RAM (Random Access Memory) and SWAP (Secondary Storage) devices. RAM and SWAP may utilize the same physical hardware but with different configurations and access mechanisms.
- RAM is directly accessible from the CPU's address bus, allowing for fast read/write

operations using CPU instructions. SWAP serves as additional storage, typically used when RAM is insufficient. Access to SWAP data requires moving it to RAM first.

- `struct framephy_struct` is used to store frame numbers, representing individual memory frames within RAM or SWAP.
- `struct memphy_struct` Contains basic fields such as storage size and access mode (random or sequential). It manages lists of free and used memory frames (`free_fp_list` and `used_fp_list`, respectively). The `rdmflg` field defines the memory access is randomly or serially access.

```
1 //From include/os-mm.h
2 /*
3  * FRAME/MEM PHY struct
4  */
5 struct framephy_struct {
6     int fpn;
7     struct framephy_struct *fp_next;
8 };
9 struct memphy_struct {
10     /* Basic field of data and size */
11     BYTE *storage;
12     int maxsz;
13     /* Sequential device fields */
14     int rdmflg;
15     int cursor;
16     /* Management structure */
17     struct framephy_struct *free_fp_list;
18     struct framephy_struct *used_fp_list;
19 };
```

3.1.3 Paging-based address translation scheme

- Supports segmentation and segmentation with paging, using a single-level paging system.
- Each process has its own page table to map virtual pages to physical frames.
- Page tables are updated to include mappings to RAM (MEMRAM) or SWAP (MEM-SWP).
- Basic memory operations like allocation (ALLOC), deallocation (FREE), read (READ), and write (WRITE) are discussed. Allocation seeks available memory regions and maps

them using page table entries. Deallocation releases memory space for reuse, and read/write operations require collaboration between modules for efficient page swapping.

- Swapping helps manage memory by moving contents between RAM and SWAP devices to free up space.

3.2 Implementation

3.2.1 mm-vm.c

1. enlist_vm_freerg_list

```
1  /*enlist_vm_freerg_list - add new rg to freerg_list
2  *@mm: memory region
3  *@rg_elmt: new region
4  *
5  */
6  // input the mm and region to be remove in mm
7  int enlist_vm_freerg_list(struct mm_struct *mm, struct vm_rg_struct rg_elmt)
8  {
9      struct vm_rg_struct *rg_node = mm->mmap->vm_freerg_list;
10
11      if (rg_elmt.rg_start >= rg_elmt.rg_end)
12          return -1;
13
14      // Reassign to head of freelist
15      rg_node = mm->mmap->vm_freerg_list;
16
17      // Init
18      if (rg_node == NULL)
19      {
20          struct vm_rg_struct *newnode = malloc(sizeof(struct vm_rg_struct));
21          newnode->rg_start = rg_elmt.rg_start;
22          newnode->rg_end = rg_elmt.rg_end;
23          newnode->rg_next = NULL;
24          mm->mmap->vm_freerg_list = newnode;
25          return 0;
26          // Init no need merge, since the number of free is 0 -> 1
27      }
28      else if (rg_elmt.rg_end <= rg_node->rg_start)
29      {
```

```
30     // Add far before first node
31     if (rg_elmt.rg_end == rg_node->rg_start)
32     {
33         rg_node->rg_start = rg_elmt.rg_start;
34         return 0;
35     }
36
37     struct vm_rg_struct *newnode = malloc(sizeof(struct vm_rg_struct));
38     newnode->rg_start = rg_elmt.rg_start;
39     newnode->rg_end = rg_elmt.rg_end;
40     newnode->rg_next = rg_node;
41     mm->mmap->vm_freerg_list = newnode;
42     return 0;
43     // No merge needed, thus exist a gap btw new node and first node
44 }
45
46 while (rg_node)
47 {
48     /* Special case when reach the end
49     Come up with 4 cases: (Btw 2 nodes)
50     * Perfect fit (Remove node require)
51     * node->end < elmt_start && elmt_end < next_node->start (between two
52       nodes) (new node require)
53     * node->end = elmt->start (to the left of current node)
54     * elmt->end = next_node->start (to the right of current node)
55     */
56     struct vm_rg_struct *next_node = rg_node->rg_next;
57     if (next_node == NULL)
58     {
59         if (rg_elmt.rg_start == rg_node->rg_end)
60         {
61             rg_node->rg_end = rg_elmt.rg_end;
62             return 0;
63         }
64
65         // Add far after last node
66         struct vm_rg_struct *newnode = malloc(sizeof(struct vm_rg_struct));
67         newnode->rg_start = rg_elmt.rg_start;
68         newnode->rg_end = rg_elmt.rg_end;
69         newnode->rg_next = NULL;
```

```
69         rg_node->rg_next = newnode;
70         return 0;
71         // No merge needed, thus exist a gap btw last node and new node
72     }
73     else if (rg_node->rg_end == rg_elmt.rg_start && rg_elmt.rg_end ==
74             next_node->rg_start)
75     {
76         // Special case that the used that fit exactly to the gap
77         // Therefore we need to merge the gap to 1 node
78
79         rg_node->rg_end = next_node->rg_end;
80         rg_node->rg_next = next_node->rg_next;
81
82         // Free the node
83         next_node->rg_end = next_node->rg_start = 0;
84         next_node->rg_next = NULL;
85         free(next_node);
86     }
87     else if (rg_node->rg_end < rg_elmt.rg_start && rg_elmt.rg_end <
88             next_node->rg_start)
89     {
90         struct vm_rg_struct *newnode = malloc(sizeof(struct vm_rg_struct));
91
92         // Copy data to new node and link to next node
93         newnode->rg_start = rg_elmt.rg_start;
94         newnode->rg_end = rg_elmt.rg_end;
95         newnode->rg_next = next_node;
96
97         // Add node btw gap
98         rg_node->rg_next = newnode;
99
100         return 0;
101     }
102     else if (rg_node->rg_end == rg_elmt.rg_start)
103     {
104         rg_node->rg_end = rg_elmt.rg_end; // Extend right
105         return 0;
106     }
107     else if (rg_elmt.rg_end == next_node->rg_start)
108     {
109         rg_elmt.rg_end = next_node->rg_end;
110         return 0;
111     }
112     }
```

```
107         next_node->rg_start = rg_elmt.rg_start; // Extend left
108         return 0;
109     }
110     rg_node = next_node;
111 }
112
113 fflush(stdout);
114
115 return 0;
116 }
```

This function is designed to manage a list of free memory regions within a virtual memory system.

2. __alloc

```
1 /*__alloc - allocate a region memory
2  *@caller: caller
3  *@vmaid: ID vm area to alloc memory region
4  *@rgid: memory region ID (used to identify variable in symbole table)
5  *@size: allocated size
6  *@alloc_addr: address of allocated memory region
7  *
8  */
9 int __alloc(struct pcb_t *caller, int vmaid, int rgid, int size, int
    *alloc_addr)
10 {
11     /*Allocate at the topproof */
12     struct vm_rg_struct rgnode;
13
14     // free has enough to put new size in
15     if (get_free_vmrg_area(caller, vmaid, size, &rgnode) == 0)
16     {
17         caller->mm->symrgtbl[rgid].rg_start = rgnode.rg_start;
18         caller->mm->symrgtbl[rgid].rg_end = rgnode.rg_end;
19
20         *alloc_addr = rgnode.rg_start;
21
22         return 0;
23     }
24     // here return -1 tell that in free not have enough space, require mmore
```

```
data in vm

25
26  /* TODO get_free_vmrg_area FAILED handle the region management (Fig.6)*/
27
28  /*Attempt to increate limit to get space */
29  struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);
30  int inc_sz = PAGING_PAGE_ALIGNSZ(size);
31  // int inc_limit_ret
32  int old_sbrk;
33
34  old_sbrk = cur_vma->sbrk;
35
36  /* TODO INCREASE THE LIMIT
37   * inc_vma_limit(caller, vmaid, inc_sz)
38   */
39  inc_vma_limit(caller, vmaid, inc_sz);
40
41  /*Successful increase limit */
42  caller->mm->symrgtbl[rgid].rg_start = old_sbrk;
43  caller->mm->symrgtbl[rgid].rg_end = old_sbrk + size;
44
45  *alloc_addr = old_sbrk;
46  cur_vma->sbrk = old_sbrk + size;
47
48  return 0;
49 }
```

The `__alloc` function is responsible for allocating memory within a specified virtual memory area. First, it checks for sufficient free space in the specified virtual memory area, then allocates memory if enough free space is available, updating the symbol table with the allocated region's start and end addresses. Finally, it stores the allocated memory's start address in the provided pointer variable.

3. `__free`

```
1  /*__free - remove a region memory
2   *@caller: caller
3   *@vmaid: ID vm area to alloc memory region
4   *@rgid: memory region ID (used to identify variable in symbole table)
5   *@size: allocated size
```

```
6  *
7  */
8  int __free(struct pcb_t *caller, int vmaid, int rgid)
9  {
10     struct vm_rg_struct *rgnode;
11
12     if (rgid < 0 || rgid > PAGING_MAX_SYMTBL_SZ)
13         return -1;
14
15     /* TODO: Manage the collect freed region to freerg_list */
16     rgnode = get_symrg_byid(caller->mm, rgid);
17
18     /*enlist the obsoleted memory region */
19     enlist_vm_freerg_list(caller->mm, *rgnode);
20
21     /*if(caller->mm->symrgtbl[rgid].rg_end != 0)
22         printf("Freed address %ld to %ld at register %d\n\n", rgnode->rg_start,
23             rgnode->rg_end, rgid);*/
23     caller->mm->symrgtbl[rgid].rg_end = caller->mm->symrgtbl[rgid].rg_start =
24         0;
25     return 0;
26 }
```

The `__free` function serves to release memory allocated within a specified virtual memory area. First, it locates the memory region associated with the given region ID in the symbol table of the process. Next, it enlists the freed memory region into the system's free memory region list. Then it resets the start and end addresses of the deallocated memory region in the symbol table to indicate that it's no longer in use.

4. `pg_getpage`

```
1  /*pg_getpage - get the page in ram
2   *@mm: memory region
3   *@pagenum: PGN
4   *@framenum: return FPN
5   *@caller: caller
6   *
7   */
8  int pg_getpage(struct mm_struct *mm, int pgn, int *fpgn, struct pcb_t *caller)
```



```
9 {
10     uint32_t pte = mm->pgd[pgn];
11
12     if (!PAGING_PAGE_PRESENT(pte))
13     { /* Page is not online, make it actively living */
14         int vicpgn, swpfpn;
15         int vicfpn;
16         uint32_t vicpte;
17
18         int tgtfpn = PAGING_SWP(pte); // the target frame storing our variable
19
20         /* TODO: Play with your paging theory here */
21         /* Find victim page */
22         if (find_victim_page(caller->mm, &vicpgn) < 0)
23             return -1;
24
25         vicpte = mm->pgd[vicpgn];
26         vicfpn = PAGING_SWP(vicpte);
27
28         /* Get free frame in MEMSWP */
29         if (MEMPHY_get_freefp(caller->active_mswp, &swpfpn) < 0)
30             return -1;
31
32         /* Do swap frame from MEMRAM to MEMSWP and vice versa*/
33         /* Copy victim frame to swap */
34         __swap_cp_page(caller->mram, vicfpn, caller->active_mswp, swpfpn);
35
36         /* Copy target frame from swap to vicfpn in RAM */
37         __swap_cp_page(caller->active_mswp, tgtfpn, caller->mram, vicfpn);
38
39         /* Update page table */
40         pte_set_swap(&vicpte, 0, swpfpn);
41
42         /* Update its online status of the target page */
43         // pte_set_fpn() & mm->pgd[pgn];
44         pte_set_fpn(&pte, vicfpn);
45
46 #ifdef CPU_TLB
47         /* Update its online status of TLB (if needed) */
48         tlb_change_all_page_tables_of(caller, caller->mram);
```

```
49 #endif
50
51     /* Keep tracking */
52     mm->pgd[pgn] = pte;
53     mm->pgd[vicpgn] = vicpte;
54     enlist_pgn_node(&caller->mm->fifo_pgn, pgn);
55 }
56
57 *fpgn = PAGING_FPN(pte);
58
59 return 0;
60 }
```

The `pg_getpage` function is responsible for fetching a page from memory in a paging system. It takes the page number as input and retrieves the corresponding frame number where the page is stored. If the page is not currently in memory, the function initiates a page swap operation to bring the required page into memory from secondary storage. This function is crucial for memory access in a paging system, ensuring that the necessary pages are available in memory for efficient processing.

