**VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY**

**HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY** FACULTY OF COMPUTER SCIENCE AND ENGINEERING



ASSIGNMENT REPORT

Group:

Course: OPERATING SYSTEMS

Code: CO2017

**Instructor - Supervisor:** PHAM HOANG ANH

HO CHI MINH CITY, MAY 11 2023

|  | Full name | Student ID | Contribution |
| --- | --- | --- | --- |
| 1 |  |  |  |
| 2 |  |  |  |
| 3 |  |  |  |

**TABLE OF CONTENTS**

[**I. Scheduling 3**](#_majyy85cc9nr)

[1. Question: What is the advantage of using priority queue in comparison with other scheduling algorithms you have learned? 3](#_uj9utvbogju7)

[2. Gantt Chart of this assignment’s scheduler 4](#_mucv3qw73thd)

[2.1. sched 4](#_l30yc5rg0054)

[2.2. sched\_0 6](#_8b9nw8939b06)

[2.2. sched\_1 8](#_hic4oetyi6ra)

[3. Code implementation 12](#_q772ygxubqry)

[3.1. Function: enqueue() and dequeue() 12](#_pzrvvkf6788f)

[**II. Memory 15**](#_xnsnl2ppes1f)

[1. Question: In this simple OS, we implement a design of multiple memory segments or memory areas in source code declaration. What is the advantage of the proposed design of multiple segments? 15](#_g8yikuclmzlg)

[2. Question: What will happen if we divide the address to more than 2-levels in the paging memory management system? 15](#_hqyv4puzlql5)

[3. Question: What is the advantage and disadvantage of segmentation with paging? 16](#_hsl4xsje6b1c)

[4. Code implementation 18](#_to38viwkiuxs)

[4.1. ALLOC 18](#_vakdzenqj93h)

[4.1.1 Function: inc\_vma\_limit() 19](#_ffug8eabr7es)

[4.1.2 Function: vm\_map\_ram() 20](#_qeb3og1b7h88)

[4.1.3 Function: alloc\_pages\_range() 20](#_ck8pjijrjl1l)

[4.1.4 Function: vmap\_page\_range() 21](#_ylke4baiepz)

[4.2. FREE 23](#_gthrbcjwrbrw)

[4.3. READ/WRITE 24](#_wjetf3awtcf2)

[5. Status of RAM after each memory allocation and deallocation 27](#_me195t6tg5cp)

[6. Synchronization problem: 27](#_8dyfk1aymxtq)

[**III. Overall 29**](#_xf0prfrrcck)

# 

# 

# **I. Scheduling**

## 1. Question: *What is the advantage of using priority queue in comparison with other scheduling algorithms you have learned?*

Priority queue, as known as multi-level queue, is an algorithm that has separate queues for each distinct priority, and in the highest-priority queue, we can use any algorithm to select a process. In this project, we combine priority queue with priority scheduling algorithm to determine the highest-priority process.

Here are other scheduling algorithms learned:

* First Come First Served (FCFS)
* Shortest Job First (SJF)
* Shortest Remaining Time First (SRTF)
* Priority Scheduling (PS)
* Round Robin (RR)

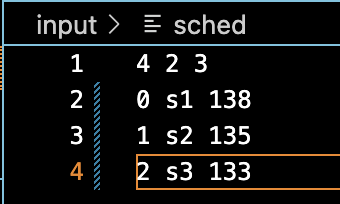
Advantages of the Priority Queue Algorithm:

* Using the time slot, with the idea of the RR algorithm with a quantum time interval, creating fairness in terms of execution time between processes, avoiding CPU usage, delaying indefinitely.
* Using multiple queues, where queues cycle through processes until the process is completed, increasing the response time for processes (the processes with lower priority that come later can still be executed before the processes with higher priority after finishing the slot). Besides that, here in this project, each queue also has a fixed slot that can avoid starvation. When the slot is used up, the system must change the resource to the other process in the next queue and leave the remaining work for the future slot even though it needs a completed round of the ready queue.
* Fairness between processes is guaranteed. For example, in the beginning, the CPU will choose the highest-priority queue, if that queue is not empty, then get a process from it and dispatch. If not then go to lower-priority queue until having an available process. If after executing a process from a highest-priority queue, the loader adds more processes to that queue, so the highest-priority queue will never be empty, and it will always run first. That situation causes starvation for other queues. Slots will solve that problem. After executing a time slot for a process, the slot of that queue will decrease. If there is no process left in that queue but the slot is still available, just jump to another queue. If there are still some processes in the queue but the slot = 0, CPU will jump to the lower-priority queue to pick a process. Until the CPU has gone through all the queues, then each slot of each queue will be renewed. The combination between priority queue, priority scheduling and remaining slot creates the fairness and the optimization for the scheduling problem.

## 2. Gantt Chart of this assignment’s scheduler

We will draw 3 Gantt charts that represent the workflow covering the following tests: sched, sched\_0, sched\_1. The implemented scheduling algorithm is the aforementioned priority queue combined with priority scheduling algorithm.

