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

OPERATING SYSTEM (CO2018)

Assignment (Semester 202, Duration: 03 weeks)

Simple Operating System

Advisor: Ms. Lê Thanh Vân
Students:

Nguyễn Hoàng (CC02)	1952255
Đỗ Đăng Khoa (CC02)	1952295
Lê Minh Đăng (CC02)	1952041

HO CHI MINH CITY, MAY 2021

Contents

Member list & Workload

No.	Full name	Student ID	Problems	Percentage of work
1	Lê Minh Đăng	1952041	Synchronization	100%
2	Nguyễn Hoàng	1952255	Scheduler	100%
3	Dỗ Đăng Khoa	1952295	Memory	100%

1 Scheduling

1.1 Priority feedback queue versus the world

The Priority Feedback Queue (PFQ) takes after some designing concept from the Multilevel Feedback Queue, i.e. using more than one queue to schedule process. The **time slot** — based on the Round Robins with Quantum time — is used to schedule the process.

The PFQ uses two queues with the following functions

- `ready_queue`: Contains all the processes and their priority to be executed. It will run before `run_queue`. When the CPU switches to a process, the dispatcher will take one from `ready_queue`.
- `run_queue`: Contains the processes which have been executed but have yet to finish. All the processes from this queue get to be executed only if the `ready_queue` is empty.
- The procedure that we use to implement `enqueue()` operation will based on the concept of basic priority queue

1.2 Advantages of Priority feedback queue

The priority feedback queue holds several advantages over other scheduling algorithms.

First and foremost, it implements a mechanism where the relative importance of each process can be defined, meaning that we can have certain processes run before the others.

Let's talk about the advantages of PFQ first:

- PFQ will ensure that all processes will be executed. In other words, it solves the starvation problem by respecting the order of the processes, ensuring all processes are executed.
- The algorithm uses a time slot like the Round-Robin algorithm with a quantum time for each process, thus allowing us to tune the quantum time to minimize average response and turnaround time.

Other scheduling algorithms pose downsides in comparison with priority feedback queue as follows

- First come first served Here, the earlier process will be allocated the CPU first, other processes have to wait until the current process has finished its execution. For example, suppose the first process has a very long burst time, and other processes have less burst time, then the later processes will have to wait longer, this will result in more average waiting time, also known as Convey effect.
- Shortest job first, Shortest runtime first Here, the process with shorter burst time is executed first, meaning that processes with long burst time will be pushed further back in the queue, and potentially never executed. This is known as starvation. In the case that any process always finishes before another one arrives, SJF/SRTF becomes FCFS.
- Round Robin The CPU cycles through all processes in the queue for the amount of time designated by the quantum time or if the process finishes before the

quantum time runs out. Choosing an optimal quantum is hard: too short and the overhead for context switching outweighs the runtime, too long and it becomes FCFS.

- Multilevel This is a combination of multiple kinds of scheduling algorithms. Processes at the lowest level might suffer from starvation if the implementation is not thoroughly considered.

1.3 Gantt chart of this assignment's scheduler

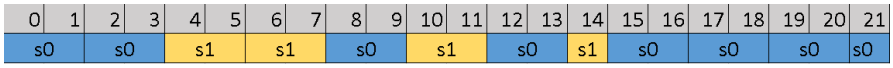
Here, we will draw 2 Gantt diagrams that represent the workflow covering the following tests: `sched_0` and `sched_1`. The implemented scheduling algorithm is the aforementioned priority feedback queue in the Round-Robin style, with a time quantum of 2 time units for both tests.

- `sched_0`

The processes of `sched_0` are summarized in the table below:

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

We obtain the following Gantt diagram that shows how the CPU chooses which process to run, assuming that the loader routine runs ahead of the CPU routine.



Hình 1: Gantt diagram of `sched_0`

The simulated OS should print something akin to this proposed schedule. Here is what `sched_0` returned. Notice that the process with PID 1 is s0 and the one with PID 2 is s1.

```
------ SCHEDULING TEST 0
----------------------------------------
./os sched_0
Time slot   0
        Loaded a process at input/proc/s0, PID: 1
Time slot   1
        CPU 0: Dispatched process  1
Time slot   2
Time slot   3
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   4
        Loaded a process at input/proc/s1, PID: 2
Time slot   5
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
```

```
Time slot   6
Time slot   7
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot   8
Time slot   9
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  1
Time slot  10
Time slot  11
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
Time slot  12
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  1 to run queue
        CPU 0: Dispatched process  2
Time slot  16
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  1
Time slot  17
Time slot  18
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  19
Time slot  20
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  21
Time slot  22
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  23
        CPU 0: Processed  1 has finished
        CPU 0 stopped

MEMORY CONTENT:
NOTE: Read file output/sched_0 to verify your result
```

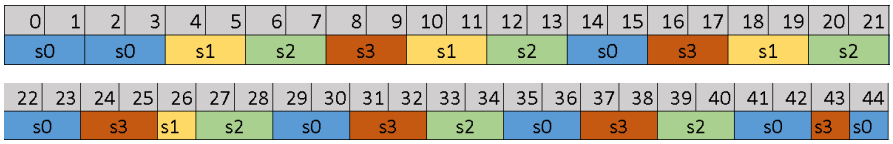
As we can see, after a process is loaded, it is not dispatched immediately. This can be attributed to the 2 threads `loader_routine` and `cpu_routine` not starting in any given order. The overall result should still resemble the hypothesized schedule.