5. `validate_overlap_vm_area`

```
1 /*validate_overlap_vm_area
2  *@caller: caller
3  *@vmaid: ID vm area to alloc memory region
4  *@vmastart: vma end
5  *@vmaend: vma end
6  *
7  */
8 int validate_overlap_vm_area(struct pcb_t *caller, int vmaid, int vmastart,
9                             int vmaend)
10 {
11     struct vm_area_struct *vma = caller->mm->mmap;
12
13     while (vma != NULL)
14     {
15         if (vmaid != vma->vm_id)
16         {
17             if (OVERLAP(vmastart, vmaend, vma->vm_start, vma->vm_end))
18             {
```

```
18         if (((vmastart != vma->vm_end) && (vmaend != vma->vm_start)) ||  
19             ((vmastart == vma->vm_end) && (vmaend == vma->vm_start)))  
20             return -1;  
21     }  
22     vma = vma->vm_next;  
23 }  
24  
25 /* TODO validate the planned memory area is not overlapped */  
26  
27 return 0;  
28 }
```

The `validate_overlap_vm_area` function is responsible for checking whether a new memory area overlaps with existing memory regions within the virtual memory space. It iterates through existing memory regions. For each memory region, it checks whether there is any overlap with the new memory region. If overlap is detected, the function returns -1, indicating the new memory area cannot be allocated. If no overlap is found, it returns 0, signifying that the new memory area is valid for allocation.

6. find_victim_page

```
1 /*find_victim_page - find victim page  
2  * @caller: caller  
3  * @pgn: return page number  
4  *  
5  */  
6 int find_victim_page(struct mm_struct *mm, int *retpgn)  
7 {  
8     struct pgn_t *pg = mm->fifo_pgn;  
9  
10     /* TODO: Implement the theoretical mechanism to find the victim page */  
11     if (!pg)  
12         return 0;  
13  
14     if (!pg->pg_next)  
15     {  
16         *retpgn = pg->pgn;  
17         mm->fifo_pgn = NULL;  
18         free(pg);
```

```
19     return 0;
20 }
21
22 struct pgn_t *prev_Page;
23 while (pg->pg_next)
24 {
25     prev_Page = pg;
26     pg = pg->pg_next;
27 }
28
29 *retpgn = pg->pgn;
30 prev_Page->pg_next = NULL;
31 free(pg);
32
33 return 0;
34 }
```

This function is responsible for selecting a victim page in a paging system, typically used in virtual memory management. In a paging system, when a page fault occurs and a page needs to be replaced, the victim page is the one chosen for eviction from memory. It iterates through a linked list of page numbers until it reaches the end, then stores the page number of the victim page for eviction. This function is essential for managing memory in a paging system by determining which page to remove from memory to make space for new pages.

3.2.2 mm-memphy.c

1. MEMPHY_dump

```
1 int MEMPHY_dump(struct memphy_struct * mp)
2 {
3     /*TODO dump memphy contnt mp->storage
4      *   for tracing the memory content
5      */
6     for(int i = 0; i < mp->maxsz; i++)
7     {
8         /*if(mp->storage[i] != 0)
9         {
10             printf("mp->storage[%d] = %d\n", i, mp->storage[i]);
11         }*/
```

```
12     }  
13  
14     return 0;  
15 }
```

The function `MEMPHY_dump` iterates over the memory storage array within the `memphy_struct` `mp` and prints out the contents (`mp->storage`).

3.2.3 mm.c

1. `vmap_page_range`

```
1  /*  
2   * vmap_page_range - map a range of page at aligned address  
3   */  
4  int vmap_page_range(struct pcb_t *caller, // process call  
5                      int addr, // start address which is aligned to  
6                          pagesz  
7                          int pgnum, // num of mapping page  
8                          struct framephy_struct *frames, // list of the mapped frames  
9                          struct vm_rg_struct *ret_rg) // return mapped region, the real  
10                             mapped fp  
11  {  
12      // no guarantee all given pages are  
13      mapped  
14      //uint32_t * pte = malloc(sizeof(uint32_t));  
15      //struct framephy_struct *fpit = malloc(sizeof(struct framephy_struct));  
16      //int fpn;  
17      int pgit;  
18      //int pgn = PAGING_PGN(addr);  
19  
20      ret_rg->rg_end = ret_rg->rg_start = addr; // at least the very first space  
21      is usable  
22  
23      struct framephy_struct *fpit = frames;  
24  
25      /* TODO map range of frame to address space  
26       * [addr to addr + pgnum*PAGING_PAGESZ  
27       * in page table caller->mm->pgd[]  
28       */  
29      addr = PAGING_PGN(addr);  
30      for(pgit = addr; pgit < addr + pgnum; pgit++)
```

```
26     {
27         pte_set_fpn(&caller->mm->pgd[pgit], fpit->fpn);
28         fpit = fpit->fp_next;
29         /* Tracking for later page replacement activities (if needed)
30          * Enqueue new usage page */
31         enlist_pgn_node(&caller->mm->fifo_pgn, pgit);
32         if(fpit == NULL) break;
33     }
34
35     return 0;
36 }
```

This function is responsible for mapping a range of pages at an aligned address for a given process. The function initializes the start and end of the mapped region to the provided address. It iterates over the range of pages to be mapped. For each page, it sets the Page Table Entry (PTE) of the process's memory management unit (`caller->mm->pgd`) with the corresponding frame number from the provided list of frames. Additionally, it enqueues each page number for later page replacement activities, if needed.

3.3 Result

3.3.1 Input

Input file: `os_1_mlq_paging_small_4K`

```
1 2 4 8
2 4096 16777216 0 0 0
3 1 p0s 130
4 2 s3 39
5 4 m1s 15
6 6 s2 120
7 7 m0s 120
8 9 p1s 15
9 11 s0 38
10 16 s1 0
```

3.3.2 Output

```

Time slot 0
ld routine
Time slot 1
  Loaded a process at input/proc/p0s, PID: 1 PRI0: 130
  CPU 1: Dispatched process 1
Time slot 2
  Loaded a process at input/proc/s3, PID: 2 PRI0: 39
Time slot 3
  CPU 1: Put process 1 to run queue
  CPU 1: Dispatched process 1
  CPU 3: Dispatched process 2
Time slot 4
  Loaded a process at input/proc/mls, PID: 3 PRI0: 15
  CPU 2: Dispatched process 3
Time slot 5
  CPU 3: Put process 2 to run queue
  CPU 3: Dispatched process 2
  CPU 1: Put process 1 to run queue
  CPU 1: Dispatched process 1
Time slot 6
  write region=1 offset=20 value=100
  print pgtbl: 0 - 512
  00000000: 80000001
  00000004: 00000000
  Loaded a process at input/proc/s2, PID: 4 PRI0: 120
  CPU 2: Put process 3 to run queue
  CPU 1: Put process 1 to run queue
  CPU 1: Dispatched process 1
  read region=1 offset=20 value=100
  print pgtbl: 0 - 512
  00000000: 80000001
  00000004: 80000000
Time slot 7
  CPU 3: Put process 2 to run queue
  CPU 3: Dispatched process 2
  CPU 2: Dispatched process 3
  Loaded a process at input/proc/m0s, PID: 5 PRI0: 120
  CPU 0: Dispatched process 4
Time slot 8
  write region=3 offset=20 value=103
  print pgtbl: 0 - 512
  00000000: 80000001
  00000004: 80000000
  CPU 3: Put process 2 to run queue
  Loaded a process at input/proc/pls, PID: 6 PRI0: 15
  CPU 0: Put process 4 to run queue
  CPU 0: Dispatched process 6
  CPU 3: Dispatched process 2
Time slot 9
  CPU 2: Put process 3 to run queue
  CPU 2: Dispatched process 3
  CPU 1: Put process 1 to run queue
  CPU 1: Dispatched process 5
Time slot 10
  CPU 3: Put process 2 to run queue
  Loaded a process at input/proc/s0, PID: 7 PRI0: 38
  00000004: 80000004
Time slot 11
  CPU 3: Dispatched process 2
  CPU 0: Put process 6 to run queue
  CPU 0: Dispatched process 6
  CPU 2: Dispatched process 3
  CPU 1: Put process 5 to run queue
  CPU 1: Dispatched process 7
Time slot 12
Time slot 13
  CPU 0: Put process 6 to run queue
  CPU 0: Dispatched process 6
  CPU 2: Processed 3 has finished
  CPU 2: Dispatched process 4
  CPU 1: Put process 7 to run queue
  CPU 1: Dispatched process 7
  CPU 3: Put process 2 to run queue
  CPU 3: Dispatched process 2
  CPU 3: Processed 2 has finished
Time slot 14
  CPU 3: Dispatched process 5
  CPU 2: Put process 4 to run queue
  CPU 0: Put process 6 to run queue
Time slot 15
  CPU 1: Put process 7 to run queue
  CPU 1: Dispatched process 7
  CPU 2: Dispatched process 4
  CPU 0: Dispatched process 6
Time slot 16
  Loaded a process at input/proc/sl, PID: 8 PRI0: 0
  CPU 3: Put process 5 to run queue
  CPU 3: Dispatched process 8
Time slot 17
  CPU 1: Put process 7 to run queue
  CPU 1: Dispatched process 7
  CPU 2: Put process 4 to run queue
  CPU 0: Put process 6 to run queue
  CPU 0: Dispatched process 6
  CPU 2: Dispatched process 5
  write region=1 offset=20 value=102
  print pgtbl: 0 - 512
  00000000: 80000005
  00000004: 80000004
Time slot 18
  CPU 3: Put process 8 to run queue
  CPU 3: Dispatched process 8
  write region=2 offset=1000 value=1
  print pgtbl: 0 - 512
  00000000: 80000005
  00000004: 80000004
  CPU 2: Put process 5 to run queue
  CPU 1: Put process 7 to run queue
  CPU 1: Dispatched process 7
  CPU 0: Processed 6 has finished
  CPU 0: Dispatched process 4
  CPU 2: Dispatched process 5
  write region=0 offset=0 value=0
  print pgtbl: 0 - 512
  00000000: c0000000
Time slot 19
  CPU 3: Put process 8 to run queue
  CPU 3: Dispatched process 8
  CPU 2: Processed 5 has finished
Time slot 20
  CPU 2: Dispatched process 1
  read region=3 offset=20 value=103
  print pgtbl: 0 - 512
  00000000: 80000001
  00000004: 80000000
Time slot 21
  CPU 0: Put process 4 to run queue
  CPU 0: Dispatched process 4
  CPU 1: Put process 7 to run queue
  CPU 1: Dispatched process 7
  CPU 3: Put process 8 to run queue
  CPU 3: Dispatched process 8
Time slot 22
  CPU 2: Processed 1 has finished
  CPU 2 stopped
Time slot 23
  CPU 0: Put process 4 to run queue
  CPU 3: Processed 8 has finished
  CPU 1: Put process 7 to run queue
  CPU 0: Dispatched process 4
  CPU 1: Dispatched process 7
  CPU 3 stopped
Time slot 24
Time slot 25
  CPU 0: Processed 4 has finished
  CPU 1: Put process 7 to run queue
  CPU 0 stopped
  CPU 1: Dispatched process 7
Time slot 26
  CPU 1: Processed 7 has finished
  CPU 1 stopped

```

Figure 7: Output of os_1_mfq_paging_small_4K

Khi thực hiện lệnh WRITE và MM_PAGING được define, chương trình sẽ gọi hàm pgwrite. Hàm này in thông tin về chỉ số của register đích, offset và data dưới dạng "write region=%d offset=%d value=%d", và sau đó gọi hàm print_pgtbl.

Tương tự, khi thực hiện lệnh READ và MM_PAGING được define, chương trình sẽ gọi hàm pgread. Hàm này in ra thông tin về chỉ số của register nguồn, offset và data dưới dạng "read region=%d offset=%d value=%d", và sau đó gọi hàm print_pgtbl.

3.4 Status of the mapped page and the index page involving TLB

This is the diagram displaying the status of the mapped page and the index page involved TLB procedure and the handling for the case when page miss occurs.