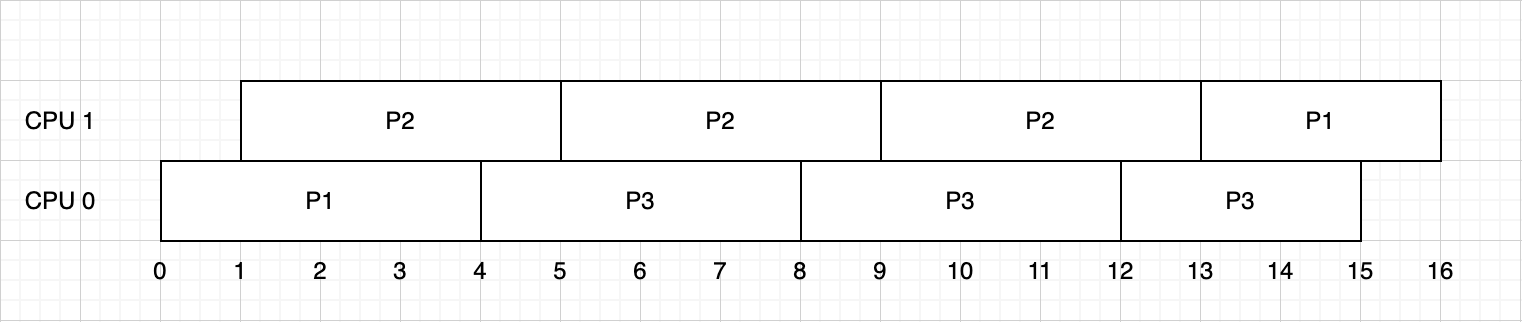
### 2.1. sched



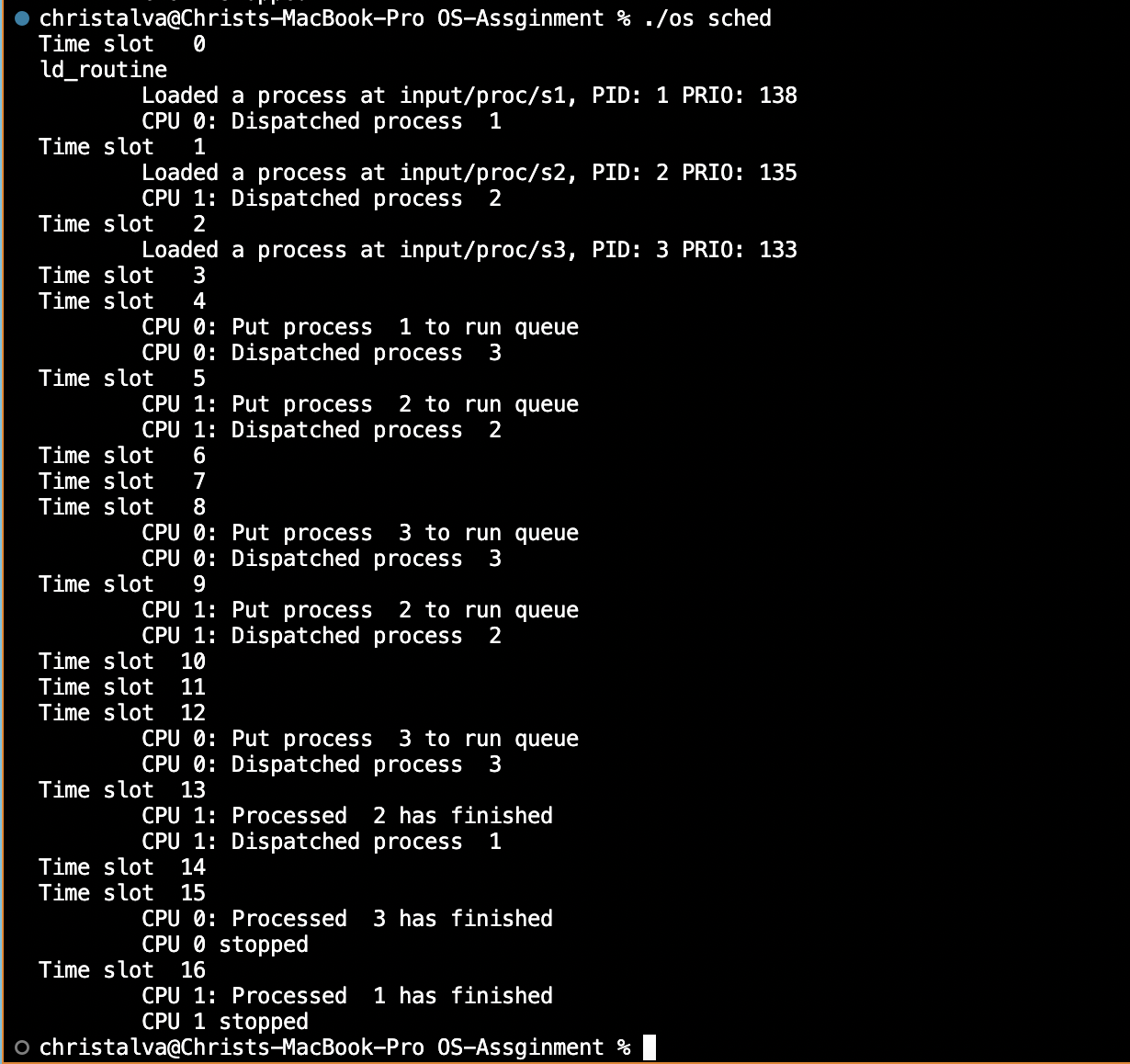
The processes of sched are summarized in the following table:

| Process | Arrival Time | Burst Time | Priority | Priority queue |
| --- | --- | --- | --- | --- |
| s1 | 0 | 7 | 20 | 138 |
| s2 | 1 | 12 | 20 | 135 |
| s3 | 2 | 11 | 7 | 133 |

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



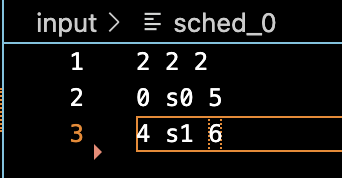
Result from console:



Explanation:

* At t = 0, proc proc 1 is loaded and because CPU0 is free so CPU will dispatch proc 1 immediately (either CPU0 or CPU1).
* Then proc proc 2 arrives at t = 1. At t = 1, CPU0 is busy with proc 1, but CPU1 is free, so CPU1 will dispatch proc 2.
* At t = 3, proc proc 3 arrives. Timeslot of this test is 4, so none of the CPUs are available, so proc 3 has to wait.
* At t = 4, CPU0 finishes, so CPU0 will continue to dispatch proc 3 because proc 3 has higher priority than proc 1.
* At t = 5, CPU1 finishes, and it continues to dispatch proc 2 because proc 2 has higher priority than proc 1.
* The routine will repeat like that till t = 13. At that time, the burst time of proc 2 = 0, only proc 1 is available so CPU1 dispatches proc 1.
* At t = 15, the burst time of proc 3 = 0, CPU0 stops.
* At t = 16, the burst time of proc 1 = 0, CPU1 stops.