- `sched_1`

Similarly, the processes of `sched_1` are summarized in the table below:

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

We expect the processes to be run as follows



Hình 2: Gantt diagram of `sched_1`

Here is how the simulated OS executed. Note that the process with PID 1 is s0, the one with PID 2 is s1, the one with PID 3 is s2 and the remaining one is s4.

```
NOTE: Read file output/sched_0 to verify your result
------ SCHEDULING TEST 1
-----------------------------------------
./os sched_1
Time slot   0
        Loaded a process at input/proc/s0, PID: 1
Time slot   1
        CPU 0: Dispatched process  1
Time slot   2
Time slot   3
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   4
        Loaded a process at input/proc/s1, PID: 2
Time slot   5
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
Time slot   6
        Loaded a process at input/proc/s2, PID: 3
Time slot   7
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
        Loaded a process at input/proc/s3, PID: 4
Time slot   8
Time slot   9
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
Time slot  10
Time slot  11
        CPU 0: Put process  4 to run queue
```

```
        CPU 0: Dispatched process  2
Time slot  12
Time slot  13
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
Time slot  14
Time slot  15
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
Time slot  16
Time slot  17
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
Time slot  18
Time slot  19
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  2
Time slot  20
Time slot  21
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
Time slot  22
Time slot  23
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
Time slot  24
Time slot  25
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
Time slot  26
Time slot  27
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  2
Time slot  28
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  3
Time slot  29
Time slot  30
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
Time slot  31
Time slot  32
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
Time slot  33
Time slot  34
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
Time slot  35
Time slot  36
```

```
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
Time slot  37
Time slot  38
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
Time slot  39
Time slot  40
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
Time slot  41
Time slot  42
        CPU 0: Processed  3 has finished
        CPU 0: Dispatched process  1
Time slot  43
Time slot  44
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
Time slot  45
        CPU 0: Processed  4 has finished
        CPU 0: Dispatched process  1
Time slot  46
        CPU 0: Processed  1 has finished
        CPU 0 stopped

MEMORY CONTENT:
NOTE: Read file output/sched_1 to verify your result
```

Once again, we can see that there are some minor differences between the observed and expected result. Aside from that, the observed result should still represent the overall structure and scheduling order of the expected result.

1.4 Code implementation for Scheduler:

1.4.1 queue.c

We need to implement 2 functions, `enqueue()` and `dequeue()`.

`enqueue()` adds a new process to a specified queue. While it is implemented using array, we will treat it as a queue, putting the processes with higher priority (smaller priority index) near the head of the queue.

```
1  void enqueue(struct queue_t *q, struct pcb_t *proc)
2  {
3    /* Put a new process to queue [q] */
4
5    if (empty(q))
6    {
7      q->proc[0] = proc;
8      (q->size) = 1;
9      return;
10   }
```

```

  if (full(q))
  {
    printf("Queue overflow");
    exit(1);
  }

  for (int i = q->size; i > 0; i -= 1)
  {
    if (q->proc[i - 1]->priority <= proc->priority)
    {
      q->proc[i] = proc;
      i = -1;
    }
    else
      q->proc[i] = q->proc[i - 1];

    if (i == 1)
      q->proc[0] = proc;
  }

  (q->size) += 1;
  return;
}
```

About `dequeue()`, we just pop the head of the queue and move everything up afterwards. In this assignment, elements are processes.

```
struct pcb_t *dequeue(struct queue_t *q)
{
  /*
   * Return a pcb whose priority is the highest
   * in the queue [q] and remember to remove it from q
   */

  if (q->size <= 0)
    return NULL;

  if (q->size >= MAX_QUEUE_SIZE) // * As if this will ever happen
    ;

  struct pcb_t *result = q->proc[0];

  for (int i = 0; i < q->size; i += 1)
    q->proc[i] = q->proc[i + 1];

  q->size -= 1;
  q->proc[q->size] = NULL;
  return result;
}
```

1.4.2 sched.c

In sched.c, we need to implement the `get_proc()` method. `get_proc()` will dequeue a process from the `ready_queue`. If there is no process in the `ready_queue`, it will check the `run_queue` and push all the processes there.

```
1  struct pcb_t *get_proc(void)
2  {
3    struct pcb_t *proc = NULL;
4    /*
5      * Get a process from [ready_queue]. If ready queue
6      * is empty, push all processes in [run_queue] back to
7      * [ready_queue] and return the highest priority one.
8      * Remember to use lock to protect the queue.
9      */
10
11   pthread_mutex_lock(&queue_lock);
12
13   if (empty(&ready_queue))
14     while (!empty(&run_queue))
15       enqueue(&ready_queue, dequeue(&run_queue));
16
17   proc = dequeue(&ready_queue);
18
19   pthread_mutex_unlock(&queue_lock);
20
21   return proc;
22 }
```

2 Memory Management

2.1 Advantage and disadvantage of segmentation with paging

Segmentation with paging is a combination of both Segmentation and Paging techniques, where the main memory is divided into many segments with varying sizes and each segment houses a page table that further maps to many pages corresponding to frames in physical storage.