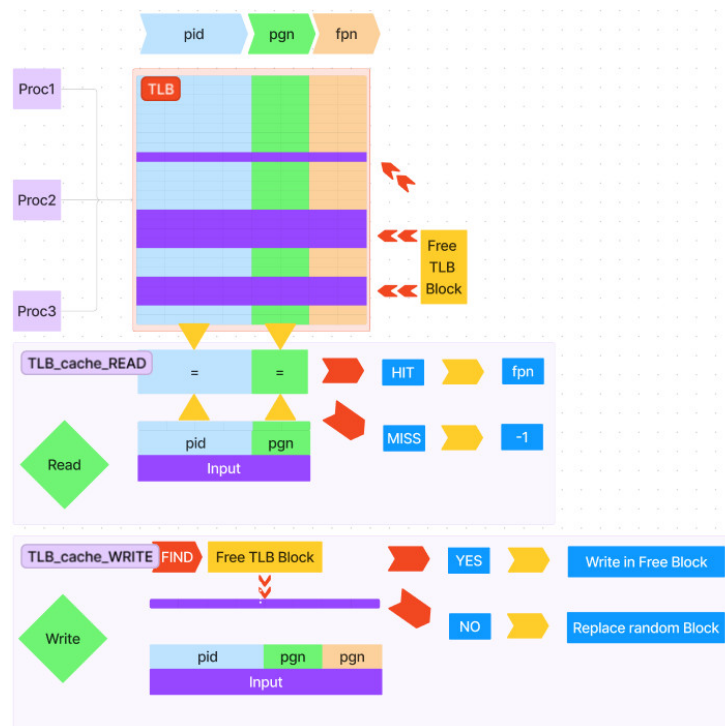


Figure 8: Status Mapped of Paging and TLB

3.5 Questions

Question 2: 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: Having multiple memory segments in the OS design offers:

- Resource Partitioning: Allows for efficient allocation and management of distinct memory regions.
- Enhanced Control: Provides finer-grained control over memory usage and access.
- Fault Isolation: Minimizes the impact of errors or faults in one segment on others, enhancing system robustness.
- Customization: Facilitates tailored memory management strategies for different system components or applications.

- **Optimized Performance:** Enables specialized handling of memory requirements, leading to better overall system performance.

Question 3: What will happen if we divide the address to more than 2-levels in the paging memory management system?

Answer: Adding more levels to the paging system increases the address space and potentially improves performance by reducing memory overhead and enhancing flexibility. However, it also raises complexity and may lead to increased overhead, memory fragmentation, slower translation times, and cache pressure.

Question 4: What is the advantage and disadvantage of segmentation with paging?

Answer:

Advantages:

- Segmentation allows for logical partitioning of memory into segments, enabling better organization and management of memory resources.
- Segmentation provides memory protection by assigning different privileges to different segments, enhancing security and preventing unauthorized access.
- Paging offers virtual memory support, allowing processes to use more memory than physically available by swapping memory pages between RAM and disk.
- Paging reduces internal fragmentation by breaking memory into fixed-size pages, making more efficient use of memory space.

Disadvantages:

- Segmentation with paging introduces additional complexity in memory management algorithms and hardware support compared to either segmentation or paging alone.
- It may suffer from both external and internal fragmentation, especially if the segments are of varying sizes and the pages are small.
- The combination of segmentation and paging can lead to increased overhead in terms of memory access time, address translation, and management operations.
- Implementing a segmentation with paging system requires sophisticated hardware support and software algorithms, making it challenging to design and maintain.

4 TLB Memory Management

4.1 Introduction of TLB

The Translation Lookaside Buffer (TLB) is a specialized cache in a computer's memory management unit (MMU) that stores recently used virtual-to-physical address translations. Its primary purpose is to accelerate virtual address translation by providing fast access to frequently accessed translation mappings.

Here's a list of its key characteristics and functionalities:

- **Address Translation Acceleration:** When a program accesses memory, it typically uses virtual addresses. These addresses need to be translated into physical addresses before accessing actual memory locations. The TLB caches recently used translations, speeding up this translation process.
- **Associative Memory:** TLBs are typically implemented as associative memory or content-addressable memory (CAM), enabling rapid lookup of translations. This associative memory allows the TLB to store mappings between virtual and physical addresses.
- **Cache Management:** TLBs have limited capacity compared to the total number of possible translations in a system. Therefore, they employ cache management policies to decide which translations to keep and which ones to evict when the cache is full. Common policies include least recently used (LRU), random replacement, or first in, first out (FIFO).
- **Performance Impact:** TLBs significantly impact system performance by reducing the time required for address translation. Without a TLB, every memory access would require a translation table lookup, which can be slow, especially in systems with large address spaces.
- **Integration with MMU:** TLBs are closely integrated with the memory management unit (MMU) of a processor. The MMU handles the translation of virtual addresses to physical addresses and relies on the TLB for fast lookup of translations.

Overall, the TLB plays a crucial role in improving the efficiency of virtual memory systems by reducing the overhead of address translation, thereby enhancing system performance.

4.2 TLB Operations in the Assignment

In the previous section the operating systems and memory management subsystem implement that each process has its own page table. This table contains page table entry which

provide the frame number. The challenge lies in optimizing access time for these entries, the 2x access time of reading page table (which is actually placed on main memory or actually... a MEMPHY) and accessing the memory data in MEMPHY.

Given a large process sizes resulting in a high overhead, TLB is proposed to leverage cache capabilities due to its high-speed nature. TLB is ordinarily MEMPHY but acts as a high-speed cache for page table entries. However, memory cache is high cost component; therefore, it has a limited capacity. These are the fundamental works of TLB:

- **TLB accessing method:** TLB is MEMPHY with support the mapping mechanism to determine how the content is associated with the identifier info. In this work, we leverage the equipped knowledge of previous course about the computer hardware, where it employs the cache mapping techniques: direct mapped/ set associative/ fully associative.
- **TLB setup:** TLB contains recently used page table entries. When the CPU generates a virtual address, it checks the TLB:
 - If a page table entry is present (a TLB hit), the corresponding frame number is retrieved.
 - If a page table entry is not found (a TLB miss), the page number is used as an index to access the page table in main memory. If the page is not in main memory, a page fault occurs, and the TLB is updated with the new page entry.
- **TLB Hit:**
 - CPU generates a virtual address.
 - TLB is checked (entry present).
 - Corresponding frame number is retrieved.
- **TLB Miss:**
 - CPU generates a virtual address.
 - TLB is checked (entry not present).
 - Page number is matched to the page table in main memory.
 - Corresponding frame number is retrieved.

Since TLB is purely a memory storage device, you can design an efficient method to determine the cache mapping. We do not fix a design, We just provide a suggestion of a mapping proposal based on pid and page number, noting that since TLB cache is shared by all system processes and is intended for usage at CPU-level; therefore, *pid* is necessary.

```
1  /*
2  * tlb_cache_read read TLB cache device
3  * @mp: memphy struct
4  * @pid: process id
5  * @pgnum: page number
6  * @value: obtained value
7  */
8  int tlb_cache_read(struct memphy_struct * mp, int pid, int pgnum, BYTE value)(...)
9  /*
10 * tlb_cache_write write TLB cache device
11 * @mp: memphy struct
12 * @pid: process id
13 * @pgnum: page number
14 * @value: obtained value
15 */
16 int tlb_cache_write(struct memphy_struct *mp, int pid, int pgnum, BYTE value)(...)
```

TLB operations (tlballoc/tlbfree/tlbread/tlbwrite) happen before memory paging works (alloc/ free/ read/ write). The translation scheme passes address through these layers in a particular order when a user sends memory requests and then again in reverse order when the address is received. Due to its native hierarchical programming paradigm, if you think it will make things easier, you are free to implement TLB CACHED updates in any module (tlb or mm-vm) via bypassing among modules. We have already reserved a designed bit in order to facilitate page manipulation purpose, we hope that the reserved portion would be helpful. For a general operation (replace xxx with alloc/free/read/write):

```
1  int tlbxxx(struct pcb_t *proc, uint32_t size, uint32_t reg_index)
2  {
3  int addr, val;
4  /* TODO preceding update TLB CACHED (if needed) */
5  /* by using tlb_cache_read()/tlb_cache_write() */
6  /* Perform paging operations using vmaid = 0 by default */
7  val = __xxx(proc, 0, reg_index, size, &addr);
8
9  /* TODO suffixing update TLB CACHED (if needed) */
10 return val;
11 }
```

4.3 Implementation

4.3.1 TLB operations

TLB operations are implemented in file `cpu-tlb.c`

1. `tlb_change_all_page_tables_of()`

```
1  int tlb_change_all_page_tables_of(struct pcb_t *proc, struct memphy_struct *
    mp)
2  {
3      /* TODO update all page table directory info
4       *      in flush or wipe TLB (if needed)
5       */
6      pthread_mutex_lock(&cache_lock);
7      /* Iterate over all entries in TLB */
8      for(int i = 0; i < TLB_SIZE * TLB_ENTRY_SIZE; i += TLB_ENTRY_SIZE)
9      {
10         // Extracting pid, pgnum, data from current tlb_entry
11         int used = mp->storage[i];
12         if ((used & 1) == 0)
13             continue;
14         int entry_pid = 0;
15         for(int j = 0; j <= 3; j++)
16         {
17             entry_pid |= (proc->tlb->storage[i + 4 - j] << (j * 8));
18         }
19         int entry_pgnum = (mp->storage[i + 5] << 8) | mp->storage[i + 6];
20         int entry_data = (mp->storage[i + 7] << 8) | mp->storage[i + 8]; //
            Frame number
21
22         /* Update the corresponding entry in the page table */
23         if(proc->pid == entry_pid)
24         {
25             pte_set_fpn(&proc->mm->pgd[entry_pgnum], entry_data);
26         }
27     }
28     pthread_mutex_unlock(&cache_lock);
29
30     return 0;
31 }
```

The function `tlb_change_all_page_tables_of()` is used to update all page table directory information in the TLB cache when necessary, ensuring that it reflects the current stage of the page tables. It iterates over all entries in the TLB cache and updates the corresponding page table directory information. This function is essential for maintaining consistency between the TLB cache and the page tables during operations such as page table updates or TLB flushes.

2. `tlb_flush_tlb_of()`

```
1 int tlb_flush_tlb_of(struct pcb_t *proc, struct memphy_struct * mp)
2 {
3     /* TODO flush tlb cached*/
4     if(proc == NULL || mp == NULL)
5         return 0;
6     pthread_mutex_lock(&cache_lock);
7     for(int i = 0; i < TLB_SIZE * TLB_ENTRY_SIZE; i += TLB_ENTRY_SIZE)
8     {
9         int entry_pid = 0;
10        for(int j = 0; j <= 3; j++)
11        {
12            entry_pid |= (proc->tlb->storage[i + 4 - j] << (j * 8));
13        }
14        if(proc->pid == entry_pid)
15            TLBMEMPHY_write(mp, i, 0);
16    }
17    pthread_mutex_unlock(&cache_lock);
18    return 0;
19 }
```

The `tlb_flush_tlb_of()` function flushes the TLB cache entries associated with a specific process. It iterates over all entries in the TLB cache, checks if the entry corresponds to the given process, and invalidates the entry if it matches. This function ensures that the TLB cache is consistent with the current state of the process's memory mappings.

3. `tlballoc()`

```
1 /*tlballoc - CPU TLB-based allocate a region memory
2  *@proc: Process executing the instruction
3  *@size: allocated size
4  *@reg_index: memory region ID (used to identify variable in symbole table)
5  */
```

```
6 int tlballoc(struct pcb_t *proc, uint32_t size, uint32_t reg_index)
7 {
8     int addr, val;
9     printf("TLB_alloc, proc: %d\n", proc->pid);
10
11     /* By default using vmaid = 0 */
12     val = __alloc(proc, 0, reg_index, size, &addr);
13
14     /* TODO update TLB CACHED frame num of the new allocated page(s)*/
15     /* by using tlb_cache_read()/tlb_cache_write()*/
16
17     /* Calculate the number of pages allocated */
18     int vpn = proc->regs[reg_index] / PAGE_SIZE;
19     int num_pages = size / PAGE_SIZE;
20     if(size % PAGE_SIZE != 0)
21         num_pages++; /* Add 1 more page if size is not a multiple of PAGE_SIZE */
22     /* Update TLB for each allocated page */
23     for(int i = vpn; i < vpn + num_pages; i++)
24     {
25         /* Get frame number from page table */
26         int fn = proc->mm->pgd[i] & PAGING_PTE_FPN_MASK;
27         /* Write the frame number to TLB */
28         tlb_cache_write(proc->tlb, proc->pid, i, &fn);
29     }
30
31     TLBMEMPHY_dump(proc->tlb);
32     return val;
33 }
```

The `tlballoc()` function allocates a region of memory for a process and updates the TLB cache with the corresponding frame numbers for the newly allocated pages. It first allocates memory using the `__alloc` function, then calculates the number of pages allocated based on the provided size. Next, it retrieves the frame numbers from the process's page table and updates the TLB cache with the frame numbers for each allocated page. Finally, it prints the TLB cache contents for debugging purposes.