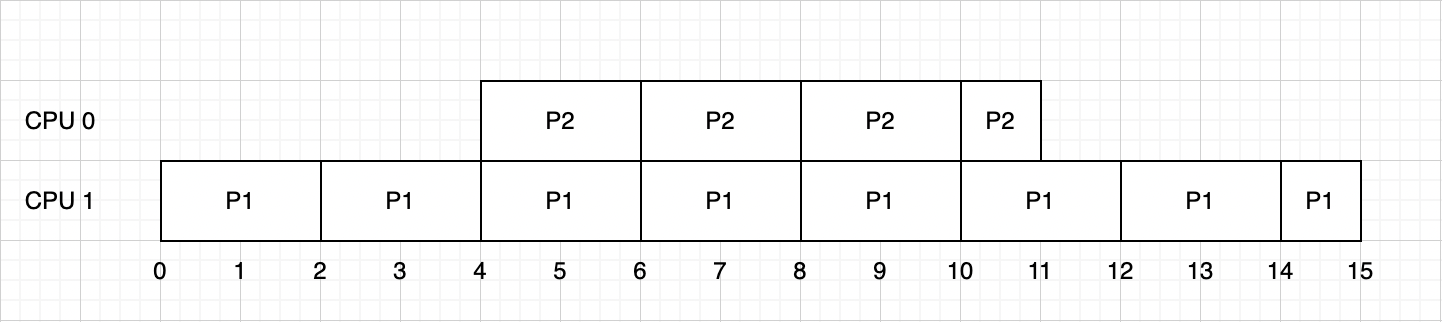
### **2.2. sched\_0**



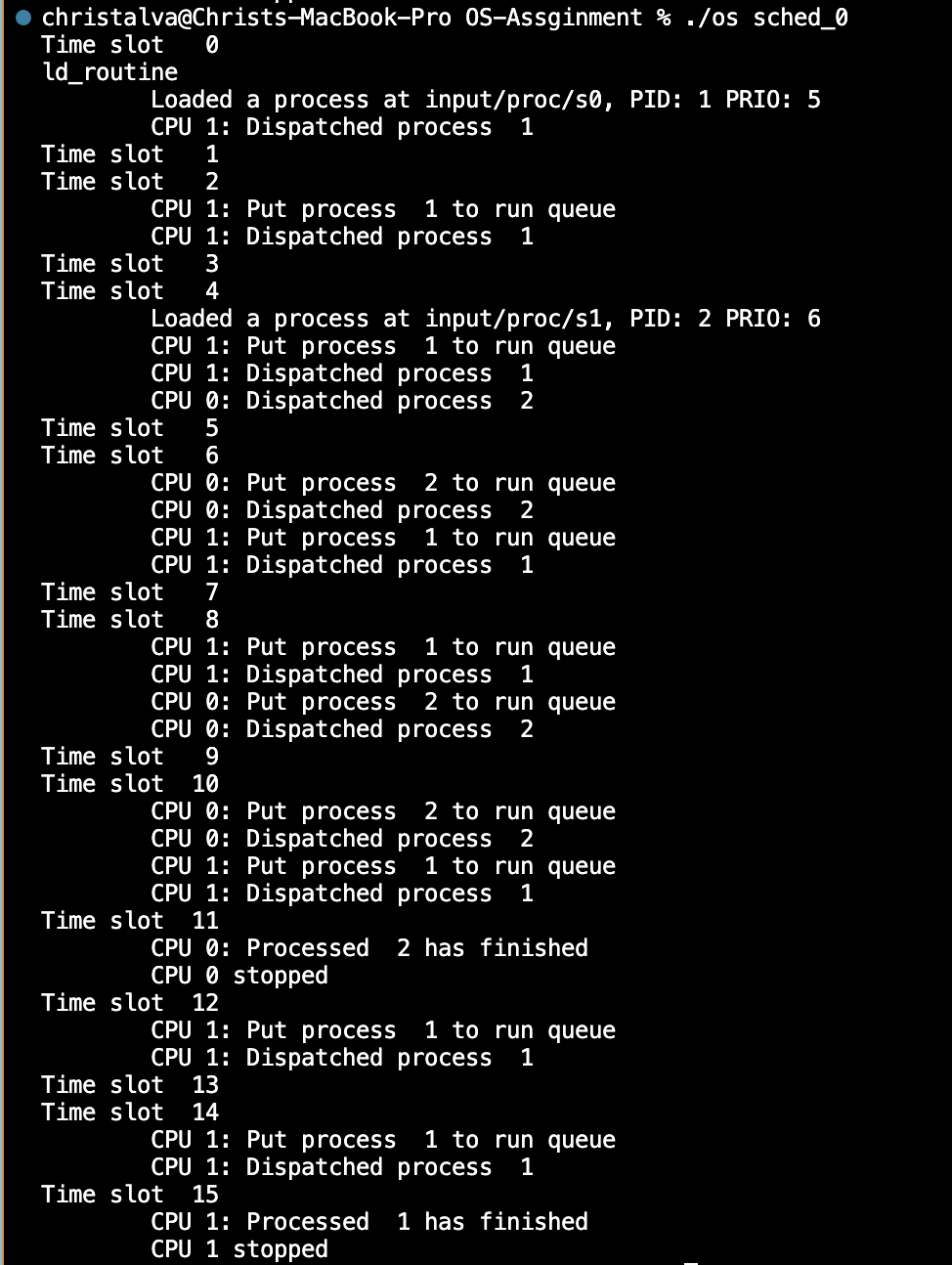
The processes of sched are summarized in the following table:

| Process | Arrival Time | Burst Time | Priority | Priority queue |
| --- | --- | --- | --- | --- |
| s0 | 0 | 15 | 12 | 5 |
| s1 | 4 | 7 | 20 | 6 |

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



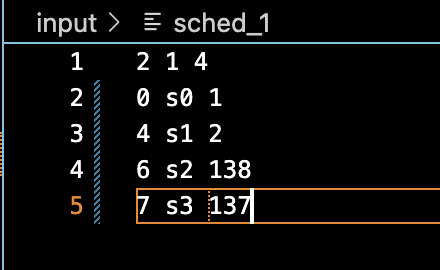
Result from console:



Explanation:

* At t = 0, proc 1 is loaded and because CPU0 is free so CPU will dispatch proc 1 immediately (either CPU0 or CPU1).
* At t = 4, proc 2 arrives. At this time, both proc 1 and proc 2 are available and CPU0, CPU1 are free, so CPU0 dispatch proc 2, CPU1 dispatch proc 1 (or vice versa)
* The routine will repeat like that till t = 11. At that time, the burst time of proc 2 = 0, CPU0 stops. CPU1 continues to execute.
* At t = 15, the burst time of proc 1 = 0, CPU1 stops.