Upon a translation request is received, the Memory Management Unit will infer the position of the requested page from the virtual address by extracting the relevant bits, treating them as the indices to look for in the segment table and subsequently, the page table in said segment.

Compared to Segmentation or Paging alone, Segmentation with Paging introduces a wide range of advantages but also contributes a fair bit to the disadvantageous side.

- **Advantages:**
  - The page table size is reduced as pages are present only for data of segments, hence reducing the memory requirements.
  - Segmentation with Paging reduces external fragmentation.
- **Disadvantages:**
  - It is much more complex, making it harder to implement and increasing the translation and memory access time.
  - Memory is still vulnerable to internal fragmentation.

2.2 The memory after each allocation and deallocation call

We will be modifying our program, namely the source file mem.c to call `dump()` right before `alloc_mem()` and `free_mem()` end. The tests we will be using are `mem_0` and `mem_1`.

- `mem_0`

```
------ MEMORY MANAGEMENT TEST 0
-----------------------------------
./mem input/proc/m0
Dumping memory content: #after alloc 13535 0
000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
001: 00400-007ff - PID: 01 (idx 001, nxt: 002)
002: 00800-00bff - PID: 01 (idx 002, nxt: 003)
003: 00c00-00fff - PID: 01 (idx 003, nxt: 004)
004: 01000-013ff - PID: 01 (idx 004, nxt: 005)
005: 01400-017ff - PID: 01 (idx 005, nxt: 006)
006: 01800-01bff - PID: 01 (idx 006, nxt: 007)
007: 01c00-01fff - PID: 01 (idx 007, nxt: 008)
008: 02000-023ff - PID: 01 (idx 008, nxt: 009)
009: 02400-027ff - PID: 01 (idx 009, nxt: 010)
010: 02800-02bff - PID: 01 (idx 010, nxt: 011)
011: 02c00-02fff - PID: 01 (idx 011, nxt: 012)
012: 03000-033ff - PID: 01 (idx 012, nxt: 013)
013: 03400-037ff - PID: 01 (idx 013, nxt: -01)
```

```
Dumping memory content: #after alloc 1568 1
000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
001: 00400-007ff - PID: 01 (idx 001, nxt: 002)
002: 00800-00bff - PID: 01 (idx 002, nxt: 003)
003: 00c00-00fff - PID: 01 (idx 003, nxt: 004)
004: 01000-013ff - PID: 01 (idx 004, nxt: 005)
005: 01400-017ff - PID: 01 (idx 005, nxt: 006)
006: 01800-01bff - PID: 01 (idx 006, nxt: 007)
007: 01c00-01fff - PID: 01 (idx 007, nxt: 008)
008: 02000-023ff - PID: 01 (idx 008, nxt: 009)
009: 02400-027ff - PID: 01 (idx 009, nxt: 010)
010: 02800-02bff - PID: 01 (idx 010, nxt: 011)
011: 02c00-02fff - PID: 01 (idx 011, nxt: 012)
012: 03000-033ff - PID: 01 (idx 012, nxt: 013)
013: 03400-037ff - PID: 01 (idx 013, nxt: -01)
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
Dumping memory content: #after free 0
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
Dumping memory content: #after alloc 1386 2
000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
001: 00400-007ff - PID: 01 (idx 001, nxt: -01)
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
Dumping memory content: #after alloc 4564 4
000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
001: 00400-007ff - PID: 01 (idx 001, nxt: -01)
002: 00800-00bff - PID: 01 (idx 000, nxt: 003)
003: 00c00-00fff - PID: 01 (idx 001, nxt: 004)
004: 01000-013ff - PID: 01 (idx 002, nxt: 005)
005: 01400-017ff - PID: 01 (idx 003, nxt: 006)
006: 01800-01bff - PID: 01 (idx 004, nxt: -01)
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
Dumping memory content: #final result
000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
        003e8: 15
001: 00400-007ff - PID: 01 (idx 001, nxt: -01)
002: 00800-00bff - PID: 01 (idx 000, nxt: 003)
003: 00c00-00fff - PID: 01 (idx 001, nxt: 004)
004: 01000-013ff - PID: 01 (idx 002, nxt: 005)
005: 01400-017ff - PID: 01 (idx 003, nxt: 006)
006: 01800-01bff - PID: 01 (idx 004, nxt: -01)
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
        03814: 66
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
NOTE: Read file output/m0 to verify your result
```