4. `tlbfree_data()`

```
1 /*pgfree - CPU TLB-based free a region memory
2 *@proc: Process executing the instruction
```

```
3  *@size: allocated size
4  *@reg_index: memory region ID (used to identify variable in symbole table)
5  */
6  int tlbfree_data(struct pcb_t *proc, uint32_t reg_index)
7  {
8      pthread_mutex_lock(&cache_lock);
9      printf("TLB_free, proc: %d\n", proc->pid);
10     __free(proc, 0, reg_index);
11
12     /* TODO update TLB CACHED frame num of freed page(s)*/
13     /* by using tlb_cache_read()/tlb_cache_write()*/
14
15     /* Get the virtual page number of the freed region */
16     int vpn = proc->regs[reg_index] / PAGE_SIZE;
17
18     /* Iterate over all entries in the TLB */
19     for(int i = 0; i < TLB_SIZE * TLB_ENTRY_SIZE; i += TLB_ENTRY_SIZE)
20     {
21         int used = proc->tlb->storage[i];
22         if ((used & 1) == 0)
23             continue;
24         int entry_pid = 0;
25         for(int j = 0; j <= 3; j++)
26         {
27             entry_pid |= (proc->tlb->storage[i + 4 - j] << (j * 8));
28         }
29         int entry_pgnum = (proc->tlb->storage[i + 5] << 8) | proc->tlb->storage[i
30             + 6];
31
32         /* Check if the entry correspond to the freed region */
33         if(proc->pid == entry_pid && vpn == entry_pgnum)
34         {
35             // Invalidate the entry
36             TLBMEMPHY_write(proc->tlb, i, 0);
37         }
38
39     }
40     pthread_mutex_unlock(&cache_lock);
41     return 0;
42 }
```


The `tlbfree_data()` function locks the cache, then deallocates a region of memory for a process and updates the TLB cache to reflect the freed pages. It first frees memory using the `__free` function, then retrieves the virtual page number of the freed region. Next, it iterates over the TLB cache entries to find and invalidate any entries corresponding to the freed region. Finally, it unlocks the cache lock to allow other threads to access the TLB cache.

5. `tlbread()`

```
1  /*tlbread - CPU TLB-based read a region memory
2  *@proc: Process executing the instruction
3  *@source: index of source register
4  *@offset: source address = [source] + [offset]
5  *@destination: destination storage
6  */
7  int tlbread(struct pcb_t * proc, uint32_t source,
8              uint32_t offset,  uint32_t destination)
9  {
10     BYTE data;
11     int frmnum = -1;
12     printf("TLB_read, proc: %d\n", proc->pid);
13
14     /* TODO retrieve TLB CACHED frame num of accessing page(s)*/
15     /* by using tlb_cache_read()/tlb_cache_write()*/
16     /* frmnum is return value of tlb_cache_read/write value*/
17     int vpn = proc->regs[source] / PAGE_SIZE;
18     tlb_cache_read(proc->tlb, proc->pid, vpn, &frmnum);
19     #ifdef IODUMP
20     if (frmnum >= 0)
21         printf("TLB hit at read region=%d offset=%d\n",
22               source, offset);
23     else
24         printf("TLB miss at read region=%d offset=%d\n",
25               source, offset);
26     #ifdef PAGETBL_DUMP
27     print_pgtbl(proc, 0, -1); //print max TBL
28     #endif
29     MEMPHY_dump(proc->mram);
30     TLBMEMPHY_dump(proc->tlb);
```

```
31 #endif
32 // Miss -> get frmnum from table
33 if(frmnum == -1)
34 {
35     frmnum = proc->mm->pgd[vpn] & PAGING_PTE_FPN_MASK;
36     tlb_cache_write(proc->tlb, proc->pid, vpn, &frmnum);
37 }
38
39 int val = __read(proc, 0, source, offset, &data);
40
41 destination = (uint32_t) data;
42
43 /* TODO update TLB CACHED with frame num of recent accessing page(s)*/
44 /* by using tlb_cache_read()/tlb_cache_write()*/
45 return val;
46 }
```

The `tlbread()` function reads data from a region of memory based on a virtual address provided by the process. It first attempts to retrieve the frame number from the TLB cache corresponding to the virtual page number derived from the source register. If the TLB cache hit occurs, it reads the data directly from the memory and updates the destination register with the retrieved value. If there's a TLB miss, it retrieves the frame number from the page table and updates the TLB cache before performing the memory read operation. Finally, it returns the result of the memory read operation.

6. `tlbwrite()`

```
1 /*tlbwrite - CPU TLB-based write a region memory
2  *@proc: Process executing the instruction
3  *@data: data to be wrttien into memory
4  *@destination: index of destination register
5  *@offset: destination address = [destination] + [offset]
6  */
7 int tlbwrite(struct pcb_t * proc, BYTE data,
8             uint32_t destination, uint32_t offset)
9 {
10     int val;
11     int frmnum = -1;
12     printf("TLB_write, proc: %d\n", proc->pid);
13 }
```

```
14  /* TODO retrieve TLB CACHED frame num of accessing page(s)*/
15  /* by using tlb_cache_read()/tlb_cache_write()
16  frmnum is return value of tlb_cache_read/write value*/
17  int vpn = proc->regs[destination] / PAGE_SIZE;
18  tlb_cache_read(proc->tlb, proc->pid, vpn, &frmnum);
19
20  #ifdef IODUMP
21      if (frmnum >= 0)
22          printf("TLB hit at write region=%d offset=%d value=%d\n",
23                destination, offset, data);
24      else
25          printf("TLB miss at write region=%d offset=%d value=%d\n",
26                destination, offset, data);
27  #ifdef PAGETBL_DUMP
28      print_pgtbl(proc, 0, -1); //print max TBL
29  #endif
30      MEMPHY_dump(proc->mram);
31      TLBMEMPHY_dump(proc->tlb);
32  #endif
33  // Miss -> get frmnum from table
34  if(frmnum == -1)
35  {
36      frmnum = proc->mm->pgd[vpn] & PAGING_PTE_FPN_MASK;
37      tlb_cache_write(proc->tlb, proc->pid, vpn, &frmnum);
38  }
39
40  val = __write(proc, 0, destination, offset, data);
41
42  /* TODO update TLB CACHED with frame num of recent accessing page(s)*/
43  /* by using tlb_cache_read()/tlb_cache_write()*/
44  return val;
45 }
```

The `tlbwrite` function writes data into a region of memory based on a virtual address provided by the process. It first attempts to retrieve the frame number from the TLB cache corresponding to the virtual page number derived from the destination register. If the TLB cache hit occurs, it writes the data directly into the memory. If there's a TLB miss, it retrieves the frame number from the page table and updates the TLB cache before performing the memory write operation. Finally, it returns the result of the memory write

operation.

4.3.2 TLB Cache operations

This report displays the TLB Cache operations in 2 ways, **Direct Mapped** (Implemented in file `cpu-tlbcache-1.c`) and **Fully Associative** (Implemented in file `cpu-tlbcache.c`). They all share the same `TLBMEMPHY_dump()` function:

```
1 int TLBMEMPHY_dump(struct memphy_struct * mp)
2 {
3     /*TODO dump memphy contnt mp->storage
4      *   for tracing the memory content
5      */
6     printf("TLBMEMPHY_dump\n");
7     printf("TLB Cache Start\n");
8     // Check whether the physical memory exists or not
9     if(mp == NULL || mp->storage == NULL)
10    {
11        printf("Physical memory doesn't exist.");
12        return -1;
13    }
14
15    pthread_mutex_lock(&cache_lock);
16    /* Iterate over all tlb_entry in mp->storage */
17    for(int i = 0; i < TLB_SIZE * TLB_ENTRY_SIZE; i += TLB_ENTRY_SIZE)
18    {
19        // Extracting pid, pgnum, data from current tlb_entry
20        int used = mp->storage[i];
21        if ((used & 1) == 0)
22            continue;
23        int entry_pid;
24        for(int j = 0; j <= 3; j++)
25        {
26            entry_pid |= (mp->storage[i + 4 - j] << (j * 8));
27        }
28        int entry_pgnum = (mp->storage[i + 5] << 8) | mp->storage[i + 6];
29        int entry_data = (mp->storage[i + 7] << 8) | mp->storage[i + 8]; // Frame
30                                number
31        printf("Entry %d:\tUsed: %d\tEntry pid: %d\tEntry pagenum: %d\tEntry
32              framenum: %d\n", i/TLB_ENTRY_SIZE, used, entry_pid, entry_pgnum,
33              entry_data);
```

```
31     }  
32  
33     pthread_mutex_unlock(&cache_lock);  
34     printf("TLB Cache End\n");  
35     return 0;  
36 }
```

The `TLBMEMPHY_dump()` function prints the contents of the TLB cache memory. It iterates over each entry in the TLB cache memory and displays information such as whether the entry is used or not, the process ID, the page number, and the frame number. This function provides a way to trace the contents of the TLB cache memory for debugging and analysis purposes.

With 2 functions `tlb_cache_read()` and `tlb_cache_write()`, there's a slight change between **Direct Mapped** and **Fully Associative**. The primary difference between those 2 ways is in how they handle the mapping of virtual addresses to TLB entries:

- **Direct Mapped:** Implemented in file `cpu-tlbcache-1.c`

- Each virtual page is mapped to a specific entry in the TLB based on a simple hashing function (often modulo operation).
- Each TLB entry corresponds to a specific virtual page, and there's only one place where a given virtual page can be stored in the TLB.
- If a virtual address maps to a TLB entry that is already occupied, it will replace the existing entry in that specific slot.

1. `tlb_cache_read()`

```
1 // need 1 bit for detect the entry is used or not: 0 = Not used, 1 = Used  
2 // 32 bit (unsigned int) = 4 bytes for pid  
3 // need 14 bit for pgnum -> 2 bytes  
4 // frame number has 13 bit = 2 byte  
5 // -/----/--/-- => 1 entry takes 9 byte  
6 // pid/pgnum/frame number  
7  
8 /*  
9  * tlb_cache_read read TLB cache device  
10  * @mp: memphy struct  
11  * @pid: process id  
12  * @pgnum: page number  
13  * @value: obtained value  
14  */
```

```
15  int tlb_cache_read(struct memphy_struct * mp, int pid, int pgnum, int*
    value)
16  {
17      /* TODO: the identify info is mapped to
18       *      cache line by employing:
19       *      direct mapped.
20       */
21      pthread_mutex_lock(&cache_lock);
22      // Calculate index of TLB cache entry using direct-mapped mapping
23      int cache_index = pgnum % TLB_SIZE;
24      int base_entry_addr = cache_index * TLB_ENTRY_SIZE;
25
26      // Extracting pid, pgnum, data from current tlb_entry
27      int used = mp->storage[base_entry_addr];
28      if ((used & 1) == 0)
29          return -1;
30      int entry_pid = 0;
31      for(int j = 3; j >= 0; j--)
32      {
33          entry_pid |= (mp->storage[base_entry_addr + 4 - j] << (j * 8));
34      }
35      int entry_pgnum = (mp->storage[base_entry_addr + 5] << 8) |
        mp->storage[base_entry_addr + 6];
36      int entry_data = (mp->storage[base_entry_addr + 7] << 8) |
        mp->storage[base_entry_addr + 8]; // Frame number
37
38      /* If pid and pgnum of current tlb_entry match with the original pid
        and pgnum */
39      if(entry_pid == pid && entry_pgnum == pgnum)
40      {
41          // Set value to current data of tlb_entry and return 0
42          *value = entry_data;
43          pthread_mutex_unlock(&cache_lock);
44          return 0;
45      }
46
47      pthread_mutex_unlock(&cache_lock);
48      // If no precise tlb_entry matched, return -1
49      return -1;
50  }
```

The `tlb_cache_read` function retrieves the frame number associated with a specific process ID and page number from the TLB cache memory. In the direct-mapped implementation, it employs a simple direct mapping scheme where each TLB entry is associated with a specific page number. The function calculates the cache index using the page number modulo the TLB size and checks if the entry matches the provided process ID and page number. If a match is found, it returns the corresponding frame number through the `value` pointer. If no match is found, it returns -1 to indicate a TLB miss.