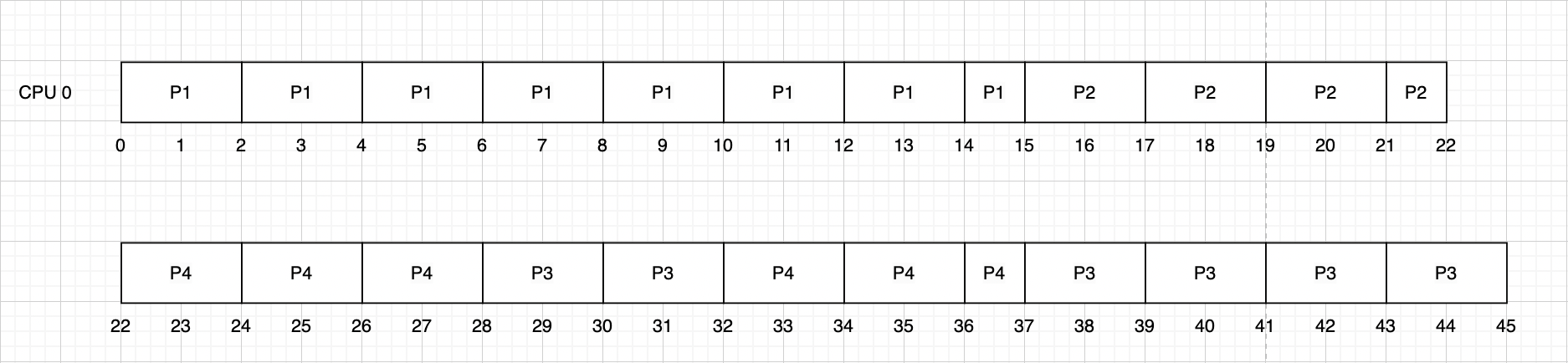
### **2.2. sched\_1**



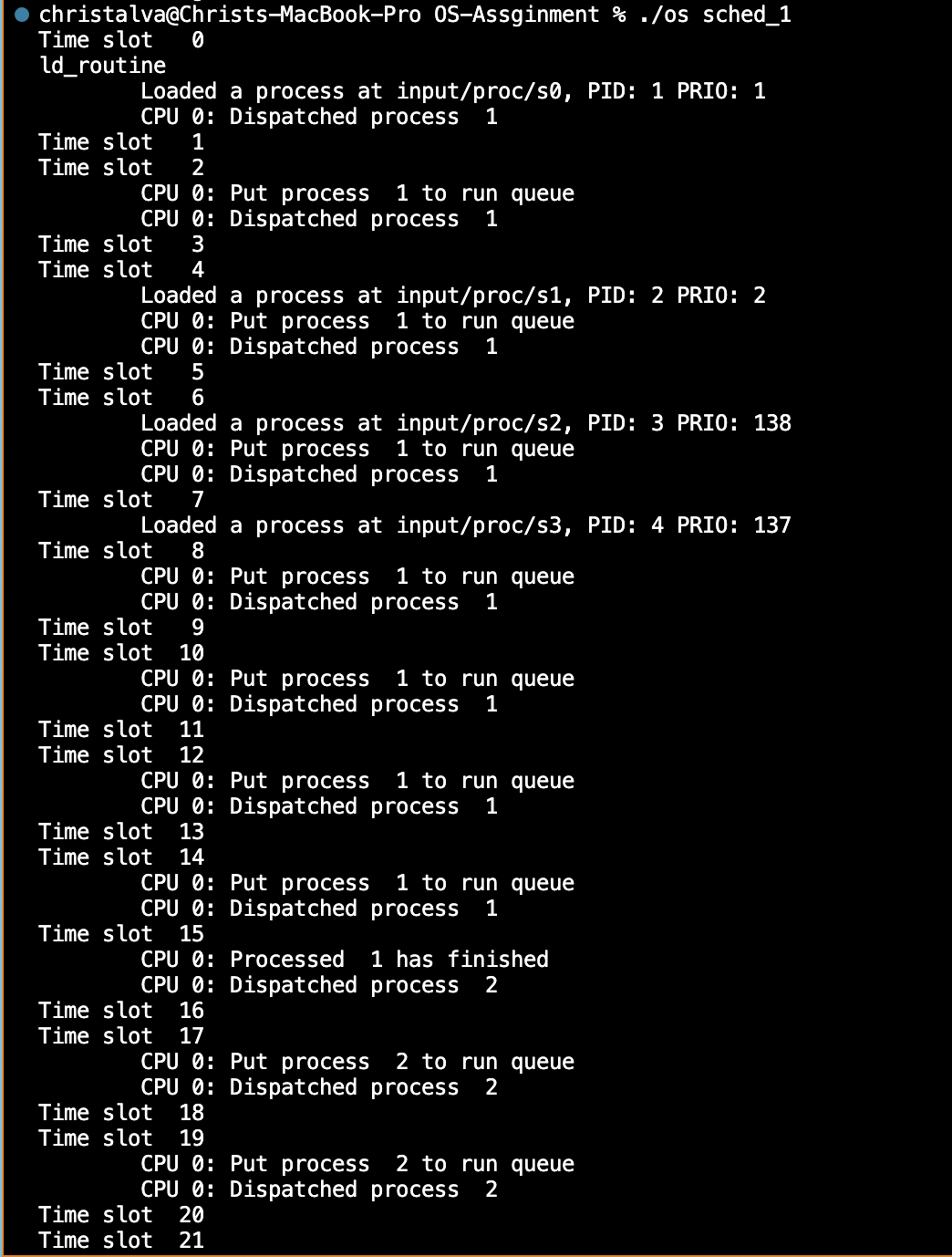
The processes of sched are summarized in the following table:

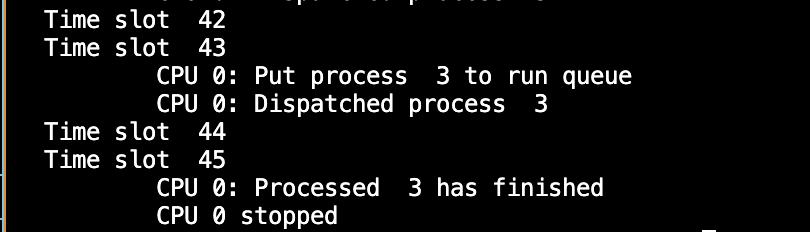
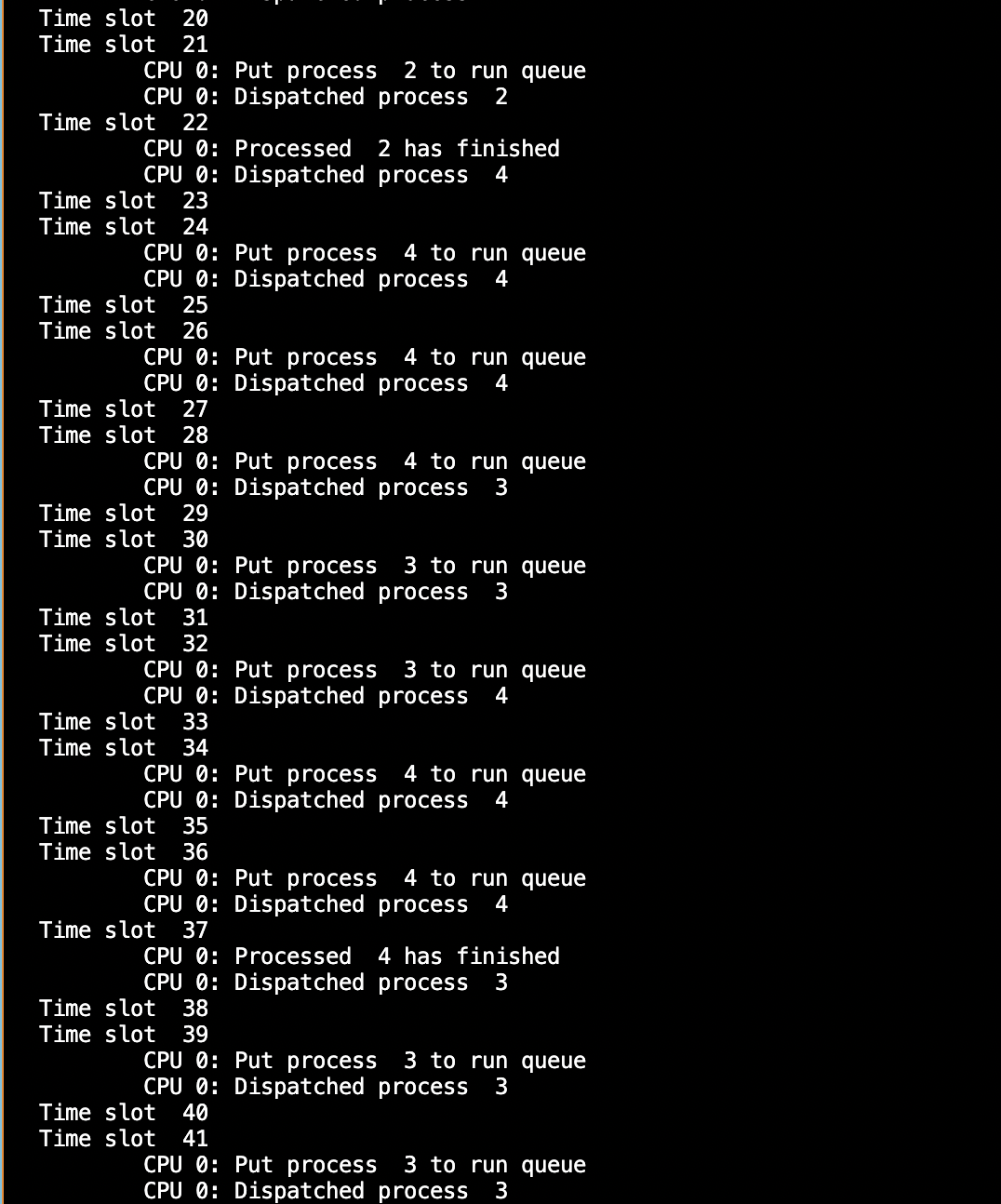
| Process | Arrival Time | Burst Time | Priority | Priority queue |
| --- | --- | --- | --- | --- |
| s0 | 0 | 15 | 12 | 1 |
| s1 | 4 | 7 | 20 | 2 |
| s2 | 6 | 12 | 20 | 138 |
| s3 | 7 | 11 | 7 | 137 |

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



Result from the console:





Explanation:

