Simple Operating System

SCHEDULER

PRIORITY FEEDBACK QUEUE

As we all know, a class of scheduling algorithms has been created for situations in which processes are classified into different groups.

Processes have different response-time requirements => leads to the priority in processes.

A multilevel queue scheduling algorithm partitions the ready queue into several separate queues.

The processes are assigned to one queue based on property of the processes (memory size, process priority, process type).

- ⇨ Each queue has its own scheduling algorithm. In our case, the ready_queue (in sched.c) in the assignment is the priority-feedback queue and other queues might use another scheduling algorithms.

The multilevel feedback queue scheduling algorithm allows a process to move between queues. In case of the assignment 2, there are an interaction between the run_queue and ready_queue in sched.c.

As in the assignment said, if a process uses to much CPU time, it will be moved to a lower priority queue. So when the process in the ready_queue is out of its execution time, it will be moved to the run_queue which has the lower priority than ready_queue.

In addition, a process that waits too long in a lower-priority queue may be moved to higher-priority queue. In case of the assignment 2, we can see that if the process is finite, the ready_queue will soon be empty which make chance for the process in the run_queue to be executed.

So “priority feedback queue” is generally based on the multilevel feedback queue algorithm which means the process in the higher priority queue and the process that have higher priority than others will be executed first.

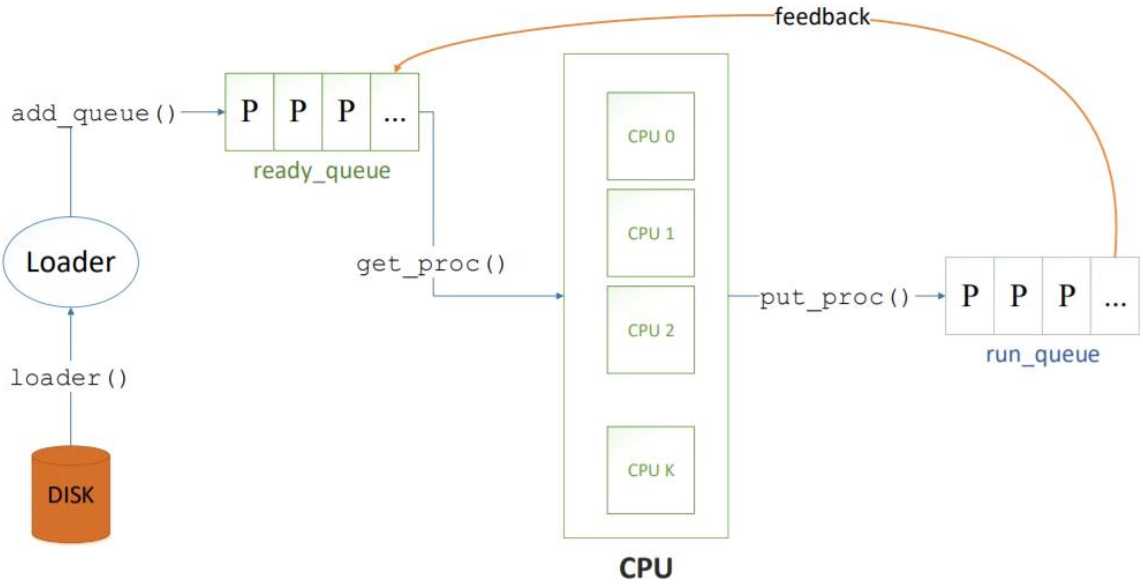
THE QUESTION: What is the advantage of using priority feedback queue in comparison with other scheduling algorithms ?

The advantages of priority scheduling:

- Easy to use scheduling method
- Processes are executed on the basis of priority so high priority does not need to wait for long which saves time

- This method provides a good mechanism where the relative important of each process may be precisely defined.
- Suitable for applications with fluctuating time and resource requirements.

BRIEF OPERATION OF THE SCHEDULER:



So as we can see here, the loader will create the PCB for the new process by load() function in load.c. Then a new process will be put the queue (ready_queue) by the enqueue() function in queue.c. After that, the get_proc() function will get a process from ready_queue. If the ready_queue is empty then push all the processes in the run queue back to ready_queue and return the highest priority one for the CPU to execute. Later on, if the process in the ready_queue runs overtime, put_proc() function will push the process in execution to the run_queue to wait. The dequeue() function is to get the highest priority process out of the queue.