- mem_1

```
------ MEMORY MANAGEMENT TEST 1
-----------------------------------
./mem input/proc/m1
Dumping memory content: #after alloc 13535 0
000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
001: 00400-007ff - PID: 01 (idx 001, nxt: 002)
002: 00800-00bff - PID: 01 (idx 002, nxt: 003)
003: 00c00-00fff - PID: 01 (idx 003, nxt: 004)
004: 01000-013ff - PID: 01 (idx 004, nxt: 005)
005: 01400-017ff - PID: 01 (idx 005, nxt: 006)
006: 01800-01bff - PID: 01 (idx 006, nxt: 007)
007: 01c00-01fff - PID: 01 (idx 007, nxt: 008)
008: 02000-023ff - PID: 01 (idx 008, nxt: 009)
009: 02400-027ff - PID: 01 (idx 009, nxt: 010)
010: 02800-02bff - PID: 01 (idx 010, nxt: 011)
011: 02c00-02fff - PID: 01 (idx 011, nxt: 012)
012: 03000-033ff - PID: 01 (idx 012, nxt: 013)
013: 03400-037ff - PID: 01 (idx 013, nxt: -01)
Dumping memory content: #after alloc 1568 1
000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
001: 00400-007ff - PID: 01 (idx 001, nxt: 002)
002: 00800-00bff - PID: 01 (idx 002, nxt: 003)
003: 00c00-00fff - PID: 01 (idx 003, nxt: 004)
004: 01000-013ff - PID: 01 (idx 004, nxt: 005)
005: 01400-017ff - PID: 01 (idx 005, nxt: 006)
006: 01800-01bff - PID: 01 (idx 006, nxt: 007)
007: 01c00-01fff - PID: 01 (idx 007, nxt: 008)
008: 02000-023ff - PID: 01 (idx 008, nxt: 009)
009: 02400-027ff - PID: 01 (idx 009, nxt: 010)
010: 02800-02bff - PID: 01 (idx 010, nxt: 011)
011: 02c00-02fff - PID: 01 (idx 011, nxt: 012)
012: 03000-033ff - PID: 01 (idx 012, nxt: 013)
013: 03400-037ff - PID: 01 (idx 013, nxt: -01)
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
Dumping memory content: #after free 0
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
Dumping memory content: #after alloc 1386 2
000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
001: 00400-007ff - PID: 01 (idx 001, nxt: -01)
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
Dumping memory content: #after alloc 4564 4
000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
001: 00400-007ff - PID: 01 (idx 001, nxt: -01)
002: 00800-00bff - PID: 01 (idx 000, nxt: 003)
003: 00c00-00fff - PID: 01 (idx 001, nxt: 004)
004: 01000-013ff - PID: 01 (idx 002, nxt: 005)
005: 01400-017ff - PID: 01 (idx 003, nxt: 006)
```

```
006: 01800-01bff - PID: 01 (idx 004, nxt: -01)
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
Dumping memory content: #after free 2
002: 00800-00bff - PID: 01 (idx 000, nxt: 003)
003: 00c00-00fff - PID: 01 (idx 001, nxt: 004)
004: 01000-013ff - PID: 01 (idx 002, nxt: 005)
005: 01400-017ff - PID: 01 (idx 003, nxt: 006)
006: 01800-01bff - PID: 01 (idx 004, nxt: -01)
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
Dumping memory content: #after free 4
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
Dumping memory content: #after free 1
Dumping memory content: #final result
NOTE: Read file output/m1 to verify your result (your
implementation should print nothing)
```

2.3 Code implementation

In this scope of the exercise, we need to complete 4 functions

- `get_page_table()`
- `translate()`
- `alloc_mem()`
- `free_mem()`

Let's discuss about the virtual memory engine used in Segmentation with Paging to manage the memory before jumping into each function above:

- By default, we are given 20 bits to represent the memory, which is equivalent to 1MB
- 5 high bits (0–4) are used for segment index
- The next 5 bits (5–9) are used for the page table index
- the last 10 low bits (10–19) are used for the offset

2.3.1 Get Page table from the Segment table

For this job, we need to get the page table from the given segment table by using the (5–9) bits and search it in the segment table. If the virtual index field of the `seg_table` has the same value as the parameter passed, we return the corresponding `page_table` through the pointer page.

```
/* Search for page table table from the a segment table */
static struct page_table_t *get_page_table(
    addr_t index, // Segment level index
    struct seg_table_t *seg_table)
{ // first level table

  /*
   * Given the segment index [index], you must go through each
   * row of the segment table [seg_table] and check if the v_index
   * field of the row is equal to the index
   */

  int i;
  for (i = 0; i < seg_table->size; i++)
  {
    // Enter your code here
    if (seg_table->table[i].v_index == index)
      return seg_table->table[i].pages;
  }
  return NULL;
}
```

2.3.2 Translate virtual address to physical address

We will need to build the full physical address from the given virtual address if said virtual address is owned by the process.

The physical address has two parts: Index and Offset. The offset of the physical address is also the offset of virtual address, and can simply be retrieved by using the `get_offset()` function.

The physical offset, which is the first 10 high bits of the physical address, can be found by looking for a segment with its `v_index` being equal to the first 0–4 bits of the virtual address, obtained by calling `get_first_level()`.

A page within said segment that has its `v_index` equal to the first 5–9 bits of the virtual address, obtained by the calling `get_second_level()`. After finding such page, we now have the `p_index`.