* At t=0, proc 1 is loaded and because CPU0 is free so CPU will dispatch proc 1 immediately.
* At t=4, proc 2 arrives. CPU0 continues to dispatch proc 1 because proc 1 has higher priority than proc 2.
* At t=6, proc 3 arrives. CPU0 continues to dispatch proc 1 because proc 1 has higher priority than proc 2 and proc 3.
* At t=7, proc 4 arrives. CPU0 continues to dispatch proc 1 because proc 1 has the highest priority.
* The routine will repeat like that till t = 15. At that time, the burst time of proc 1 = 0. CPU0 continues to dispatch proc 2 because proc 2 has the highest priority.
* The routine will repeat like that till t = 22. At that time, the burst time of proc 2 = 0. There are 2 processes left, proc 3 and proc 4. CPU0 continues to dispatch proc 4 because proc 4 has higher priority than proc 3.
* Because the slot of proc 4 = 140 - 137 = 3, proc 4 can only be dispatched and executed 3 times. After that, it will stop and let proc 3 execute.
* At t = 28, proc 4 runs out of slots, so CPU0 will dispatch proc 3 to execute.
* The slot of proc 3 = 140 - 138 = 2, so it can only be dispatched and executed 2 times. After that, it will stop and let proc 4 execute.
* At t = 32, proc 3 runs out of slots, so CPU0 will dispatch proc 4 to execute.
* At t = 37, the burst time of proc 4 = 0 and proc 4 still has the available slot. CPU0 continues to dispatch proc 3.
* Routine repeats till the end. At t = 45, the burst time of proc 3 = 0, CPU0 stops.

## **3. Code implementation**

In this part, we implement 3 functions: enqueue(), dequeue(), get\_mlq\_proc().

### 3.1. Function: enqueue() and dequeue()

Code file: queue.c

Explanation:

* In the enqueue function, we add a new proc in the array of that queue at index q->size. Then we increase size by 1.
* In the dequeue function, if the queue is empty, return NULL. If not, then we will find the process that has the highest priority in the queue and remove it. First, we set min as the priority of the first process. We use a for loop, if the priority of any queue is smaller than min then assign min to it. Create temp and assign to the proc at index min and return it at the end. All the processes in the queue accumulate to the previous index. Finally, decrease the size by 1.

void enqueue(struct queue\_t \* q, struct pcb\_t \* proc) {

/\* TODO: put a new process to queue [q] \*/

q->proc[q->size]=proc;

q->size++;

}

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

int min=0;

for (int i=1;i<q->size;i++)

{

if (q->proc[min]->priority > q->proc[i]->priority)

{

min=i;

}

}

struct pcb\_t\* temp=q->proc[min];

for (int i=min;i<q->size-1;i++)

{

q->proc[i]=q->proc[i+1];

}

q->size--;

return temp;

}

Function: get\_mlq\_proc()

Code file: sched.c

Explanation:

* In get\_mlq\_proc(), we will use priority queue and priority scheduling to get a process out of the queue. First, we use mutex lock to make sure that other CPUs can get a process from a queue only when this CPU has already got the process. Create proc=NULL. Using a for loop, run from 0 to the number of queues (MAX\_PRIO), the lower the index is, the higher the priority is. We check from the highest-priority queue, if it is empty then go to the lower one. If the queue is not empty and has an available slot to run, then dequeue that queue, decrease the slot of that queue by 1 and break the loop.
* If we have gone through all the queues but cannot get a process, there are 2 reasons. First, all the queues are empty or second, there is still a queue that is not empty, but doesn't have the available slot to run. When we have gone through all the queues, the index = MAX\_PRIO. If the index = MAX\_PRIO, we will check that condition. First, having a flag\_queue = 0. Using a for loop with prio run from 0 to , set the slot of the queue again, slot = MAX\_PRIO - prio. After setting, check again if that queue is empty or not. If not, then dequeue it, and assign flag\_queue = 1. This flag\_queue is to notify that a process has been dequeued, so all the after queues don’t dequeue anymore. Finally, return proc.

struct pcb\_t \* get\_mlq\_proc(void) {

struct pcb\_t \* proc = NULL;

/\*TODO: get a process from PRIORITY [ready\_queue].

\* Remember to use lock to protect the queue.

\* \*/

pthread\_mutex\_lock(&queue\_lock);

int index;

for (index=0; index<MAX\_PRIO; index++)

{

if (!empty(&mlq\_ready\_queue[index]) && (mlq\_ready\_queue[index].slot > 0))

{

proc = dequeue(&mlq\_ready\_queue[index]);

mlq\_ready\_queue[index].slot--;

break;

}

}

int flag\_dequeue=0;

if (index == MAX\_PRIO) // has gone through all the queues but all are empty

{

for (int prio=0; prio<MAX\_PRIO; prio++)

{

mlq\_ready\_queue[prio].slot = MAX\_PRIO - prio;

if (!empty(&mlq\_ready\_queue[prio]) && (flag\_dequeue == 0))

{

proc=dequeue(&mlq\_ready\_queue[prio]);

flag\_dequeue = 1;

}

}

}

pthread\_mutex\_unlock(&queue\_lock);

return proc;

}

# **II. Memory**

## **1. Question:** *In this simple OS, we implement a design of multiple memory segments or memory areas in source code declaration. What is the advantage of the proposed design of multiple segments?*

Segmentation in an operating system is a memory management technique that divides the memory into variable-sized segments. Each segment can contain different types of data, such as code, stack, heap, global variable, etc. Segmentation requires a segment table that stores the base address and the limit of each segment in the process’s address space. In our case:

* The struct vm\_area\_struct represents the segmentation table.
* The struct vm\_rg\_struct represents the segments, with its members rg\_start and rg\_end represent the start and end address of the segment respectively.

Some advantages of the proposed design of multiple segmentsare:

* It provides a powerful memory management mechanism that allows the user to divide the program into modules and map them to physical memory.
* It allows for sharing of memory segments between processes, which can be useful for inter-process communication or for sharing code libraries.
* It provides a level of protection between segments, preventing one process from accessing or modifying another process's memory segment.
* It reduces internal fragmentation.

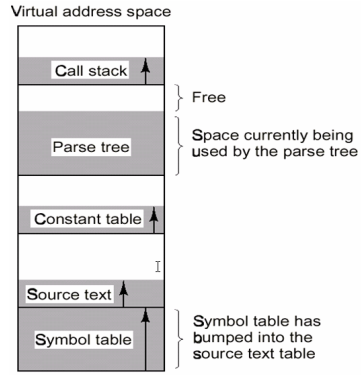
## 2. **Question:** *What will happen if we divide the address to more than 2-levels in the paging memory management system?*

Dividing the CPU address to more than 2-levels in the paging memory management system is called multi-level paging. It is a technique that uses a hierarchy of page tables to reduce the size of the page table and the memory requirements. Multi-level paging is needed when the page table is too large to fit in a single frame of physical memory.

Dividing the address into more than 2 levels in a paging memory management system helps save memory space and supports large virtual address spaces. However, it also increases the complexity and overhead of the memory management process, and requires more memory access to get the physical address.