2. `tlb_cache_write()`

```
1  /*
2   * tlb_cache_write write TLB cache device
3   * @mp: memphy struct
4   * @pid: process id
5   * @pgnum: page number
6   * @value: obtained value
7   */
8  int tlb_cache_write(struct memphy_struct *mp, int pid, int pgnum, int
   *value)
9  {
10     /* TODO: the identify info is mapped to
11      *      cache line by employing:
12      *      direct mapped.
13      */
14     pthread_mutex_lock(&cache_lock);
15     int cache_index = pgnum % TLB_SIZE;
16     int base_entry_addr = cache_index * TLB_ENTRY_SIZE;
17     // Write pid to cache
18     for(int i = 0; i <= 3; i++)
19     {
20         TLBMEMPHY_write(mp, base_entry_addr + 4 - i, (pid >> (i * 8)) &
           0xFF);
21     }
22     // Write pgnum to cache
23     TLBMEMPHY_write(mp, base_entry_addr + 6, pgnum & 0xFF);
24     TLBMEMPHY_write(mp, base_entry_addr + 5, (pgnum >> 8) & 0xFF);
25     // Write value(frame number) to cache
26     TLBMEMPHY_write(mp, base_entry_addr + 8, *value & 0xFF);
```

```
27     TLBMEMPHY_write(mp, base_entry_addr + 7, (*value >> 8) & 0xFF);
28
29     // Mark used entry
30     TLBMEMPHY_write(mp, base_entry_addr, 1);
31
32     pthread_mutex_unlock(&cache_lock);
33     return 0;
34 }
```

The `tlb_cache_write` function writes a new entry into the TLB cache memory based on a direct-mapped scheme. It associates the provided process ID and page number with the given frame number in the TLB cache. In a direct-mapped implementation, each page number maps to a specific location in the TLB cache, determined by the page number modulo the TLB size. If the corresponding entry is already occupied, it replaces the existing entry.

- **Fully Associative:** Implemented in file `cpu-tlbcache.c`

- Virtual pages can be mapped to any TLB entry, allowing for more flexibility.
- There's no restriction on where a virtual page can be stored in the TLB. Any free entry can be used.
- Replacement policies such as LRU (Least Recently Used) or FIFO (First-In, First-Out) are typically employed to determine which entry to replace when the TLB is full.

1. `tlb_cache_read()`

```
1  // need 1 bit for detect the entry is used or not: 0 = Not used, 1 = Used
2  // 32 bit (unsigned int) = 4 bytes for pid
3  // need 14 bit for pgnum -> 2 bytes
4  // frame number has 13 bit = 2 byte
5  // -/----/--/-- => 1 entry takes 9 byte
6  // pid/pgnum/frame number
7
8  /*
9   * tlb_cache_read read TLB cache device
10  * @mp: memphy struct
11  * @pid: process id
12  * @pgnum: page number
13  * @value: obtained value
```



```
14  */
15  int tlb_cache_read(struct memphy_struct * mp, int pid, int pgnum, int*
    value)
16  {
17      /* TODO: the identify info is mapped to
18       *      cache line by employing:
19       *      associated mapping etc.
20       */
21      pthread_mutex_lock(&cache_lock);
22      /* Iterate over all tlb_entry in mp->storage, cause we are using fully
        associative */
23      for(int i = 0; i < TLB_SIZE * TLB_ENTRY_SIZE; i += TLB_ENTRY_SIZE)
24      {
25          // Extracting pid, pgnum, data from current tlb_entry
26          int used = mp->storage[i];
27          if ((used & 1) == 0)
28              continue;
29          int entry_pid = 0;
30          for(int j = 3; j >= 0; j--)
31          {
32              entry_pid |= (mp->storage[i + 4 - j] << (j * 8));
33          }
34          int entry_pgnum = (mp->storage[i + 5] << 8) | mp->storage[i + 6];
35          int entry_data = (mp->storage[i + 7] << 8) | mp->storage[i + 8]; //
            Frame number
36
37          /* If pid and pgnum of current tlb_entry match with the original pid
            and pgnum */
38          if(entry_pid == pid && entry_pgnum == pgnum)
39          {
40              // Set value to current data of tlb_entry and return 0
41              *value = entry_data;
42              pthread_mutex_unlock(&cache_lock);
43              return 0;
44          }
45      }
46      pthread_mutex_unlock(&cache_lock);
47      // If no precise tlb_entry matched, return -1
48      return -1;
49  }
```

The `tlb_cache_read` function retrieves the frame number associated with the given process ID and page number from the TLB cache memory in a fully associative manner. It searches through all entries in the TLB cache to find the entry matching the provided process ID and page number. If a match is found, the function returns the corresponding frame number; otherwise, it indicates a TLB miss by returning -1.

2. `tlb_cache_write()`

```
1  /*
2   * tlb_cache_write write TLB cache device
3   * @mp: memphy struct
4   * @pid: process id
5   * @pgnum: page number
6   * @value: obtained value
7   */
8  int tlb_cache_write(struct memphy_struct *mp, int pid, int pgnum, int
   *value)
9  {
10     /* TODO: the identify info is mapped to
11      *      cache line by employing:
12      *      associated mapping etc.
13      */
14     pthread_mutex_lock(&cache_lock);
15     int free_entry = -1;
16     /* Iterate over all tlb_entry in mp->storage, cause we are using fully
17      * associative */
18     for(int i = 0; i < TLB_SIZE * TLB_ENTRY_SIZE; i += TLB_ENTRY_SIZE)
19     {
20         int flag = 0;
21         for(int j = 0; j <= 8; j++)
22         {
23             if(mp->storage[i + j] != 0)
24             {
25                 flag = 1;
26                 break;
27             }
28         }
29         if(flag == 0)
```

```
29     {
30         free_entry = i / TLB_ENTRY_SIZE;
31         break;
32     }
33 }
34 if(free_entry == -1)
35 { // Full TLB
36     free_entry = rand() % TLB_SIZE;
37 }
38 int base_entry_addr = free_entry * TLB_ENTRY_SIZE;
39 // Write pid to cache
40 for(int i = 0; i <= 3; i++)
41 {
42     TLBMEMPHY_write(mp, base_entry_addr + 4 - i, (pid >> (i * 8)) &
43         0xFF);
44 }
45 // Write pgnum to cache
46 TLBMEMPHY_write(mp, base_entry_addr + 6, pgnum & 0xFF);
47 TLBMEMPHY_write(mp, base_entry_addr + 5, (pgnum >> 8) & 0xFF);
48 // Write value(frame number) to cache
49 TLBMEMPHY_write(mp, base_entry_addr + 8, *value & 0xFF);
50 TLBMEMPHY_write(mp, base_entry_addr + 7, (*value >> 8) & 0xFF);
51 // Mark used entry
52 TLBMEMPHY_write(mp, base_entry_addr, 1);
53
54 pthread_mutex_unlock(&cache_lock);
55 return 0;
56 }
```

The `tlb_cache_write` function writes the process ID, page number, and frame number into the TLB cache memory in a fully associative manner. It searches for an empty or available entry in the TLB cache to store the provided information. If no empty entry is found, it randomly selects an entry to overwrite. Once the entry is selected, it updates the cache with the provided process ID, page number, and frame number.



4.4 Result

4.4.1 Input

Input file: os_1_tlbsz_singleCPU_mlq.txt

```
1 2 1 8
2 40000
3 1 s4 4
4 2 s3 3
5 4 m1s 2
6 6 s2 3
7 7 m0s 3
8 9 p1s 2
9 11 s0 1
10 16 s1 0
```

The first line is: [time slice] [N = Number of CPU] [M = Number of Processes to be run].

The second line is TLB size in CPU.

From line 3: [time i] [path i] [priority i].

To compile, first we have to adjust in file `os-cfg.h`: Turn on `MM_FIXED_MEMSZ`, and turn off `CPUTLB_FIXED_TLBSZ`, because we are implementing TLB so we need a fixed memory size, and we have TLB size changes (it is passed in input file)

```
1 #ifndef OSCFG_H
2 #define OSCFG_H
3
4 #define MLQ_SCHEDULED 1
5 #define MAX_PRIO 140
6
7 #define CPU_TLB
8 //#define CPUTLB_FIXED_TLBSZ
9 #define MM_PAGING
10 #define MM_FIXED_MEMSZ
11 //#define VMDBG 1
12 //#define MMDBG 1
13 #define IODUMP 1
14 #define PAGETBL_DUMP 1
15
16 #endif
```

Then we run order make all, then run ./os os_1_tlbsz_singleCPU_m1q

```
antwerp2004@antwerp2004-VirtualBox:~/ossim_mm_tlb_hk241$ make all
gcc -Iinclude -Wall -c -g src/cpu.c -o obj/cpu.o
gcc -Iinclude -Wall -c -g src/cpu-tlb.c -o obj/cpu-tlb.o
gcc -Iinclude -Wall -c -g src/mem.c -o obj/mem.o
gcc -Iinclude -Wall -c -g src/loader.c -o obj/loader.o
gcc -Iinclude -Wall -c -g src/queue.c -o obj/queue.o
gcc -Iinclude -Wall -c -g src/os.c -o obj/os.o
gcc -Iinclude -Wall -c -g src/sched.c -o obj/sched.o
gcc -Iinclude -Wall -c -g src/timer.c -o obj/timer.o
gcc -Iinclude -Wall -c -g src/mm-vm.c -o obj/mm-vm.o
gcc -Iinclude -Wall -c -g src/mm.c -o obj/mm.o
gcc -Iinclude -Wall -c -g src/mm-memphy.c -o obj/mm-memphy.o
gcc -Iinclude -Wall -g obj/cpu.o obj/cpu-tlb.o obj/cpu-tlbcache-1.o obj/mem.o obj/loader.o obj/queue.o obj/os.o obj/sched.o obj/timer.o obj/mm-vm.o obj/mm.o obj/mm-memphy.o -o os -lpthread
antwerp2004@antwerp2004-VirtualBox:~/ossim_mm_tlb_hk241$ ./os os_1_tlbsz_singleCPU_m1q
Time slot 0
ld_routine
Time slot 1
    Loaded a process at input/proc/s4, PID: 1 PRI0: 4
Time slot 2
    CPU 0: Dispatched process 1
    Loaded a process at input/proc/s3, PID: 2 PRI0: 3
Time slot 3
Time slot 4
```