To make the physical address, we shift the `p_index` to the left by `OFFSET_LEN` bits and use `OR` operator — `|` to concatenate it and the `offset` together.

```
/*
 * Translate virtual address to physical address. If [virtual_addr] is valid,
 * return 1 and write its physical counterpart to [physical_addr].
 * Otherwise, return 0
 */
static int translate(
    addr_t virtual_addr,   // Given virtual address
    addr_t *physical_addr, // Physical address to be returned
    struct pcb_t *proc)
{ // Process uses given virtual address

  /* Offset of the virtual address */
  addr_t offset = get_offset(virtual_addr);
  /* The first layer index */
  addr_t first_lv = get_first_lv(virtual_addr);
  /* The second layer index */
  addr_t second_lv = get_second_lv(virtual_addr);

  /* Search in the first level */
  struct page_table_t *page_table = NULL;
  page_table = get_page_table(first_lv, proc->seg_table);
  if (page_table == NULL)
    return 0;

  int i;
  for (i = 0; i < page_table->size; i++)
  {
    if (page_table->table[i].v_index == second_lv)
    {
      /*
       * Concatenate the offset of the virtual addess
       * to [p_index] field of page_table->table[i] to
       * produce the correct physical address and save it to
       * [*physical_addr]
```

```
35          */
36
37          *physical_addr = (page_table->table[i].p_index <<
            OFFSET_LEN) | offset;
38          return 1;
39        }
40      }
41      return 0;
42    }
```

What exactly did we do here?

Step 1: We get the segment index (`first_level`), page index (`second_level`) and offset.

Step 2: Use the segment index, to find the corresponding `page_table`.

- If the `page_table` exists, we move on with Step 3.
- If the corresponding `page_table` does not exist, the return value of `translate()` is 0.

Step 3: Traverse the `page_table`, find a page which has the same `v_index` field as page index we got from earlier.

Step 4: If we find a matching page with matching `v_index`, concatenate the `p_index` there using "|" with the offset after shifting it to the left by `OFFSET_LEN`.

2.3.3 Memory allocation

First we must check if the amount of free memory in virtual address space and physical address space is large enough to represent the amount of required memory. If so, set `mem_avail` to 1.

```
1     /* iterate over frames, note the indices of free frames
2     * while also counting for physical space */
3     uint32_t free_page = 0;
4     int free_frames[req_page];
5     int i = 0, j = 0;
6     while (free_page < req_page && i < NUM_PAGES)
7     {
8       if (_mem_stat[i].proc == 0)
9       {
10        free_page += 1;
11        free_frames[j] = i;
12        j += 1;
13      }
14      i += 1;
15    }
16
17    // * Minimum space required
18    if (free_page >= req_page)
19      mem_avail = 1;
```

In this section of the program, we will walk through a special structure called `mem_stat` to find the number of free frames (where `proc == 0`), we also store their indices into an array called `free_page_index` for later use.

In "checking virtual space", `1 << ADDRESS_SIZE` is the max range of memory we are given. If the memory needed (`num_pages * PAGE_SIZE`) does not overflow the memory space, we set `mem_avail` to true.

```
1    if (free_page >= req_page)
2      mem_avail = 1;
```

If the `mem_avail` is set to true, we have permission to update the status of physical frames and map them to virtual pages.

Our next task is to update the status of physical frames and virtual pages which will be allocated to `proc` in `mem_stat`.

There are two steps:

Step 1: Update `proc`, `index`, and `next`

Step 2: Add entries to segment table page tables of `proc` to ensure accesses to allocated memory slot is valid

```
1      /* We could allocate new memory region to the process */
2      head_ptr = proc->bp;
3      proc->bp = head_ptr + req_page * PAGE_SIZE;
4
5      /*
6       * Update status of physical pages which will be allocated
7       * to [proc] in _mem_stat. Tasks to do:
8       *        - Update [proc], [index], and [next] field
9       *        - Add entries to segment table page tables of [proc]
10      *          to ensure accesses to allocated memory slot is
11      *          valid.
12      */
13
14     /* physical
15      * update index, next and proc in frames with saved indices */
16     for (j = 0; j < req_page; j++)
17     {
18       int frame_idx = free_frames[j];
19       _mem_stat[frame_idx].proc = proc->pid;
20       _mem_stat[frame_idx].index = j;
21       _mem_stat[frame_idx].next = -1;
22
23       if (j < req_page - 1)
24         _mem_stat[frame_idx].next = free_frames[j + 1];
25     }
```

By using the current un-updated break pointer, we get the address of the first byte in the memory region allocated to the process to return it later.

```
    head_ptr = proc->bp;
    proc->bp = head_ptr + req_page * PAGE_SIZE;
```

The final task we need to do to complete `alloc_mem()` is managing the virtual pages.

```
    /* virtual
     * infer 1st and 2nd lv index, use them to search for seg and
     * page with matching v_index
     * if there isn't any, alloc/assign to a new one at (size - 1)
     * for both tables, then assign the values needed
     */
    addr_t itr_ptr = head_ptr;
    for (j = 0; j < req_page; j++)
    {
      addr_t first_lv = get_first_lv(itr_ptr);
      addr_t second_lv = get_second_lv(itr_ptr);
      struct seg_table_t *seg_table = proc->seg_table;
      struct page_table_t *page_table = get_page_table(first_lv,
      seg_table);

      // create new entry
      if (page_table == NULL)
      {
        int seg_size = seg_table->size;
        seg_table->table[seg_size].v_index = first_lv;
        page_table = (struct page_table_t *)malloc(sizeof(struct
        page_table_t));
        seg_table->table[seg_size].pages = page_table;
        page_table->size = 0;
        seg_table->size++;
      }

      // assign values
      int page_size = page_table->size;
      page_table->table[page_size].v_index = second_lv;
      page_table->table[page_size].p_index = free_frames[j];
      page_table->size++;

      itr_ptr += PAGE_SIZE;
    }
  }
```