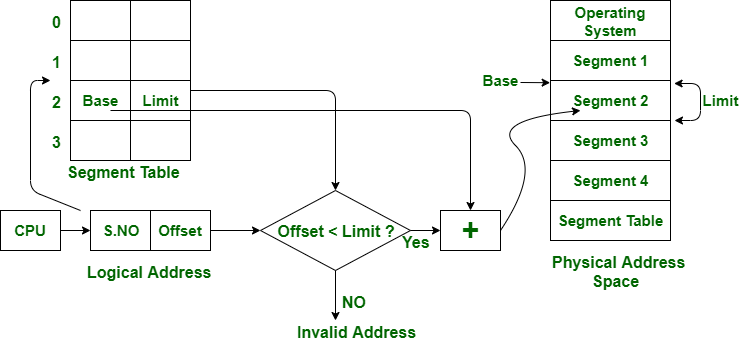
## 3. **Question**: *What is the advantage and disadvantage of segmentation with paging?*

Paging is a memory allocation mechanism that allocates non-contiguous memory. External fragmentation can be avoided by paging; nevertheless, internal fragmentation might occur if all frame memory is not utilized up. In addition, programs are frequently structured into multiple segments for ease of management: code segment, data segment, heap segment, stack segment, and so on. Although the virtual area is very big with paging, there are several constraints if the programmer wants to structure the program into segments as described above:



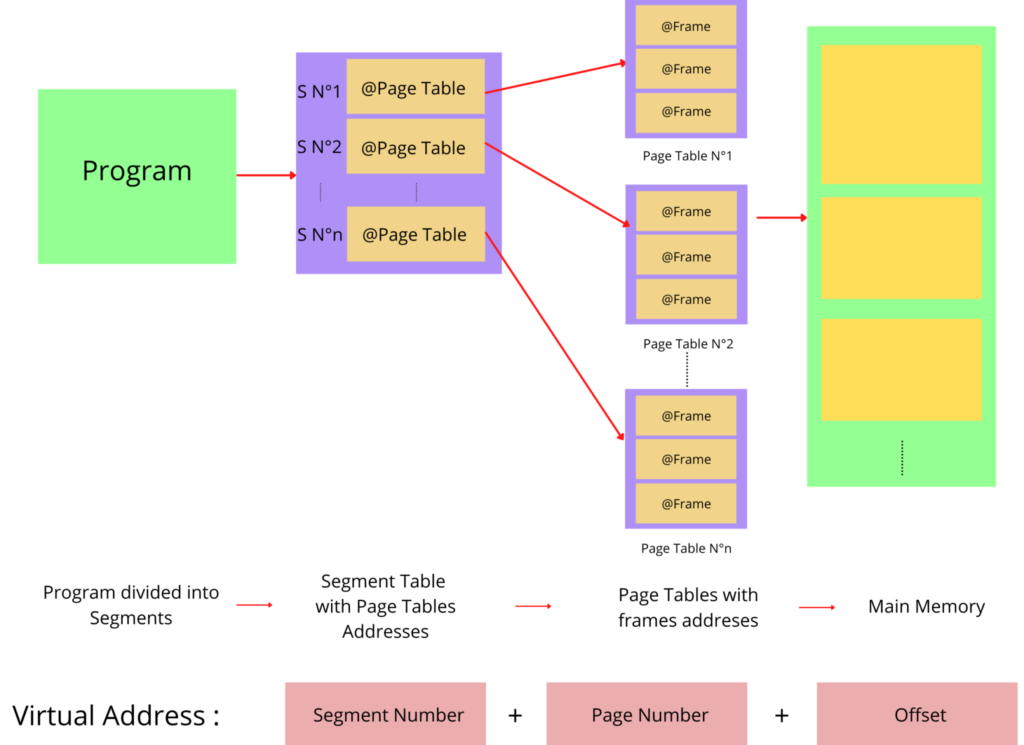
We can observe from the sample above that some segments are out of storage and overlap with other segments.

Segmentation, on the other hand, can solve this problem. Segmentation divides each process's virtual space into multiple segments and maps each segment to its corresponding segment in physical memory.



External fragmentation, on the contrary, might cause segmentation problems.

As a result, another method that combines paging with segmentation yields a highly strong method for memory management. Generally, each program can be divided into many segment and each segment is divided into a fixed-size pages:



Previously, under the paging technique, each process had its own page table, and the size of the page table depended on each system, which may grow significantly if the virtual area is large. Despite its huge size, as described above, the program is broken into several segments, some of which are not used entirely, resulting in useless page table entries.

Segmentation with paging can easily lower the size of the page table because each segment can limit the size of its own page table. Also, in a program, there will be some segments that are not commonly utilized; therefore, the OS designer can store the page table for those infrequently used segments in secondary memory storage reserved some space in the main memory.

To summarize, segmentation with paging has the following advantages:

* Reduce the size of the page table as well as the amount of storage required to hold the page table in main memory.
* Reduces external fragmentation in comparison with segmentation.
* Provides programmer views of memory being divided into several pieces.

Disadvantages of segmentation with paging:

* Internal fragmentation still remains.
* Because the translation is sequential, the memory access time increases, resulting in greater complexity.
* External fragmentation takes place as a result of variable page table and segment table sizes.

## **4. Code implementation**

In the memory section, we have to implement several functions to support ALLOC, FREE and READ/WRITE operations of each process.

### **4.1. ALLOC**

Instructions’s syntax: alloc [size] [reg]

In order to perform an ALLOC operation, we must first locate a free memory location in virtual memory. If there is no free region, virtual memory should be allocated new pages, which will be mapped to the real frame via the page table. Diagram 1 depicts the detailed function call made when the ALLOC action is performed:

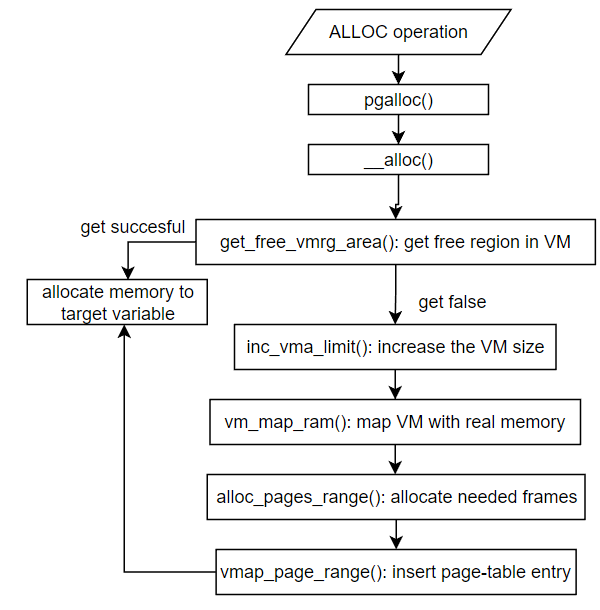


Diagram 1: ALLOC operation

In this part, we implement 4 functions: inc\_vma\_limit(), vm\_map\_ram(), alloc\_pages\_range() and vmap\_page\_range().

#### **4.1.1 Function: inc\_vma\_limit()**

Code file: mm-vm.c

Explanation:

* The virtual memory area of each process is first initialized using the function init\_mm(), with the size of the VM set to 0. As a result, if each process runs out of free VM regions and requires more pages, we will allocate exactly the number of pages required.
* In this section, we kept the source code but changed it slightly by making the sbrk always equal to vm\_end, as we stated earlier that we will use all of the VM space.

