ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC BÁCH KHOA

KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

ASSIGNMENT

SIMPLE OPERATING SYSTEM

Class: CC04

No.	Full name	Student ID	% of work
1	Lê Việt Hoàng	2052093	25%
2	Lê Ngọc Minh Thư	2053476	25%
3	Lê Hoàng Minh	2052595	25%
4	Tôn Huỳnh Long	2052153	25%

I. Scheduler:

1. Question:

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

A priority feedback queue uses 2 queues:

- *ready_queue*: This queue contains processes in the order of their priority values, with the one having the highest priority value being the one chosen to be executed once the CPU is free to do so.
- *run_queue*: This queue contains processes waiting for the next execution as their work is not completed. When the CPU is free and the ready_queue is empty, the contents of this queue are moved back to ready_queue.

Compared to FIFO, Round Robin:

We can exploit the priority of processes with the Priority feedback queue but cannot with FIFO and RR because FIFO and RR just consider the order of arrival of each process to the queue.

Priority feedback queue supports preemption, which is switching the process. If there is a process with higher priority than the running process, the higher priority process will preempt the running process at the end of the time slice. This feature does not occur at FIFO.

Compared to Priority Scheduling:

The priority feedback queue prevents the indefinite blocking (starvation) of low-priority processes since all processes within the *ready_queu*e must be executed first before processes in *run_queue* can have a chance to go back and continue. In Priority scheduling, if high-priority processes keep coming to the queue frequently, the low-priority processes may get blocked indefinitely if there's no aging mechanism.

However, one possible weakness the priority feedback queue may have is that if new lower-priority processes keep coming to the *ready_queue*, higher-priority processes in the *run_queue* might also be blocked for a long time, resulting in processes having worse turnout time.

I.2. Implementation:

First, we need to implement the functions *enqueue()* and *dequeue()* in queue.c:

Although we call them queues, here they are actually implemented as dynamic arrays.

- **enqueue():** Add a PCB to the waiting queue. The queue is organized such that the process that has the lowest priority is at the first index, while the process with the lowest priority is located at the end.

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

        //find position of proc based on priority
        int idx;
        for(idx = 0; idx < q->size; idx++){
            if (proc->priority <= q->proc[idx]->priority) break;
        }

        //move the back to make space for incoming proc
        for(int i = q->size; i > idx; i--){
            q->proc[i] = q->proc[i - 1];
        }

        //put proc to idx in queue, increment size
        q->proc[idx] = proc;
        q->size++;
}
```

- **dequeue()**: Get the process at the end of the array, which according to how we enqueued the processes, means we're getting the one with the highest priority

```
struct pcb_t * dequeue(struct queue_t * q) {
        /* TODO: return a pcb whose prioprity is the highest
         * in the queue [q] and remember to remove it from q
         * */
        //if empty, return null
        if (empty(q)) return NULL;

        //else get proc at last index (highest priority)
        q->size--;
        return q->proc[q->size];
}
```

2. Scheduler:

We implement *get_proc()* in scheh.c to get the PCB of a process in *ready_queue*. If the queue is empty when *get_proc()* is executing, we move every PCB of processes waiting in *run_queue* back to *ready_queue* before trying to get a process from *ready_queue()* again.

```c
struct pcb_t * get_proc(void) {
        struct pcb_t * proc = NULL;
        /*TODO: get a process from [ready_queue]. If ready queue
         * is empty, push all processes in [run_queue] back to
         * [ready_queue] and return the highest priority one.
         * Remember to use lock to protect the queue.
         * */
        pthread_mutex_lock(&queue_lock);
        if (empty(&ready_queue))
        {
                //copy all proc from run_queue to ready_queue
                for(int i = 0; i < run_queue.size; i++){
                    ready_queue.proc[i] = run_queue.proc[i];
                }

                //new sizes
                ready_queue.size = run_queue.size;
                run_queue.size = 0;
        }

        //dequeue, it returns null if still empty
        proc = dequeue(&ready_queue);
        pthread_mutex_unlock(&queue_lock);

        return proc;
}
```

I.3: Result:

Compile code by Makefile

make sched

Running the test:

make test_sched

Results:

- **sched_0**