Using the first byte of the range of memory region, we will get the corresponding:

- `seg_table` index — first_lv
- `page_table` index — second_lv

By concatenating the current `seg_table` which is from `proc->seg_table` and `first_level`, we can get the corresponding `page_table`. If there is no `page_table`, we will make a new one with empty size and also increase the size of `seg_table` by

1. Then we assign a virtual index to that page table (a new one if we didn't find anything or a corresponding one if we did) and move to the next page.

2.3.4 Free memory

In this task we need to release memory region allocated by `proc`. The first byte of this region is indicated by `address`.

The procedure should be:

- Set flag `proc` of physical page use by the memory block back to zero to indicate that it is free
- Remove unused entries in segment table and page tables of the process `proc`

We will complete the `free_mem()` in the following steps:

Step 1: We need to check the given address (it is virtual address). If this virtual address does not have a corresponding physical address, return 1 to finish the function immediately.

```
1   /*
2    * Release memory region allocated by [proc]. The first
    byte of
3    * this region is indicated by [address]. Task to do:
4    *        - Set flag [proc] of physical page use by the
    memory block
5    *          back to zero to indicate that it is free.
6    *        - Remove unused entries in segment table and page
    tables of
7    *          the process [proc].
8    *        - Remember to use lock to protect the memory from
    other
9    *          processes.
10   */
11
12  pthread_mutex_lock(&mem_lock);
13
14  // swap return values?
15  // possible fix for issue#2?
16  addr_t p_addr;
17  addr_t v_addr = address;
18  if (translate(v_addr, &p_addr, proc) == 0)
19    return 1;
```

Else, we move on with step 2

Step 2: Calculate the number of frame pages in `mem_stat` that unused.

```
1   /* mark frames in _mem_stat as unused
2     * count how many were marked */
3   int freed_pages = 0;
4   int p_index = (p_addr >> OFFSET_LEN);
5   while (p_index != -1)
```

```
6     {
7       _mem_stat[p_index].proc = 0;
8       freed_pages += 1;
9       p_index = _mem_stat[p_index].next;
10    }
```

Step 3: If the page we need to deallocate is not the last page, bump it up one by one and update the address of registers. Else, move on with step 4

```
1     /*
2     * Preventing fragmentation:
3     * We are removing page entries of the process
4     * hence bumping all pages forward to maintain contiguity
5     * Uses new_v_addr to get the location of the destination
      and
6     * old_v_addr for the source through, achieved by getting
7     * their first and second levels
8     */
9     if (proc->bp != v_addr)
10    {
11      addr_t new_v_addr = address; // Address of next data
        chunk after dealloc this process
12      addr_t old_v_addr = v_addr;  // Address of next data
        chunk
13
14      // move one page up at a time
15      while (proc->bp != old_v_addr)
16      {
17        // retrieve their first and second levels
18        addr_t new_first_lv = get_first_lv(new_v_addr);
19        addr_t new_second_lv = get_second_lv(new_v_addr);
20        addr_t old_first_lv = get_first_lv(old_v_addr);
21        addr_t old_second_lv = get_second_lv(old_v_addr);
22
23        /* locate the dest and src page based on levels
24         * use their normal indices to move */
25        struct seg_table_t *seg_table = proc->seg_table;
26        struct page_table_t *new_page_table =
          get_page_table(new_first_lv, seg_table);
27        struct page_table_t *old_page_table =
          get_page_table(old_first_lv, seg_table);
28        if (new_page_table != NULL && old_page_table != NULL)
29        {
30          int new_page_index;
31          int old_page_index;
32
33          for (new_page_index = 0; new_page_index <
            new_page_table->size; new_page_index++)
34            if (new_page_table->table[new_page_index].v_index
              == new_second_lv)
```

```
35              break;
36          for (old_page_index = 0; old_page_index <
            old_page_table->size; old_page_index++)
37            if (old_page_table->table[old_page_index].v_index
              == old_second_lv)
38              break;
39
40          new_page_table->table[new_page_index].p_index =
            old_page_table->table[old_page_index].p_index;
41          new_page_table->table[new_page_index].v_index =
            new_second_lv;
42        }
43
44        // update registers that hold old addresses
45        for (int i = 0; i < 10; i++)
46        {
47          if (proc->regs[i] == old_v_addr)
48          {
49            proc->regs[i] = new_v_addr;
50            break;
51          }
52        }
53
54        new_v_addr += PAGE_SIZE;
55        old_v_addr += PAGE_SIZE;
56      }
57    }
```