Figure 9: Running the file

4.4.2 Output of Direct Mapped TLB

```
antwerp2004@antwerp2004-VirtualBox:~/ossim_mm_tlb_hk241$ ./os os_1_tlbsz_singleCPU_m1q
Time slot 0
ld_routine
Time slot 1
    Loaded a process at input/proc/s4, PID: 1 PRI0: 4
Time slot 2
    CPU 0: Dispatched process 1
    Loaded a process at input/proc/s3, PID: 2 PRI0: 3
Time slot 3
Time slot 4
    CPU 0: Put process 1 to run queue
    CPU 0: Dispatched process 2
    Loaded a process at input/proc/m1s, PID: 3 PRI0: 2
Time slot 5
Time slot 6
    CPU 0: Put process 2 to run queue
    CPU 0: Dispatched process 3
    TLB_alloc register 0, proc: 3
    TLBMEMPHY_dump
    TLB Cache Start
    Entry 0: Used: 1 Entry pid: 32767 Entry pagenum: 0 Entry framenum: 1
    TLB Cache End
    Loaded a process at input/proc/s2, PID: 4 PRI0: 3
Time slot 7
    TLB_alloc register 1, proc: 3
    Unsuccessful tlb_alloc.
    Loaded a process at input/proc/m0s, PID: 5 PRI0: 3
Time slot 8
    CPU 0: Put process 3 to run queue
    CPU 0: Dispatched process 3
    TLB_free register 0, proc: 3
Time slot 9
    TLB_alloc register 2, proc: 3
    TLBMEMPHY_dump
    TLB Cache Start
    Entry 0: Used: 1 Entry pid: 32767 Entry pagenum: 0 Entry framenum: 1
    TLB Cache End
    Loaded a process at input/proc/p1s, PID: 6 PRI0: 2
Time slot 10
    CPU 0: Put process 3 to run queue
    CPU 0: Dispatched process 6
Time slot 11
    Loaded a process at input/proc/s0, PID: 7 PRI0: 1
Time slot 12
    CPU 0: Put process 6 to run queue
Time slot 13
    CPU 0: Put process 6 to run queue
    CPU 0: Dispatched process 7
Time slot 14
    CPU 0: Put process 7 to run queue
    CPU 0: Dispatched process 7
Time slot 15
    CPU 0: Put process 7 to run queue
    CPU 0: Dispatched process 7
Time slot 16
    CPU 0: Put process 7 to run queue
    CPU 0: Dispatched process 7
    Loaded a process at input/proc/s1, PID: 8 PRI0: 0
Time slot 17
Time slot 18
    CPU 0: Put process 7 to run queue
    CPU 0: Dispatched process 8
    TLB_alloc register 0, proc: 8
    TLBMEMPHY_dump
    TLB Cache Start
    Entry 0: Used: 1 Entry pid: 32767 Entry pagenum: 0 Entry framenum: 3
    TLB Cache End
Time slot 19
Time slot 20
    CPU 0: Put process 8 to run queue
    CPU 0: Dispatched process 8
Time slot 21
Time slot 22
    CPU 0: Put process 8 to run queue
    CPU 0: Dispatched process 8
Time slot 23
Time slot 24
    CPU 0: Put process 8 to run queue
    CPU 0: Dispatched process 8
Time slot 25
    CPU 0: Processed 8 has finished
    CPU 0: Dispatched process 7
Time slot 26
Time slot 27
    CPU 0: Put process 7 to run queue
    CPU 0: Dispatched process 7
Time slot 28
Time slot 29
    CPU 0: Put process 7 to run queue
    CPU 0: Dispatched process 7
Time slot 30
```



```
Time slot 31
  CPU 0: Put process 7 to run queue
  CPU 0: Dispatched process 7
Time slot 32
Time slot 33
  CPU 0: Put process 7 to run queue
  CPU 0: Dispatched process 7
Time slot 34
  CPU 0: Processed 7 has finished
  CPU 0: Dispatched process 3
TLB_free register 2, proc: 3
Time slot 35
  TLB_free register 1, proc: 3
Time slot 36
  CPU 0: Put process 3 to run queue
  CPU 0: Dispatched process 6
Time slot 37
Time slot 38
  CPU 0: Put process 6 to run queue
  CPU 0: Dispatched process 3
Time slot 39
  TLB_free register 0, proc: 3
Time slot 40
  CPU 0: Processed 3 has finished
  CPU 0: Dispatched process 6
Time slot 41
Time slot 42
  CPU 0: Put process 6 to run queue
  CPU 0: Dispatched process 6
Time slot 43
Time slot 44
  CPU 0: Put process 6 to run queue
  CPU 0: Dispatched process 6
Time slot 45
Time slot 46
  CPU 0: Processed 6 has finished
  CPU 0: Dispatched process 2
Time slot 47
Time slot 48
  CPU 0: Put process 2 to run queue
  CPU 0: Dispatched process 4
Time slot 49
Time slot 50
  CPU 0: Put process 4 to run queue
  CPU 0: Dispatched process 5
Time slot 51
  TLB_alloc register 1, proc: 5
  Unsuccessful tlb_alloc.
Time slot 52
  CPU 0: Put process 5 to run queue
  CPU 0: Dispatched process 2
Time slot 53
Time slot 54
  CPU 0: Put process 2 to run queue
  CPU 0: Dispatched process 4
Time slot 55
Time slot 56
  CPU 0: Put process 4 to run queue
  CPU 0: Dispatched process 5
Time slot 57
  TLB_free register 0, proc: 5
  TLB_alloc register 2, proc: 5
  TLBMEMPHY_dump
  TLB Cache Start
  Entry 0:      Used: 1 Entry pid: 32767      Entry pagenum: 0      Entry framenum: 5
  TLB Cache End
Time slot 58
  CPU 0: Put process 5 to run queue
  CPU 0: Dispatched process 2
Time slot 59
Time slot 60
  CPU 0: Put process 2 to run queue
  CPU 0: Dispatched process 4
Time slot 61
Time slot 62
  CPU 0: Put process 4 to run queue
  CPU 0: Dispatched process 5
  TLB_write, proc: 5
  TLB_hit at write region=1 offset=20 value=102
  print_pgtbl: 0 - 512
  00000000: 80000005
  00000004: 80000004
  TLBMEMPHY_dump
  TLB Cache Start
  Entry 0:      Used: 1 Entry pid: 7      Entry pagenum: 0      Entry framenum: 5
  TLB Cache End
Time slot 63
  TLB_write, proc: 5
  TLB_hit at write region=2 offset=1000 value=1
  print_pgtbl: 0 - 512
  00000000: 80000005
  00000004: 80000004
  TLBMEMPHY_dump
  TLB Cache Start
  Entry 0:      Used: 1 Entry pid: 7      Entry pagenum: 0      Entry framenum: 5
  TLB Cache End
Time slot 64
  CPU 0: Put process 5 to run queue
  CPU 0: Dispatched process 2
Time slot 65
Time slot 66
  CPU 0: Put process 2 to run queue
  CPU 0: Dispatched process 4
Time slot 67
Time slot 68
  CPU 0: Put process 4 to run queue
  CPU 0: Dispatched process 5
  TLB_write, proc: 5
  TLB_hit at write region=0 offset=0 value=0
  print_pgtbl: 0 - 512
  00000000: c0000000
  00000004: 80000004
  TLBMEMPHY_dump
  TLB Cache Start
  Entry 0:      Used: 1 Entry pid: 7      Entry pagenum: 0      Entry framenum: 5
  TLB Cache End
Time slot 69
  CPU 0: Processed 5 has finished
  CPU 0: Dispatched process 2
Time slot 70
  CPU 0: Processed 2 has finished
  CPU 0: Dispatched process 4
Time slot 71
Time slot 72
  CPU 0: Put process 4 to run queue
  CPU 0: Dispatched process 4
Time slot 73
Time slot 74
  CPU 0: Processed 4 has finished
  CPU 0: Dispatched process 1
Time slot 75
Time slot 76
  CPU 0: Put process 1 to run queue
  CPU 0: Dispatched process 1
Time slot 77
Time slot 78
  CPU 0: Put process 1 to run queue
  CPU 0: Dispatched process 1
Time slot 79
  CPU 0: Processed 1 has finished
  CPU 0 stopped
```

Figure 12: Result of Direct Mapped TLB



4.4.3 Output of Fully Associative TLB

```
antwerp2004@antwerp2004-VirtualBox:~/osin_mm_tlb_hk240$ ./os_os_1_tlbsz_singleCPU_mfq
Time slot 0
ld routine
Time slot 1
Loaded a process at input/proc/s4, PID: 1 PRI0: 4
Time slot 2
CPU 0: Dispatched process 1
Loaded a process at input/proc/s3, PID: 2 PRI0: 3
Time slot 3
Time slot 4
CPU 0: Put process 1 to run queue
CPU 0: Dispatched process 2
Loaded a process at input/proc/mis, PID: 3 PRI0: 2
Time slot 5
Time slot 6
CPU 0: Put process 2 to run queue
CPU 0: Dispatched process 3
TLB_alloc register 0, proc: 3
TLBMEMPHY_dump
TLB Cache Start
Entry 0: Used: 1 Entry pid: 3 Entry pagenum: 0 Entry framenum: 1
TLB Cache End
Loaded a process at input/proc/s2, PID: 4 PRI0: 3
Time slot 7
TLB_alloc register 1, proc: 3
Unsuccessful tlb_alloc.
Loaded a process at input/proc/m0s, PID: 5 PRI0: 3
Time slot 8
CPU 0: Put process 3 to run queue
CPU 0: Dispatched process 3
TLB_free register 0, proc: 3
Time slot 9
TLB_alloc register 2, proc: 3
TLBMEMPHY_dump
TLB Cache Start
Entry 1: Used: 1 Entry pid: 11 Entry pagenum: 0 Entry framenum: 1
TLB Cache End
Loaded a process at input/proc/pls, PID: 6 PRI0: 2
Time slot 10
CPU 0: Put process 3 to run queue
CPU 0: Dispatched process 6
Time slot 11
Loaded a process at input/proc/s0, PID: 7 PRI0: 1
Time slot 12
CPU 0: Put process 6 to run queue
Time slot 12
CPU 0: Put process 6 to run queue
CPU 0: Dispatched process 7
Time slot 13
Time slot 14
CPU 0: Put process 7 to run queue
CPU 0: Dispatched process 7
Time slot 15
Time slot 16
CPU 0: Put process 7 to run queue
CPU 0: Dispatched process 7
Loaded a process at input/proc/s1, PID: 8 PRI0: 0
Time slot 17
Time slot 18
CPU 0: Put process 7 to run queue
CPU 0: Dispatched process 8
TLB_alloc register 0, proc: 8
TLBMEMPHY_dump
TLB Cache Start
Entry 1: Used: 1 Entry pid: 19 Entry pagenum: 0 Entry framenum: 1
Entry 2: Used: 1 Entry pid: 27 Entry pagenum: 0 Entry framenum: 3
TLB Cache End
Time slot 19
Time slot 20
CPU 0: Put process 8 to run queue
CPU 0: Dispatched process 8
Time slot 21
Time slot 22
CPU 0: Put process 8 to run queue
CPU 0: Dispatched process 8
Time slot 23
Time slot 24
CPU 0: Put process 8 to run queue
CPU 0: Dispatched process 8
Time slot 25
CPU 0: Processed 8 has finished
CPU 0: Dispatched process 7
Time slot 26
Time slot 27
CPU 0: Put process 7 to run queue
CPU 0: Dispatched process 7
Time slot 28
Time slot 29
CPU 0: Put process 7 to run queue
CPU 0: Dispatched process 7
Time slot 30
Time slot 31
CPU 0: Put process 7 to run queue
CPU 0: Dispatched process 7
Time slot 32
Time slot 33
CPU 0: Put process 7 to run queue
CPU 0: Dispatched process 7
Time slot 34
CPU 0: Processed 7 has finished
CPU 0: Dispatched process 3
TLB_free register 2, proc: 3
Time slot 35
TLB_free register 1, proc: 3
Time slot 36
CPU 0: Put process 3 to run queue
CPU 0: Dispatched process 6
Time slot 37
Time slot 38
CPU 0: Put process 6 to run queue
CPU 0: Dispatched process 3
TLB_free register 0, proc: 3
Time slot 39
TLB_free register 0, proc: 3
Time slot 40
CPU 0: Processed 3 has finished
CPU 0: Dispatched process 6
Time slot 41
Time slot 42
CPU 0: Put process 6 to run queue
CPU 0: Dispatched process 6
Time slot 43
Time slot 44
CPU 0: Put process 6 to run queue
CPU 0: Dispatched process 6
Time slot 45
Time slot 46
CPU 0: Processed 6 has finished
CPU 0: Dispatched process 2
Time slot 47
Time slot 48
CPU 0: Put process 2 to run queue
CPU 0: Dispatched process 4
Time slot 49
Time slot 50
Time slot 50
CPU 0: Put process 4 to run queue
CPU 0: Dispatched process 5
TLB_alloc register 0, proc: 5
TLBMEMPHY_dump
TLB Cache Start
Entry 2: Used: 1 Entry pid: 27 Entry pagenum: 0 Entry framenum: 3
Entry 3: Used: 1 Entry pid: 31 Entry pagenum: 0 Entry framenum: 5
TLB Cache End
Time slot 51
TLB_alloc register 1, proc: 5
Unsuccessful tlb_alloc.
Time slot 52
CPU 0: Put process 5 to run queue
CPU 0: Dispatched process 2
Time slot 53
Time slot 54
CPU 0: Put process 2 to run queue
CPU 0: Dispatched process 4
Time slot 55
Time slot 56
CPU 0: Put process 4 to run queue
CPU 0: Dispatched process 5
TLB_free register 0, proc: 5
Time slot 57
TLB_alloc register 2, proc: 5
TLBMEMPHY_dump
TLB Cache Start
Entry 2: Used: 1 Entry pid: 44 Entry pagenum: 0 Entry framenum: 3
Entry 4: Used: 1 Entry pid: 45 Entry pagenum: 0 Entry framenum: 5
TLB Cache End
Time slot 58
CPU 0: Put process 5 to run queue
CPU 0: Dispatched process 2
Time slot 59
Time slot 60
CPU 0: Put process 2 to run queue
CPU 0: Dispatched process 4
Time slot 61
Time slot 62
CPU 0: Put process 4 to run queue
CPU 0: Dispatched process 5
TLB_write, proc: 5
TLB hit at write region=1 offset=20 value=102
print_pgtbl: 0 - 512
```

```

Time slot 62
CPU 0: Put process 4 to run queue
CPU 0: Dispatched process 5
TLB_write, proc: 5
TLB hit at write region=1 offset=20 value=102
print_pgbl: 0 - 512
00000000: 80000005
00000004: 80000004
TLBMEMPHY_dump
TLB Cache Start
Entry 2:      Used: 1 Entry pid: 10  Entry pagenum: 0      Entry framenum: 3
Entry 4:      Used: 1 Entry pid: 15  Entry pagenum: 0      Entry framenum: 5
TLB Cache End
Time slot 63
TLB_write, proc: 5
TLB hit at write region=2 offset=1000 value=1
print_pgbl: 0 - 512
00000000: 80000005
00000004: 80000004
TLBMEMPHY_dump
TLB Cache Start
Entry 2:      Used: 1 Entry pid: 10  Entry pagenum: 0      Entry framenum: 3
Entry 4:      Used: 1 Entry pid: 15  Entry pagenum: 0      Entry framenum: 5
TLB Cache End
Time slot 64
CPU 0: Put process 5 to run queue
CPU 0: Dispatched process 2
Time slot 65
Time slot 66
CPU 0: Put process 2 to run queue
CPU 0: Dispatched process 4
Time slot 67
Time slot 68
CPU 0: Put process 4 to run queue
CPU 0: Dispatched process 5
TLB_write, proc: 5
TLB hit at write region=0 offset=0 value=0
print_pgbl: 0 - 512
00000000: c0000000
00000004: 80000004
TLBMEMPHY_dump
TLB Cache Start
Entry 2:      Used: 1 Entry pid: 10  Entry pagenum: 0      Entry framenum: 3
Entry 4:      Used: 1 Entry pid: 15  Entry pagenum: 0      Entry framenum: 5
TLB Cache End

Time slot 69
CPU 0: Processed 5 has finished
CPU 0: Dispatched process 2
Time slot 70
CPU 0: Processed 2 has finished
CPU 0: Dispatched process 4
Time slot 71
Time slot 72
CPU 0: Put process 4 to run queue
CPU 0: Dispatched process 4
Time slot 73
Time slot 74
CPU 0: Processed 4 has finished
CPU 0: Dispatched process 1
Time slot 75
Time slot 76
CPU 0: Put process 1 to run queue
CPU 0: Dispatched process 1
Time slot 77
Time slot 78
CPU 0: Put process 1 to run queue
CPU 0: Dispatched process 1
Time slot 79
CPU 0: Processed 1 has finished
CPU 0 stopped