```
------ 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
        Loaded a process at input/proc/s1, PID: 2
Time slot   4
Time slot   5
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
Time slot   6
Time slot   7
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot   8
Time slot   9
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  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
```

process		p1				p2				p1		p2		p1		p2	p1						
time slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	18	19	20	21	22	23

Average waiting time = 1

Average turnaround time = 17

```
------ 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
```

```
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
remin2k2@DESKTOP-6NE4O16:/mnt/c/Simple-Operating-system-main$
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
remin2k2@DESKTOP-6NE4O16:/mnt/c/Simple-Operating-system-main$
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
```

- **sched_1**

process		p1				p2		p3		p4		p2		p3		p1		p4		p2		p3	
time slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

process	p1		p4		p2	p3		p1		p4		p3		p1		p4		p3		p1		p4	p1	
time slot	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46

Average waiting time: 1.25

Average turnaround time: 35.25

II. Memory Management

1. Question:

In which system is segmentation with paging used (give an example of at least one system)? Explain clearly the advantages and disadvantages of segmentation with paging.

IA-32 systems (short for Intel Architecture, 32-bit) use segmentation with paging. Its segment is 4GB large.

Advantages:

Reduce memory usage compared to the paging technique. Because it can reduce the size of the page table which is stored in RAM. A program can be divided into segments and its segments have their page table. Therefore, the page table size will be reduced depending on the number of segments we divide.

Solve the external fragmentation problem because it uses a fix-sized page for the memory block. Every hole between allocated memory can be assigned to processes. It can be discontiguous, but it has a table for managing the pages.

Disadvantages:

There is internal fragmentation. This problem will occur when the memory required is not divisible by the page size. That is, a frame in RAM will not be used totally. For example, the page size is 5 byte and the program wants to allocate 11 bytes, there will be 3 frames allocated in the RAM and there is a frame that just needs 1 byte of it.

More complex than the paging technique because it requires the implementation of a page table and segment table

Increases the memory access time because it has to access the segment table and the page table to reach the actual memory it needs.

The page table must be contiguous in the memory because we access the frame in RAM through the offset of the base address of the page table. This may lead to external fragmentation.

2. Implementation:

1. **Mapping from virtual address to physical address:**

   We use 20 bits to represent the address, in which the first 5 bits encode the segment index, the next 5 bits represent the page index and the last 10 bits denote the offset.

Next, we make the transition from the virtual address of a process to the address. only physically by completing 2 functions: get_page_table() and translated in mem file.c.

get_page_table(): Find a paging table with a segment index for a process.

```
static struct page_table_t *get_page_table(
        addr_t index, // Segment level index
        struct seg_table_t *seg_table)
{ // first level table
        /*
         * TODO: 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;
}
```

translate(): use get_page_table( ) to convert from virtual address to physical address.

```
/* 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)
                {
                        /* TODO: 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] */
                        if(_mem_stat[page_table->table[i].p_index].proc != proc->pid) return 0;
                        *physical_addr = (page_table->table[i].p_index << OFFSET_LEN) + offset;
                        return 1;
                }
        }
        return 0;
}
```

2. Memory allocation and recovery

Memory allocation:

allocate: Support function for alloc_mem function.

Input:

- ret_mem: The address of the first byte in the memory area will be allocated.

- num_pages: The number of pages allocated to the process in the virtual address space.

- proc: The process to be allocated with more memory areas.

Purpose: Update the status of the physical frame will be initiated in the structure

_mem_stat structure, add entries to segment table, page table

- **alloc_mem:** Allocate memory to a process of size and save the address of thebyte

  the first in the memory area that has been allocated

```
addr_t alloc_mem(uint32_t size, struct pcb_t *proc)
{
	pthread_mutex_lock(&mem_lock);
	addr_t ret_mem = 0;

	uint32_t num_pages = (size % PAGE_SIZE) ? size / PAGE_SIZE + 1 : size / PAGE_SIZE;
	//uint32_t num_pages = (size % PAGE_SIZE) ? size / PAGE_SIZE : size / PAGE_SIZE + 1; // Number of pages we will use
	int mem_avail = 0; // We could allocate new memory region or not?


	proc->seg_table->size = 1 << SEGMENT_LEN;
	int num_physical_page_avail = 0;