Step 4: Decreases the size and free `page_table` if empty afterwards

```
1     // decreases size and free page_table if empty afterwards
2     int tmp = freed_pages;
3     while (freed_pages > 0)
4     {
5       int seg_size = proc->seg_table->size;
6       struct seg_table_t *seg_table = proc->seg_table;
7       struct page_table_t *page_table = get_page_table(seg_size
        - 1, seg_table);
8       if (page_table != NULL)
9       {
10        page_table->size--;
11        if (page_table->size == 0)
12        {
13          free(page_table);
14          seg_table->size--;
15        }
16      }
17      freed_pages--;
18    }
```

Step 5: Update break pointer

```
1    proc->bp -= tmp * PAGE_SIZE;
```

3 Put them all together

To check if our implementation of the operating system works correctly or not, so far we have run a few tests on separate module of `mem` and `sched`. Now, we test both of them together using the testing scheme `os`. This end-all-be-all scheme consists of 2 tests `os_0` and `os_1` which involve multiple processes performing a wide range of instructions on multiple CPU threads. This is what our operating system returned:

- `os_0`

```
----- OS TEST 0
-------------------------------------------------
./os os_0
Time slot   0
        Loaded a process at input/proc/p0, PID: 1
Time slot   1
        CPU 0: Dispatched process  1
Time slot   2
        Loaded a process at input/proc/p1, PID: 2
Time slot   3
        CPU 1: Dispatched process  2
        Loaded a process at input/proc/p1, PID: 3
Time slot   4
        Loaded a process at input/proc/p1, PID: 4
Time slot   5
Time slot   6
Time slot   7
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  3
Time slot   8
Time slot   9
        CPU 1: Put process  2 to run queue
        CPU 1: Dispatched process  4
Time slot  10
Time slot  11
Time slot  12
Time slot  13
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
Time slot  14
Time slot  15
        CPU 1: Put process  4 to run queue
        CPU 1: Dispatched process  2
Time slot  16
Time slot  17
        CPU 0: Processed  1 has finished
        CPU 0: Dispatched process  3
```

```
Time slot  18
Time slot  19
        CPU 1: Processed  2 has finished
        CPU 1: Dispatched process  4
Time slot  20
Time slot  21
        CPU 0: Processed  3 has finished
        CPU 0 stopped
Time slot  22
Time slot  23
        CPU 1: Processed  4 has finished
        CPU 1 stopped

MEMORY CONTENT:
000: 00000-003ff - PID: 03 (idx 000, nxt: 001)
001: 00400-007ff - PID: 03 (idx 001, nxt: 006)
002: 00800-00bff - PID: 02 (idx 000, nxt: 003)
003: 00c00-00fff - PID: 02 (idx 001, nxt: 004)
004: 01000-013ff - PID: 02 (idx 002, nxt: 005)
005: 01400-017ff - PID: 02 (idx 003, nxt: -01)
006: 01800-01bff - PID: 03 (idx 002, nxt: 012)
007: 01c00-01fff - PID: 02 (idx 000, nxt: 008)
008: 02000-023ff - PID: 02 (idx 001, nxt: 009)
009: 02400-027ff - PID: 02 (idx 002, nxt: 010)
        025e7: 0a
010: 02800-02bff - PID: 02 (idx 003, nxt: 011)
011: 02c00-02fff - PID: 02 (idx 004, nxt: -01)
012: 03000-033ff - PID: 03 (idx 003, nxt: -01)
013: 03400-037ff - PID: 04 (idx 000, nxt: 028)
014: 03800-03bff - PID: 03 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 03 (idx 001, nxt: 016)
016: 04000-043ff - PID: 03 (idx 002, nxt: 017)
        041e7: 0a
017: 04400-047ff - PID: 03 (idx 003, nxt: 018)
018: 04800-04bff - PID: 03 (idx 004, nxt: -01)
019: 04c00-04fff - PID: 04 (idx 000, nxt: 020)
020: 05000-053ff - PID: 04 (idx 001, nxt: 021)
021: 05400-057ff - PID: 04 (idx 002, nxt: 022)
        055e7: 0a
022: 05800-05bff - PID: 04 (idx 003, nxt: 023)
023: 05c00-05fff - PID: 04 (idx 004, nxt: -01)
028: 07000-073ff - PID: 04 (idx 001, nxt: 029)
029: 07400-077ff - PID: 04 (idx 002, nxt: 030)
030: 07800-07bff - PID: 04 (idx 003, nxt: -01)
047: 0bc00-0bfff - PID: 01 (idx 000, nxt: -01)
        0bc14: 64
NOTE: Read file output/os_0 to verify your result
```