```

Figure 15: Result of Fully Associative TLB

* Explain Output: The first entry unfilled in the TLB is automatically selected.

- **Time slot 6:** TLB alloc register 0, process 3 → Entry 0 is filled, mark 1 as used.
- **Time slot 7:** TLB alloc register 1, process 3 unsuccessfully.
- **Time slot 8:** TLB free register 0, process 3 → Entry 0 is marked 0 as unused.
- **Time slot 9:** TLB alloc register 2, process 3 → Entry 1 is filled, mark as used.
- **Time slot 18:** TLB alloc register 0, process 8 → Entry 2 is filled, mark as used → Entry 1, 2 in the TLB.
- **Time slot 34:** TLB free register 2, process 3 → Entry 1 is marked as unused → Only Entry 2 in TLB.
- **Time slot 35:** TLB free register 1, process 3 (Not filled yet).
- **Time slot 38:** TLB free register 0, process 3 (Currently marked as unused).
- **Time slot 39:** TLB free register 0, process 3 (Currently marked as unused).
- **Time slot 50:** TLB alloc register 0, process 5 → Entry 3 is filled, mark as used → Entry 2, 3 in the TLB.
- **Time slot 51:** TLB alloc register 1, process 5 unsuccessfully.

- **Time slot 56:** TLB free register 0, process 5 → Entry 3 is marked as unused → Only Entry 2 in TLB.
- **Time slot 57:** TLB alloc register 2, process 5 → Entry 4 is filled, mark as used → Entry 2, 4 in the TLB.
- **Time slot 62:** TLB write at process 5 → Hit (Entry 4).
- **Time slot 63:** TLB write at process 5 → Hit (Entry 4).
- **Time slot 68:** TLB write at process 5 → Hit (Entry 4).

4.5 Question

Question 5: 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:

* If a multi-core system allows each CPU core to operate in a different context with its own MMU and TLB, several implications arise:

1. **Isolation:** Each CPU core can execute its own set of processes or threads independently without interference from other cores. This isolation ensures that tasks running on one core do not affect the operation of tasks running on other cores.
2. **Concurrency:** The system can achieve higher levels of concurrency as each CPU core can handle its own memory management tasks, including address translation and caching, concurrently with other cores. This concurrency leads to better utilization of system resources and improved overall system performance.
3. **Reduced Contention:** With separate MMUs and TLBs for each core, contention for memory management resources is minimized. Each core can perform address translation and caching operations without waiting for access to shared resources, such as a centralized TLB or page table structures.
4. **Scalability:** The system can scale efficiently with the addition of more CPU cores since each core operates independently with its own MMU and TLB. This scalability allows for the effective utilization of hardware resources and supports the execution of a larger number of concurrent tasks.

5. **Complexity:** Managing memory consistency and coherence across multiple cores with their own MMUs and TLBs introduces complexity. Mechanisms for ensuring TLB coherence and maintaining consistency between local TLBs and shared memory structures are necessary to prevent data corruption and ensure correct program execution.
6. **Synchronization Overhead:** Although each core operates independently, coordination may still be required between cores for certain operations that involve shared resources or data. Synchronization mechanisms, such as locks or atomic operations, are necessary to ensure data integrity and consistency in multi-core environments.

In summary, a multi-core system with individual CPU cores having their own MMUs and TLBs offers benefits such as improved concurrency, reduced contention, and scalability. However, it also introduces challenges related to managing memory consistency and synchronization across multiple cores.

* The introduction of 2-level TLBs (Translation Lookaside Buffers) in modern CPUs has several impacts on memory management and translation schemes:

1. **Improved TLB Efficiency:** With a 2-level TLB architecture, the TLB is divided into two levels: a small, fast, and typically fully associative L1 TLB (Level 1 TLB), and a larger, slower, and typically set-associative or even fully associative L2 TLB (Level 2 TLB). This architecture allows for more efficient use of TLB space by prioritizing frequently accessed translations in the L1 TLB while accommodating a larger number of translations in the L2 TLB.
2. **Reduced TLB Miss Rate:** By having a hierarchical TLB structure, the likeliness of TLB misses is reduced compared to a single-level TLB design. Frequently accessed translations are more likely to be found in the faster L1 TLB, thereby reducing the frequency of accesses to the slower main memory or page tables.
3. **Enhanced TLB Management:** The use of a 2-level TLB enables more sophisticated TLB management strategies. For example, the operating system can prioritize certain translations to be stored in the L1 TLB based on access patterns or criticality, while less frequently accessed translations are relegated to the L2 TLB. This dynamic management can help optimize TLB utilization and improve overall system performance.
4. **Complexity:** The introduction of a 2-level TLB adds complexity to the memory management subsystem. Hardware and software must coordinate to manage TLB entries across both levels efficiently. Additionally, managing TLB coherence between the L1 and L2 TLBs requires careful design to ensure consistency and correctness in translation results.

5. **Impact on Translation Overhead:** While 2-level TLBs can reduce TLB miss rates and improve overall performance, the additional level of translation introduces overhead in terms of TLB lookup latency and complexity. Accessing translations stored in the L2 TLB may take longer than accessing those in the L1 TLB, impacting memory access latency.
6. **Scalability:** The hierarchical TLB architecture provides scalability benefits by allowing for the accommodation of a larger number of translations without significantly increasing access times. As memory demands grow, the L2 TLB can be expanded to support additional translations while maintaining efficient access times.

In summary, the adoption of 2-level TLBs in modern CPUs brings benefits such as improved TLB efficiency, reduced TLB miss rates, and enhanced TLB management capabilities. However, it also introduces complexity and overhead, necessitating careful design considerations in memory management and translation schemes.

5 Synchronization

5.1 Overview

In this assignment, mutex locks are used for synchronization to ensure that only one thread can access critical sections of code at a time. This is important because the critical sections involve shared resources, such as data structures or memory locations, that can be accessed or modified by multiple threads concurrently. Without proper synchronization, concurrent access to these shared resources could lead to race conditions, where the outcome of the program becomes dependent on the timing of thread execution, resulting in unpredictable behavior and potential data corruption.

By using mutex locks, threads acquire exclusive access to critical sections, preventing other threads from accessing them until the lock is released. This ensures that only one thread can execute the critical section at any given time, maintaining the integrity of shared resources and preventing race conditions.

Mutex locks should be used wherever shared resources are accessed or modified concurrently to prevent race conditions and ensure thread safety.

5.2 Implementation

As we have displayed the code throughout the report, we have used mutex lock for synchronization in those following functions:

- Scheduler

In file `sched.c`:

1. `get_mlq_proc()`

2. `put_mlq_proc()`

```
1 void put_mlq_proc(struct pcb_t *proc)
2 {
3     pthread_mutex_lock(&queue_lock);
4     enqueue(&mlq_ready_queue[proc->prio], proc);
5     pthread_mutex_unlock(&queue_lock);
6 }
```

3. `add_mlq_proc()`

```
1 void add_mlq_proc(struct pcb_t *proc)
2 {
```

```
3   pthread_mutex_lock(&queue_lock);
4   enqueue(&mlq_ready_queue[proc->prio], proc);
5   pthread_mutex_unlock(&queue_lock);
6 }
```

• Paging-based Memory Management

In file mm-memphy.c:

1. MEMPHY_get_freefp()

```
1  int MEMPHY_get_freefp(struct memphy_struct *mp, int *retfpn)
2  {
3      pthread_mutex_lock(&mem_lock);
4
5      struct framephy_struct *fp = mp->free_fp_list;
6
7      if (fp == NULL)
8          return -1;
9
10     *retfpn = fp->fpn;
11     mp->free_fp_list = fp->fp_next;
12
13     /* MEMPHY is iteratively used up until its exhausted
14      * No garbage collector acting then it not been released
15      */
16     free(fp);
17
18     pthread_mutex_unlock(&mem_lock);
19     return 0;
20 }
```

2. MEMPHY_put_freefp()

```
1  int MEMPHY_put_freefp(struct memphy_struct *mp, int fpn)
2  {
3      pthread_mutex_lock(&mem_lock);
4
5      struct framephy_struct *fp = mp->free_fp_list;
6      struct framephy_struct *newnode = malloc(sizeof(struct
           framephy_struct));
```

```
7
8  /* Create new node with value fpn */
9  newnode->fpn = fpn;
10 newnode->fp_next = fp;
11 mp->free_fp_list = newnode;
12
13 pthread_mutex_unlock(&mem_lock);
14 return 0;
15 }
```

- TLB Memory Management

In file `cpu-tlb.c`

1. `tlb_change_all_page_tables_of()`
2. `tlb_flush_tlb_of()`
3. `tlbfree_data()`

In file `cpu-tlbcache.c`

1. `tlb_cache_read()`
2. `tlb_cache_write()`
3. `TLBMEMPHY_dump()`

5.3 Question

Question 6: 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. *Note: You need to run two versions of your simple OS: the program with/without synchronization, then observe their performance based on demo results and explain their differences.*

Answer: If synchronization is not handled properly in a simple operating system, it can lead to various issues such as race conditions, data corruption, and inconsistent behavior. Let's illustrate this problem with an example:

Consider a scenario in our simple operating system where two processes, A and B, both attempt to access and modify a shared resource (e.g., a shared variable or a shared data structure) concurrently without proper synchronization.

Process A and Process B are both trying to increment a shared counter variable `count`:

```
1 // Shared variable
2 int count = 0;
```

```
3
4 // Process A
5 void process_A() {
6     // Increment count
7     count++;
8 }
9
10 // Process B
11 void process_B() {
12     // Increment count
13     count++;
14 }
```

Now, let's imagine that Process A and Process B are executed concurrently by the operating system. Without proper synchronization mechanisms such as mutex locks, the following sequence of events might occur:

1. Process A reads the value of `count`, which is 0.
2. Process B reads the value of `count`, which is still 0.
3. Process A increments `count` to 1.
4. Process B also increments `count` to 1, unaware that it has been modified by Process A.
5. Both processes complete their execution, and `count` ends up being 1 instead of 2.