Sched_0:

```
hoangtranminh@ubuntu:~/Downloads/source_code_assignment2_updated/source_code_assignment2$ make sched
gcc -Iinclude  -Wall -g obj/cpu.o obj/loader.o obj/mem.o obj/queue.o obj/os.o obj/sched.o obj/timer.o -o os -lpthread
hoangtranminh@ubuntu:~/Downloads/source_code_assignment2_updated/source_code_assignment2$ make test_sched
------ 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
```

Gantt chart :

CPU 0		P1				P2				P1		P2		P1		P2	P1						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

Sched_1

```
------ SCHEDULING TEST 1 ----------------------------------------
./os sched_1
Time slot   0
        Loaded a process at input/proc/s0, PID: 1
        CPU 0: Dispatched process  1
Time slot   1
Time slot   2
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   3
Time slot   4
        Loaded a process at input/proc/s1, PID: 2
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
Time slot   5
Time slot   6
        Loaded a process at input/proc/s2, PID: 3
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
Time slot   7
        Loaded a process at input/proc/s3, PID: 4
Time slot   8
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
Time slot   9
Time slot  10
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  2
Time slot  11
Time slot  12
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
Time slot  13
Time slot  14
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
Time slot  15
Time slot  16
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
Time slot  17
Time slot  18
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  2
```