Code detail:

| /\*inc\_vma\_limit - increase vm area limits to reserve space for new \*/ int inc\_vma\_limit(struct pcb\_t \*caller, int vmaid, int inc\_sz){  ...  cur\_vma->sbrk = cur\_vma->vm\_end;  ... } |
| --- |

#### 4.1.2 Function: vm\_map\_ram()

Code file: mm.c

Explanation:

For this function, we add an int array frm\_loc and pass as an argument to function alloc\_pages\_range() and vmap\_page\_range() (the usage of this array will be explained later).

Code detail:

| /\* vm\_map\_ram - do the mapping all vm are to ram storage device \*/ int vm\_map\_ram(struct pcb\_t \*caller, int astart, int aend, int mapstart, int incpgnum, struct vm\_rg\_struct \*ret\_rg){  struct framephy\_struct \*frm\_lst = NULL;  int \*frm\_loc = malloc(incpgnum\*sizeof(int));  int ret\_alloc;  ret\_alloc = alloc\_pages\_range(caller, incpgnum, &frm\_lst, frm\_loc);  ...  vmap\_page\_range(caller, mapstart, incpgnum, frm\_lst, ret\_rg, frm\_loc);  return 0; } |
| --- |

#### 4.1.3 Function: alloc\_pages\_range()

Code file: mm.c

Explanation:

* To get the free frame from physical memory to map with the newly allocated pages in the VM, we use the technique of first getting the frame from main memory and then getting the free frame from the secondary memory device if there is no free frame in main memory. If there is no free frame in both devices raise an ERROR.
* We do not swap used frames from main memory to secondary memory because if the RAM is little and the number of needed frames is too great, we may end up swapping newly allocated frames, which is inefficient. As a result, we will leave the swapping of the frame between main memory and secondary memory for READ and WRITE operations.
* The frm\_loc array is used to keep track of whether the frame is in RAM or SWAP device.

Code detail:

| /\* alloc\_pages\_range - allocate req\_pgnum of frame in ram \*/ int alloc\_pages\_range(struct pcb\_t \*caller, int req\_pgnum, struct framephy\_struct\*\* frm\_lst, int\* frm\_loc){  int pgit, fpn, swpfpn;   for(pgit = 0; pgit < req\_pgnum; pgit++)  {  struct framephy\_struct \*newfp = malloc(sizeof(struct framephy\_struct));  if(MEMPHY\_get\_freefp(caller->mram, &fpn) == 0)  {  newfp->fpn = fpn;  newfp->fp\_next = \*frm\_lst;  newfp->owner = caller->mm;  \*frm\_lst = newfp;  frm\_loc[req\_pgnum - pgit - 1] = 0;  }   else   { // ERROR CODE of obtaining somes but not enough frames  if (MEMPHY\_get\_freefp(caller->active\_mswp, &swpfpn) == 0)  {  newfp->fpn = swpfpn;  newfp->fp\_next = \*frm\_lst;  newfp->owner = caller->mm;  \*frm\_lst = newfp;  frm\_loc[req\_pgnum - pgit - 1] = 1;  }  else  return -3000;  }   }  return 0; } |
| --- |

#### 4.1.4 Function: vmap\_page\_range()

Code file: mm.c

Explanation:

* Finally we will use this function to change the page table for mapping between each VM’s pages and physical memory’s frames.
* In this section, we will change the content of each corresponding entry. In addition, we will add these newly mapped frames to the used frame list in physical memory for later page replacement activities. And we added a new attribute to the framephy\_struct called pgn to control which page table entry is holding this frame so that when we swap the frame between RAM and SWAP device, we can easily modify the content of this page table entry.

Code detail:

| /\* vmap\_page\_range - map a range of page at aligned address \*/ int vmap\_page\_range(struct pcb\_t \*caller, // process call  int addr, // start address which is aligned to pagesz  int pgnum, // num of mapping page  struct framephy\_struct \*frames, // list of the mapped frames  struct vm\_rg\_struct \*ret\_rg, // return mapped region, the real mapped fp   // no guarantee all given pages are mapped  int\* frm\_loc) // frame location list ( 0 : in RAM or 1 : SWAP memory){   struct framephy\_struct \*fpit = frames;  int pgit;  int pgn = PAGING\_PGN(addr);  ret\_rg->rg\_end = ret\_rg->rg\_start = addr; // at least the very first space is usable  for (pgit = 0; pgit < pgnum; pgit++)  {  struct framephy\_struct \*temp = fpit; // Free allocate mem for mapped frames list  uint32\_t \*pte = malloc(sizeof(uint32\_t));  \*pte = 0;   if (frm\_loc[pgit] == 0) init\_pte(pte,   1, // Presented  fpit->fpn, // FPN  0, // No dirty usage  0, // No swap  0, // Swap type  0); // Swap offset  else init\_pte(pte,   1, // Presented  0, // FPN  0, // No dirty usage  1, // Swap  0, // Swap type  fpit->fpn); // Swap offset   caller->mm->pgd[pgn+pgit] = \*pte;  fpit = fpit->fp\_next;    /\* Tracking for later page replacement activities (if needed) \*/  temp->pgn = pgn+pgit;   // Put new frame to used frame list  if (frm\_loc[pgit] == 0) MEMPHY\_put\_usefp(caller->mram, temp);  else MEMPHY\_put\_usefp(caller->active\_mswp, temp);    free(temp);   free(pte);   }  free(frm\_loc);   ret\_rg->rg\_end = addr + pgnum\*PAGING\_PAGESZ;  return 0; } |
| --- |

### 4.2. FREE

Instructions’s syntax: free [reg]

In order to implement FREE operation, we modified the function \_\_free(). To begin, we will obtain the target variable's virtual memory region, then add it to the process's virtual memory free region list, and finally set the variable's rg\_start and rg\_end to -1 to indicate that it has not been allocated.

Code detail:

| int \_\_free(struct pcb\_t \*caller, int vmaid, int rgid) {  struct vm\_rg\_struct\* rgnode;   if(rgid < 0 || rgid > PAGING\_MAX\_SYMTBL\_SZ)  return -1;   /\* **TODO:** Manage the collect freed region to freerg\_list \*/  struct vm\_area\_struct \*rgarea = get\_vma\_by\_num(caller->mm, vmaid);  rgnode = get\_symrg\_byid(rgarea->vm\_mm, rgid, 2);   if(rgnode == NULL) return -1;   /\* create a new rgnode to add to the free region in the vm \*/  struct vm\_rg\_struct\* new\_rgnode = init\_vm\_rg(rgnode->rg\_start, rgnode->rg\_end);   /\* Set region start end to -1 indicate not allocated \*/  rgnode->rg\_start = rgnode->rg\_end = -1;    /\* Enlist the obsoleted memory region \*/  enlist\_vm\_rg\_node(&(rgarea->vm\_freerg\_list), new\_rgnode);  return 0; } |
| --- |

### 4.3. READ/WRITE

Instructions’s syntax of READ and WRITE:

* read [source] [offset] [destination]
* write [data] [destination] [offset]

In order to perform READ/WRITE operation, we must find the corresponding frame number of the page in RAM through the pg\_getpage() function and then perform read or write on RAM given the frame number. Diagram 2 depicts the detailed function call made when READ/WRITE action is performed. In this section, we need to modify the function pg\_getpage(). This is a function used to get the mapping frame number in RAM given the page number in virtual memory. In this function we perform the following step:

1. Get the page table entry from the page table with the given page number.
2. Check the present bit of the pte:

- If (present bit == 0) meaning the page is not allocated: raise ERROR.

- Else: continue the next step.