	/*array to store the index of available frame in physical memory*/
	int *avail_physical_index = (int *)malloc(sizeof(int) * (num_pages + 1));
	for (int i = 0; i < NUM_PAGES; i++)
	{
		if (_mem_stat[i].proc == 0)
		{ // this page is free
			avail_physical_index[num_physical_page_avail] = i;
			num_physical_page_avail++;
		}
		if (num_physical_page_avail == num_pages)
		{
			mem_avail = 1; // we have enough physiccal memory
			break;
		}
	}
	if (proc->bp + num_pages * PAGE_SIZE >= RAM_SIZE)
		mem_avail = 0; // not logical memory
	/* end checking */
	if (mem_avail)
	{
		ret_mem = proc->bp;
		//proc->bp += num_pages * PAGE_SIZE;
```

```
            int loop = 0;

            for (; loop < num_pages; loop++)
            {
                  |
                  _mem_stat[avail_physical_index[loop]].proc = proc->pid;
                  _mem_stat[avail_physical_index[loop]].index = loop;
                  _mem_stat[avail_physical_index[loop]].next = avail_physical_index[loop + 1];
                  /* The first layer index */
                  addr_t first_lv = get_first_lv(proc->bp);
                  /* The second layer index */
                  addr_t second_lv = get_second_lv(proc->bp);
                  /* if the page haven't been initialized, then create it */
                  if (!proc->seg_table->table[first_lv].pages)
                  {
                        proc->seg_table->table[first_lv].pages = (struct page_table_t *)malloc(sizeof(struct page_table_t));
                        proc->seg_table->table[first_lv].pages->size = 1 << PAGE_LEN;
                  }
                  /* Update the page table fỏ further accesses */
                  proc->seg_table->table[first_lv].v_index = first_lv;
                  proc->seg_table->table[first_lv].pages->table[second_lv].v_index = second_lv;
                  proc->seg_table->table[first_lv].pages->table[second_lv].p_index = avail_physical_index[loop];
                  proc->bp += PAGE_SIZE;
            }
            _mem_stat[avail_physical_index[loop - 1]].next = -1;  // the last element's next field is -1
      }
      pthread_mutex_unlock(&mem_lock);
      //free(avail_physical_index);
      return ret_mem;
}
```

- **free_mem:** Free up the allocated memory area.

```
int free_mem(addr_t address, struct pcb_t *proc)
{
      /*TODO: Release memory region allocated by [proc]. The first byte of
       * this region is indicated by [address]. Task to do:
       *    - 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].
       *    - Remember to use lock to protect the memory from other
       *      processes.  */
      pthread_mutex_lock(&mem_lock);
      /* The first layer index */
      addr_t first_lv = get_first_lv(address);
      /* The second layer index */
      addr_t second_lv = get_second_lv(address);
      /* First frame to free */
      addr_t frame_index = proc->seg_table->table[first_lv].pages->table[second_lv].p_index;

      int  allow_to_free = 1;
      /* If the address not pointing to the first address of the allocated block*/
      if(_mem_stat[frame_index].index != 0) allow_to_free = 0;
      /* We also not allow to free other processes' memory */
      if(_mem_stat[frame_index].proc != proc->pid) allow_to_free = 0;
      /* Not to free the space that have not been allocated yet */
      if(_mem_stat[frame_index].proc == 0) allow_to_free = 0;

      while (allow_to_free)
      {
            _mem_stat[frame_index].proc = 0;
            frame_index = _mem_stat[frame_index].next;
            if (frame_index == -1)
                  break;
      }
      pthread_mutex_unlock(&mem_lock);

      return allow_to_free;
}
```

3. Result:

Run test:

make test_mem

This is the result that records the states of RAM in the program:

```
------ MEMORY MANAGEMENT TEST 0 ----------------------------------

/mem input/proc/m0
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
```

III. Put It All Together:

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

Firstly, a problem may occur with the queues (run queue and ready queue). In the scenario where the ready queue is almost full (1 slot left) and there exist two threads that try to enqueue, the system will encounter an error. The error can occur like this:

- Thread 1 check the queue (not full, remain 1 slot)
- Thread 2 check the queue (not full, remain 1 slot)
- Thread 1 enqueue (valid)
- Thread 2 enqueue (error occurs because queue is full)

The same problem can occur when the queue is almost empty (1 process left) and multiple CPUs try to dequeue.

Secondly, a problem may occur within the memory. There will be conflicts in getting the allocated memory.

Suppose that two different threads try to allocate the same page to itself, the first thread may find that the page at location 0x00 has not been used and so does the second thread. Then, both threads decided to take that page for allocation. The result is that two processes will record the same memory frame as theirs. The _mem_stat table will only record one of the two threads as the true owner, causing one process to not have enough

space for its memory. There may also be a problem in deallocating memory. For example, thread A tries to set the flag to indicate that the page is free. Then, thread B sees that page is free, so it gets that page to be allocated. However, thread A continues its job, which is to remove the unused entries in the page table and segment table of that page. Therefore, thread B will fail when it tries to reference that page.

Thirdly, if we do not use the mutex mechanism, there is no way the timer will work. The timeslot keeps incrementing despite unfinished tasks.

Finally, we combine Scheduler and VME to make a simple operating system.

Run the test:

```
make test_all
```

RESULT:

File os-0:

```
./os os_0
Time slot   0
Loaded a process at input/proc/p0, PID: 1
Time slot   1
CPU 1: Dispatched process  1
Time slot   2
Loaded a process at input/proc/p1, PID: 2
Time slot   3
CPU 0: 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
CPU 1: Put process  1 to run queue
CPU 1: Dispatched process  4
Time slot   7
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
```

CPU 0																							
Process		P2							P3							P2				P3				
time slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

CPU 1																								
Process		P1					P4											P1						
time slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

Memory States:

MEMORY CONTENT:
000: 00000-003ff - PID: 02 (idx 000, nxt: 001)
001: 00400-007ff - PID: 02 (idx 001, nxt: 007)
002: 00800-00bff - PID: 03 (idx 000, nxt: 003)
003: 00c00-00fff - PID: 03 (idx 001, nxt: 004)
004: 01000-013ff - PID: 03 (idx 002, nxt: 005)
005: 01400-017ff - PID: 03 (idx 003, nxt: -01)
007: 01c00-01fff - PID: 02 (idx 002, nxt: 008)
01de7: 0a
008: 02000-023ff - PID: 02 (idx 003, nxt: 009)
009: 02400-027ff - PID: 02 (idx 004, nxt: -01)
010: 02800-02bff - PID: 01 (idx 000, nxt: -01)
02814: 64
015: 03c00-03fff - PID: 04 (idx 000, nxt: 016)
016: 04000-043ff - PID: 04 (idx 001, nxt: 017)
017: 04400-047ff - PID: 04 (idx 002, nxt: 018)
045e7: 0a
018: 04800-04bff - PID: 04 (idx 003, nxt: 019)
019: 04c00-04fff - PID: 04 (idx 004, nxt: -01)
020: 05000-053ff - PID: 03 (idx 000, nxt: 021)
021: 05400-057ff - PID: 03 (idx 001, nxt: 022)
022: 05800-05bff - PID: 03 (idx 002, nxt: 023)
059e7: 0a
023: 05c00-05fff - PID: 03 (idx 003, nxt: 024)
024: 06000-063ff - PID: 03 (idx 004, nxt: -01)
029: 07400-077ff - PID: 04 (idx 000, nxt: 030)
030: 07800-07bff - PID: 04 (idx 001, nxt: 031)
031: 07c00-07fff - PID: 04 (idx 002, nxt: 032)
032: 08000-083ff - PID: 04 (idx 003, nxt: -01)
033: 08400-087ff - PID: 02 (idx 000, nxt: 034)
034: 08800-08bff - PID: 02 (idx 001, nxt: 035)
035: 08c00-08fff - PID: 02 (idx 002, nxt: 036)
036: 09000-093ff - PID: 02 (idx 003, nxt: -01)
NOTE: Read file output/os_0 to verify your result

File os_1:

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

Time slot 10
CPU 0: Put process 5 to run queue
CPU 0: Dispatched process 3
Loaded a process at input/proc/s0, PID: 7
CPU 2: Put process 6 to run queue
CPU 2: Dispatched process 2
CPU 3: Put process 4 to run queue
CPU 3: Dispatched process 7
Time slot 11
CPU 1: Processed 1 has finished
CPU 1: Dispatched process 6
Time slot 12
CPU 0: Put process 3 to run queue
CPU 0: Dispatched process 5
CPU 3: Put process 7 to run queue
CPU 3: Dispatched process 4
Time slot 13
CPU 2: Put process 2 to run queue
CPU 2: Dispatched process 7
CPU 1: Put process 6 to run queue
CPU 1: Dispatched process 3
Time slot 14
CPU 0: Put process 5 to run queue
CPU 0: Dispatched process 2
Time slot 15
CPU 1: Processed 3 has finished
CPU 1: Dispatched process 5
CPU 3: Put process 4 to run queue
CPU 3: Dispatched process 6
CPU 2: Put process 7 to run queue
CPU 2: Dispatched process 4
Loaded a process at input/proc/s1, PID: 8
Time slot 16
CPU 0: Put process 2 to run queue
CPU 0: Dispatched process 8
CPU 3: Put process 6 to run queue
CPU 3: Dispatched process 7
Time slot 17
CPU 1: Put process 5 to run queue
CPU 1: Dispatched process 2
CPU 2: Put process 4 to run queue
CPU 2: Dispatched process 5
CPU 1: Processed 2 has finished
CPU 1: Dispatched process 6
Time slot 18
CPU 0: Put process 8 to run queue
CPU 0: Dispatched process 8
CPU 2: Processed 5 has finished
remin2k2@DESKTOP-6NE4O16:/mnt/c/Simple-Operating-system-main$
CPU 3: Put process 7 to run queue
CPU 3: Dispatched process 7
Time slot 19
Time slot 20
CPU 2: Put process 4 to run queue
CPU 1: Put process 6 to run queue
CPU 1: Dispatched process 6
CPU 0: Put process 8 to run queue
CPU 0: Dispatched process 8
CPU 2: Dispatched process 4
Time slot 21
CPU 3: Put process 7 to run queue
CPU 3: Dispatched process 7
Time slot 22
CPU 1: Processed 6 has finished
CPU 1 stopped
CPU 0: Put process 8 to run queue
CPU 0: Dispatched process 8
CPU 2: Processed 4 has finished
CPU 2 stopped
Time slot 23
CPU 3: Put process 7 to run queue
CPU 3: Dispatched process 7
CPU 0: Processed 8 has finished
CPU 0 stopped
Time slot 24
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

CPU 0																												
Process						P4			P5			P3			P5		P2				P8								
time slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28

CPU 1																													
Process	P1				P2				P1			P6		P3		P5			P6										
time slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28

CPU 2																													
Process		P3						P2	P6			P2			P4	P7		P5		P4									
time slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28

CPU 3																													
Process					P1				P4							P6	P7												
time slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28

Memory States:

MEMORY CONTENT:
000: 00000-003ff - PID: 05 (idx 000, nxt: 001)
003e8: 15
001: 00400-007ff - PID: 05 (idx 001, nxt: -01)
002: 00800-00bff - PID: 05 (idx 000, nxt: 003)
003: 00c00-00fff - PID: 05 (idx 001, nxt: 004)
004: 01000-013ff - PID: 05 (idx 002, nxt: 005)
005: 01400-017ff - PID: 05 (idx 003, nxt: 006)
006: 01800-01bff - PID: 05 (idx 004, nxt: -01)
011: 02c00-02fff - PID: 06 (idx 000, nxt: 012)
012: 03000-033ff - PID: 06 (idx 001, nxt: 013)
013: 03400-037ff - PID: 06 (idx 002, nxt: 014)
014: 03800-03bff - PID: 06 (idx 003, nxt: -01)
019: 04c00-04fff - PID: 01 (idx 000, nxt: -01)
04c14: 64
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

CONCLUSION

This assignment provides us with knowledge about how a simple operating system works by implementing Scheduler and Virtual Memory Engine(VME) and enhancing our teamwork skills.

The combination of Scheduler and VME makes fundamental concepts of scheduling, synchronization, and memory management, allowing the CPU to run multiprocess with priority algorithms.

This report still has many points to improve upon and we look forward to receiving your feedback.