In this example, the expected behavior was for the shared variable `count` to be incremented by 2 (once by Process A and once by Process B). However, due to the lack of synchronization, the final value of `count` is incorrect, leading to data inconsistency.

This inconsistency in shared data can cause unpredictable behavior in the operating system, leading to incorrect results, program crashes, or other issues. Therefore, proper synchronization mechanisms such as mutex locks are essential to ensure the integrity of shared resources and prevent such problems.

* Now we will compare the result with/without synchronization. First we simply delete the locks in Paging-based Memory Management (2 function `MEMPHY_get_freefp()` and `MEMPHY_put_freefp()`)

- **Input file: `os_1_mlq_paging_small_1K.c`**

```
1 2 4 8
2 2048 16777216 0 0 0
3 1 p0s 130
4 2 s3 39
```



```
5 4 m1s 15
6 6 s2 120
7 7 m0s 120
8 9 p1s 15
9 11 s0 38
10 16 s1 0
```

- With synchronization (Mutex lock)

```
antwerp2004@antwerp2004-VirtualBox:~/osath_nm_tlb_hk241$ ./os_os_1_mld_paging_small_1K
Time slot 0
ld_routine
Time slot 1
  Loaded a process at input/proc/p0s, PID: 1 PRI0: 130
Time slot 2
  CPU 0: Dispatched process 1
  Loaded a process at input/proc/s3, PID: 2 PRI0: 39
Time slot 3
  CPU 2: Dispatched process 2
Time slot 4
  CPU 0: Put process 1 to run queue
  CPU 0: Dispatched process 1
  Loaded a process at input/proc/m1s, PID: 3 PRI0: 15
Time slot 5
  CPU 1: Dispatched process 3
  CPU 2: Put process 2 to run queue
  CPU 2: Dispatched process 2
Time slot 6
  CPU 0: Put process 1 to run queue
  CPU 0: Dispatched process 1
  Loaded a process at input/proc/s2, PID: 4 PRI0: 120
Time slot 7
  CPU 3: Dispatched process 4
  CPU 2: Put process 2 to run queue
  CPU 2: Dispatched process 2
  CPU 1: Put process 3 to run queue
  CPU 1: Dispatched process 3
write region=1 offset=20 value=100
print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000000
  Loaded a process at input/proc/m0s, PID: 5 PRI0: 120
Time slot 8
  CPU 0: Put process 1 to run queue
  CPU 0: Dispatched process 5
Time slot 9
  CPU 3: Put process 4 to run queue
  CPU 3: Dispatched process 4
  CPU 2: Put process 2 to run queue
  CPU 2: Dispatched process 2
  CPU 1: Put process 3 to run queue
  CPU 1: Dispatched process 3
  Loaded a process at input/proc/p1s, PID: 6 PRI0: 15
```

```
Time slot 10
  CPU 0: Put process 5 to run queue
  CPU 0: Dispatched process 6
Time slot 11
  CPU 3: Put process 4 to run queue
  CPU 3: Dispatched process 5
  CPU 2: Put process 2 to run queue
  CPU 2: Dispatched process 2
  CPU 1: Put process 3 to run queue
  CPU 1: Dispatched process 3
  Loaded a process at input/proc/s0, PID: 7 PRI0: 38
Time slot 12
  CPU 0: Put process 6 to run queue
  CPU 0: Dispatched process 6
Time slot 13
  CPU 2: Put process 2 to run queue
  CPU 2: Dispatched process 7
  CPU 3: Put process 5 to run queue
  CPU 3: Dispatched process 2
  CPU 1: Processed 3 has finished
  CPU 1: Dispatched process 4
Time slot 14
  CPU 3: Processed 2 has finished
  CPU 3: Dispatched process 5
write region=1 offset=20 value=102
print_pgtbl: 0 - 512
00000000: 80000005
00000004: 80000004
  CPU 0: Put process 6 to run queue
  CPU 0: Dispatched process 6
Time slot 15
  CPU 2: Put process 7 to run queue
  CPU 2: Dispatched process 7
write region=2 offset=1000 value=1
print_pgtbl: 0 - 512
00000000: 80000005
00000004: 80000004
  CPU 1: Put process 4 to run queue
  CPU 1: Dispatched process 4
Time slot 16
  CPU 3: Put process 5 to run queue
  CPU 3: Dispatched process 5
write region=0 offset=0 value=0
print_pgtbl: 0 - 512
```




```
Time slot 16
CPU 3: Put process 5 to run queue
CPU 3: Dispatched process 5
write region=0 offset=0 value=0
print_pgtbl: 0 - 512
00000000: c0000000
00000004: 80000004
CPU 0: Put process 6 to run queue
CPU 0: Dispatched process 6
Loaded a process at input/proc/s1, PID: 8 PRIO: 0
Time slot 17
CPU 2: Put process 7 to run queue
CPU 2: Dispatched process 8
CPU 3: Processed 5 has finished
CPU 3: Dispatched process 7
CPU 1: Put process 4 to run queue
CPU 1: Dispatched process 4
Time slot 18
CPU 0: Put process 6 to run queue
CPU 0: Dispatched process 6
Time slot 19
CPU 2: Put process 8 to run queue
CPU 2: Dispatched process 8
CPU 3: Put process 7 to run queue
CPU 3: Dispatched process 7
CPU 1: Put process 4 to run queue
CPU 1: Dispatched process 4
Time slot 20
CPU 0: Processed 6 has finished
CPU 0: Dispatched process 1
read region=1 offset=20 value=100
print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000000
Time slot 21
CPU 2: Put process 8 to run queue
CPU 2: Dispatched process 8
CPU 3: Put process 7 to run queue
CPU 3: Dispatched process 7
write region=3 offset=20 value=103
print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000000
CPU 1: Processed 4 has finished
CPU 1 stopped
```

```
Time slot 22
CPU 0: Put process 1 to run queue
CPU 0: Dispatched process 1
read region=3 offset=20 value=103
print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000000
Time slot 23
CPU 2: Put process 8 to run queue
CPU 2: Dispatched process 8
CPU 3: Put process 7 to run queue
CPU 3: Dispatched process 7
Time slot 24
CPU 2: Processed 8 has finished
CPU 2 stopped
CPU 0: Processed 1 has finished
CPU 0 stopped
Time slot 25
CPU 3: Put process 7 to run queue
CPU 3: Dispatched process 7
Time slot 26
Time slot 27
CPU 3: Put process 7 to run queue
CPU 3: Dispatched process 7
Time slot 28
CPU 3: Processed 7 has finished
CPU 3 stopped
```

Figure 17: Input of code with synchronization

- Without synchronization

```
antwerp2004@antwerp2004-VirtualBox:~/os1_mq_pagination_small_1k
ld_routine
Time slot 1
Loaded a process at input/proc/p0s, PID: 1 PRIO: 130
Time slot 2
CPU 2: Dispatched process 1
Loaded a process at input/proc/s3, PID: 2 PRIO: 39
Time slot 3
CPU 3: Dispatched process 2
Time slot 4
CPU 2: Put process 1 to run queue
CPU 2: Dispatched process 1
Loaded a process at input/proc/mis, PID: 3 PRIO: 15
Time slot 5
CPU 0: Dispatched process 3
CPU 3: Put process 2 to run queue
CPU 3: Dispatched process 2
Time slot 6
CPU 2: Put process 1 to run queue
CPU 2: Dispatched process 1
Loaded a process at input/proc/s2, PID: 4 PRIO: 120
Time slot 7
CPU 1: Dispatched process 4
CPU 3: Put process 2 to run queue
CPU 3: Dispatched process 2
CPU 0: Put process 3 to run queue
CPU 0: Dispatched process 3
write region=1 offset=20 value=100
print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000000
Loaded a process at input/proc/m0s, PID: 5 PRIO: 120
Time slot 8
CPU 2: Put process 1 to run queue
CPU 2: Dispatched process 5
Time slot 9
CPU 1: Put process 4 to run queue
CPU 1: Dispatched process 4
CPU 3: Put process 2 to run queue
CPU 3: Dispatched process 2
CPU 0: Put process 3 to run queue
CPU 0: Dispatched process 3
Loaded a process at input/proc/pis, PID: 6 PRIO: 15
```

```
Time slot 10
CPU 2: Put process 5 to run queue
CPU 2: Dispatched process 6
Time slot 11
CPU 3: Put process 2 to run queue
CPU 3: Dispatched process 2
CPU 0: Put process 3 to run queue
CPU 0: Dispatched process 3
CPU 1: Put process 4 to run queue
CPU 1: Dispatched process 5
Loaded a process at input/proc/s0, PID: 7 PRIO: 38
Time slot 12
CPU 2: Put process 6 to run queue
CPU 2: Dispatched process 6
Time slot 13
CPU 3: Put process 2 to run queue
CPU 3: Dispatched process 7
CPU 0: Processed 3 has finished
CPU 0: Dispatched process 2
CPU 1: Put process 5 to run queue
CPU 1: Dispatched process 4
Time slot 14
CPU 2: Put process 6 to run queue
CPU 2: Dispatched process 6
CPU 0: Processed 2 has finished
CPU 0: Dispatched process 5
write region=1 offset=20 value=102
print_pgtbl: 0 - 512
00000000: 80000005
00000004: 80000004
Time slot 15
CPU 3: Put process 7 to run queue
CPU 3: Dispatched process 7
CPU 1: Put process 4 to run queue
CPU 1: Dispatched process 4
write region=2 offset=1000 value=1
print_pgtbl: 0 - 512
00000000: 80000005
00000004: 80000004
Time slot 16
CPU 2: Put process 6 to run queue
CPU 2: Dispatched process 6
CPU 0: Put process 5 to run queue
CPU 0: Dispatched process 5
write region=0 offset=0 value=0
```

```

Time slot 16
CPU 2: Put process 6 to run queue
CPU 2: Dispatched process 6
CPU 0: Put process 5 to run queue
CPU 0: Dispatched process 5
write region=0 offset=0 value=0
print_pgtbl: 0 - 512
00000000: c0000000
00000004: 80000004
Loaded a process at input/proc/s1, PID: 8 PRIO: 0
Time slot 17
CPU 3: Put process 7 to run queue
CPU 3: Dispatched process 8
CPU 1: Put process 4 to run queue
CPU 1: Dispatched process 7
CPU 0: Processed 5 has finished
CPU 0: Dispatched process 4
Time slot 18
CPU 2: Put process 6 to run queue
CPU 2: Dispatched process 6
Time slot 19
CPU 1: Put process 7 to run queue
CPU 1: Dispatched process 7
CPU 3: Put process 8 to run queue
CPU 3: Dispatched process 8
CPU 0: Put process 4 to run queue
CPU 0: Dispatched process 4
Time slot 20
CPU 2: Processed 6 has finished
CPU 2: Dispatched process 1
read region=1 offset=20 value=100
print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000000
Time slot 21
CPU 1: Put process 7 to run queue
CPU 1: Dispatched process 7
write region=3 offset=20 value=103
print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000000
CPU 3: Put process 8 to run queue
CPU 3: Dispatched process 8
CPU 0: Processed 4 has finished
CPU 0 stopped

```

```

Time slot 22
CPU 2: Put process 1 to run queue
CPU 2: Dispatched process 1
read region=3 offset=20 value=103
print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000000
Time slot 23
CPU 1: Put process 7 to run queue
CPU 1: Dispatched process 7
CPU 3: Put process 8 to run queue
CPU 3: Dispatched process 8
Time slot 24
CPU 3: Processed 8 has finished
CPU 3 stopped
CPU 2: Processed 1 has finished
CPU 2 stopped
Time slot 25
CPU 1: Put process 7 to run queue
CPU 1: Dispatched process 7
Time slot 26
Time slot 27
CPU 1: Put process 7 to run queue
CPU 1: Dispatched process 7
Time slot 28
CPU 1: Processed 7 has finished
CPU 1 stopped

```

Figure 19: Input of code without synchronization

There's a huge difference between the output of 2 implementations with/without synchronization, as a result of the lack of mutex locks leading to inconsistency in shared data, which cause unpredictable behavior in the operating system, leading to incorrect results, program crashes, and other issues.