- os_1

```
----- OS TEST 1
--------------------------------------------------
./os os_1
Time slot   0
        Loaded a process at input/proc/p0, PID: 1
Time slot   1
Time slot   2
        CPU 3: Dispatched process  1
        Loaded a process at input/proc/s3, PID: 2
Time slot   3
        CPU 2: Dispatched process  2
        Loaded a process at input/proc/m1, PID: 3
Time slot   4
        CPU 3: Put process  1 to run queue
        CPU 3: Dispatched process  3
Time slot   5
        CPU 2: Put process  2 to run queue
        CPU 2: Dispatched process  2
        CPU 1: Dispatched process  1
Time slot   6
        Loaded a process at input/proc/s2, PID: 4
        CPU 3: Put process  3 to run queue
        CPU 3: Dispatched process  4
Time slot   7
        CPU 2: Put process  2 to run queue
        CPU 2: Dispatched process  2
        CPU 0: Dispatched process  3
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
        Loaded a process at input/proc/m0, PID: 5
Time slot   8
        CPU 3: Put process  4 to run queue
        CPU 3: Dispatched process  5
Time slot   9
        Loaded a process at input/proc/p1, PID: 6
        CPU 2: Put process  2 to run queue
        CPU 2: Dispatched process  6
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  4
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  2
Time slot  10
        CPU 3: Put process  5 to run queue
        CPU 3: Dispatched process  1
Time slot  11
        Loaded a process at input/proc/s0, PID: 7
        CPU 1: Put process  4 to run queue
        CPU 1: Dispatched process  7
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  4
```

```
        CPU 2: Put process  6 to run queue
        CPU 2: Dispatched process  2
Time slot  12
        CPU 3: Put process  1 to run queue
        CPU 3: Dispatched process  3
Time slot  13
        CPU 2: Put process  2 to run queue
        CPU 2: Dispatched process  5
        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  2
Time slot  14
        CPU 3: Put process  3 to run queue
        CPU 3: Dispatched process  1
        CPU 1: Processed  2 has finished
        CPU 1: Dispatched process  6
Time slot  15
        CPU 2: Put process  5 to run queue
        CPU 2: Dispatched process  7
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
        Loaded a process at input/proc/s1, PID: 8
Time slot  16
        CPU 3: Processed  1 has finished
        CPU 3: Dispatched process  8
        CPU 1: Put process  6 to run queue
        CPU 1: Dispatched process  5
Time slot  17
        CPU 2: Put process  7 to run queue
        CPU 2: Dispatched process  4
        CPU 0: Processed  3 has finished
        CPU 0: Dispatched process  7
        CPU 3: Put process  8 to run queue
        CPU 3: Dispatched process  6
        CPU 1: Put process  5 to run queue
        CPU 1: Dispatched process  8
Time slot  18
Time slot  19
        CPU 2: Put process  4 to run queue
        CPU 2: Dispatched process  5
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  4
Time slot  20
        CPU 3: Put process  6 to run queue
        CPU 3: Dispatched process  7
        CPU 1: Put process  8 to run queue
        CPU 1: Dispatched process  8
        CPU 2: Processed  5 has finished
        CPU 2: Dispatched process  6
```

```
Time slot  21
        CPU 0: Processed  4 has finished
        CPU 0 stopped
Time slot  22
        CPU 3: Put process  7 to run queue
        CPU 3: Dispatched process  7
        CPU 2: Put process  6 to run queue
        CPU 2: Dispatched process  6
        CPU 1: Put process  8 to run queue
        CPU 1: Dispatched process  8
Time slot  23
        CPU 1: Processed  8 has finished
        CPU 1 stopped
Time slot  24
        CPU 3: Put process  7 to run queue
        CPU 3: Dispatched process  7
        CPU 2: Processed  6 has finished
        CPU 2 stopped
Time slot  25
Time slot  26
        CPU 3: Put process  7 to run queue
        CPU 3: Dispatched process  7
Time slot  27
Time slot  28
        CPU 3: Put process  7 to run queue
        CPU 3: Dispatched process  7
Time slot  29
        CPU 3: Processed  7 has finished
        CPU 3 stopped

MEMORY CONTENT:
000: 00000-003ff - PID: 01 (idx 000, nxt: -01)
        00014: 64
001: 00400-007ff - PID: 05 (idx 000, nxt: 002)
        007e8: 15
002: 00800-00bff - PID: 05 (idx 001, nxt: -01)
003: 00c00-00fff - PID: 05 (idx 000, nxt: 004)
004: 01000-013ff - PID: 05 (idx 001, nxt: 005)
005: 01400-017ff - PID: 05 (idx 002, nxt: 006)
006: 01800-01bff - PID: 05 (idx 003, nxt: 007)
007: 01c00-01fff - PID: 05 (idx 004, nxt: -01)
012: 03000-033ff - PID: 06 (idx 000, nxt: 013)
013: 03400-037ff - PID: 06 (idx 001, nxt: 014)
014: 03800-03bff - PID: 06 (idx 002, nxt: 015)
015: 03c00-03fff - PID: 06 (idx 003, nxt: -01)
024: 06000-063ff - PID: 05 (idx 000, nxt: 025)
        06014: 66
025: 06400-067ff - PID: 05 (idx 001, nxt: -01)
026: 06800-06bff - PID: 06 (idx 000, nxt: 027)
027: 06c00-06fff - PID: 06 (idx 001, nxt: 028)
```

```
028: 07000-073ff - PID: 06 (idx 002, nxt: 029)
        071e7: 0a
029: 07400-077ff - PID: 06 (idx 003, nxt: 030)
030: 07800-07bff - PID: 06 (idx 004, nxt: -01)
NOTE: Read file output/os_1 to verify your result
```

It can be seen that the output after each execution is inconsistent and rarely identical to the given sample outputs. The reasons for such behavior is that the processes are run concurrently, thus no chronological logic is created between the processes.