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

```
Time slot  36
Time slot  37
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
Time slot  38
Time slot  39
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
Time slot  40
Time slot  41
        CPU 0: Processed  3 has finished
        CPU 0: Dispatched process  1
Time slot  42
Time slot  43
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
Time slot  44
        CPU 0: Processed  4 has finished
        CPU 0: Dispatched process  1
Time slot  45
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Gantt chart:

CPU 0		P1			P2		P3	P4				P2		P3		P1		P4		P2		P3	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

P3		P4		P2	P3		P1		P4		P3		P1		P4		P3		P1		P4	P1
24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46

MEMORY MANAGEMENT:

PAGED SEGMENTATION (SEGMENTATION WITH PAGING)

To take the advantage of both paging and segmentation, we can combine both of these this approaches. This approach is called paged segmentation or segmentation with paging. In this technique, segment is viewed as a collection of pages. Logical address generated by CPU is divided into 3 parts- the segment, the page and offset.

- The segment is used as an index in segment table. Entry in the segment table contains the base address of the page table that holds the pages belong to that segment.
- Page number is used as an index in page table and selects an entry within page table. Page table is used to store frame number of each page in physical memory.

This frame number is actually the base address of the page. This frame number + offset part of logical address forms the physical address. The physical address is the actual address in computer physical memory corresponding to the logical address generated by the CPU.

THE QUESTION (related to paged segmentation): What is the advantage and disadvantage of segmentation with paging.

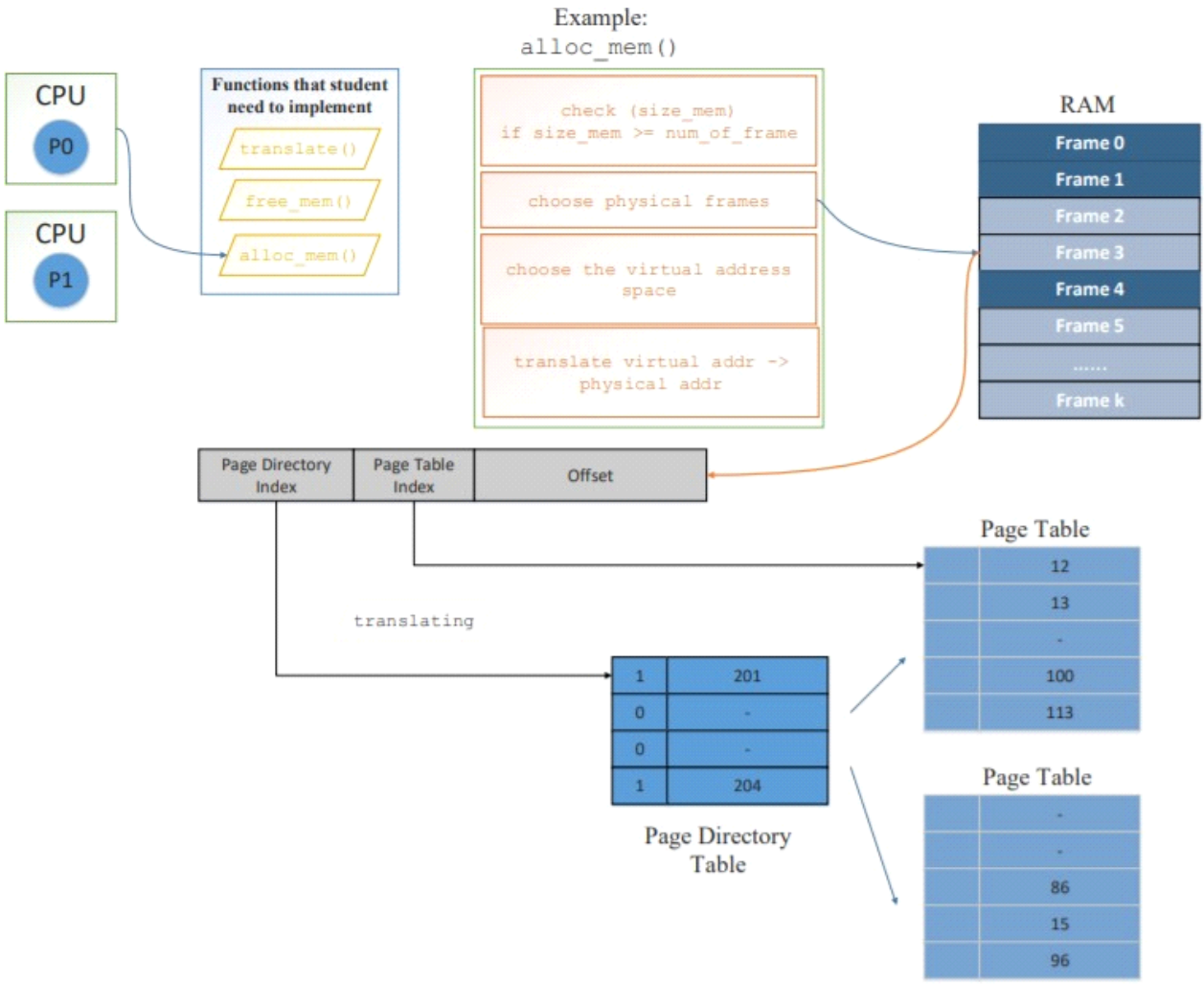
>The advantages:

- No external fragmentation
- Segment table has only one entry corresponding to one actual segment
- Page table size is limited by segment size
- Reduce memory usage.

• The disadvantages:

- Internal fragmentation
- Complexity level is higher paging and segmentation.

BRIEF OPERATION OF MEMORY MANAGEMENT:



So after, the process is ready for execution, it needs RAM. However, the operating system should not allocate the memory wastefully. The alloc_mem() function in mem.c will allocate the size in the memory for the process (proc) – the picked one and save the address of the first byte in the allocated memory region to ret_mem parameter. Later on, if we want to free or deallocate the memory space of the given process, the free_mem() function will release memory region allocated by the process (proc) by setting flag for physical page used by the memory block to zero unit to indicate that it is free. Then it remove the entries in segment table and page table of the given process which earlier created in alloc_mem() function. The interaction of the process and the RAM cannot be direct but through the virtual memory (in our case, the virtual memory engine uses Paged Segmentation mechanism). So the translate() function will transfer the virtual memory to physical memory as described before. And in order to translate, we have the get_page_table() function to check in all of the index of the page in the page table (first_lv or the segment part of the logical address) if there are pages that have the similar value to the given address. And the given address here in the translate() function is the (second_lv or the page table index of the logical address).

The

resu

lt:

M0:

```
000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
        00000: 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)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
NOTE: Read file output/m0 to verify your result
```