1. Check the swap bit of the pte:

- If (swap bit == 0) meaning the page is on RAM: continue to the last step.

- Else: perform place replacement algorithm to swap frame from SWAP device to RAM device.

## 

Diagram 2: READ/WRITE operation

1. Find free frame (victim frame) in the memory:

- If there is no free frame: we will use the FIFO algorithm to retrieve the used frame from the used\_fp\_list, which contains a list of all frames that have been used.

- Else: continue the next step.

1. Find a free frame from SWAP and copy the victim frame to this frame.
2. Copy target frame from SWAP to victime frame in RAM.
3. Put the target frame in SWAP to the list of free frames in SWAP.
4. Update the pte of the victim page as offline status (indicate that mapped frame is stored in SWAP device).
5. Update online status (indicate that mapped frame is stored in RAM device) of the target page.
6. Get and return the frame number from the target pte.

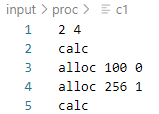
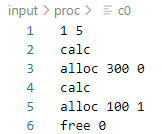
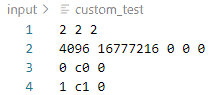
Code detail:

| /\*pg\_getpage - get the page in ram \*/ int pg\_getpage(struct mm\_struct \*mm, int pgn, int \*fpn, struct pcb\_t \*caller) {    uint32\_t pte = mm->pgd[pgn];    if (!PAGING\_PAGE\_PRESENT(pte)) {printf("Attempt to access inaccessible memory space!!!\n"); return -1;}  if(PAGING\_PTE\_SWP(pte)){  /\* Page is not online, make it actively living \*/  int vicfpn, swpfpn;   int tgtfpn = PAGING\_PTE\_SWPOFF(pte); //the target frame storing our variable  struct framephy\_struct \*retfp = NULL;  uint32\_t vicpte = 0;    /\* **TODO:** Play with your paging theory here \*/  /\* Find free frame in memory \*/  if(MEMPHY\_get\_freefp(caller->mram, &vicfpn) == -1){  /\* There are no free frame in the ram \*/  /\* Find victim page \*/  MEMPHY\_get\_usefp(caller->mram, &retfp);  vicfpn = retfp->fpn;  }   /\* Get free frame in MEMSWP \*/  MEMPHY\_get\_freefp(caller->active\_mswp, &swpfpn);  /\* Do swap frame from MEMRAM to MEMSWP and vice versa\*/  /\* Copy victim frame to swap \*/  \_\_swap\_cp\_page(caller->mram, vicfpn, caller->active\_mswp, swpfpn);  /\* Copy target frame from swap to mem \*/  \_\_swap\_cp\_page(caller->active\_mswp, tgtfpn, caller->mram, vicfpn);  /\* Put tgtfpn to free frame \*/  MEMPHY\_put\_freefp(caller->active\_mswp, tgtfpn);   /\* Update pte of victim page \*/  if(retfp != NULL){  pte\_set\_swap(&vicpte, 0, swpfpn);   retfp->owner->pgd[retfp->pgn] = vicpte;  }    /\* Update its online status of the target page \*/  pte\_set\_fpn(&pte, vicfpn);  mm->pgd[pgn] = pte;   if(retfp != NULL){  retfp->owner = mm;  retfp->pgn = pgn;  MEMPHY\_put\_usefp(caller->mram, retfp);  free(retfp);  }  //enlist\_pgn\_node(&caller->mm->fifo\_pgn,pgn);  }     \*fpn = PAGING\_PTE\_FPN(pte);    return 0; } |
| --- |

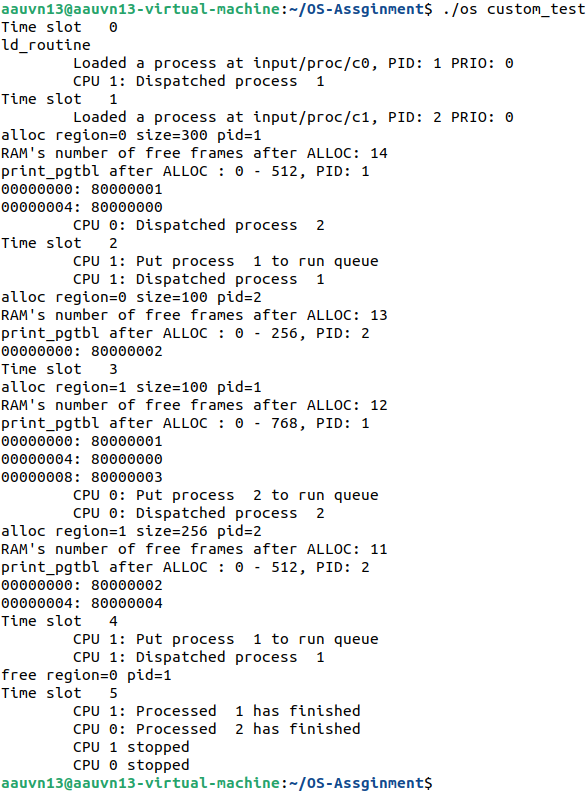
## 5. Status of RAM after each memory allocation and deallocation

*Example 1:*

We consider this input, where RAM is initialized with 4096 bytes (or 16 free frames), and SWP is initialized with 16 mb, with two processes c0 and c1.



Running the input, we obtain the following results:

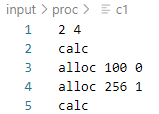
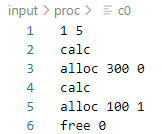
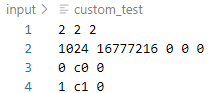


Explanation for every memory allocation and deallocation call in execution order:

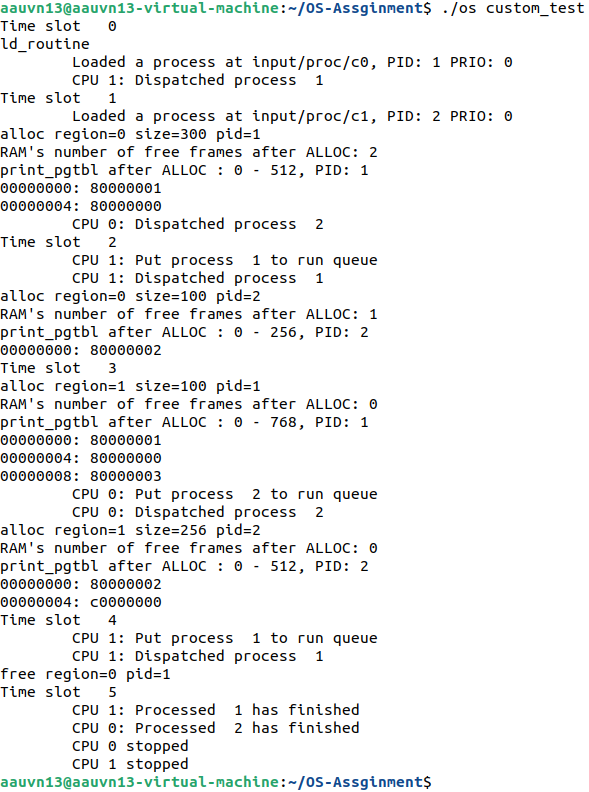
1. Instruction ‘ alloc 300 0 ’ from process c0 requires 300 bytes from RAM for register 0, meaning 2 frames (512 bytes) from RAM are needed for this allocation. RAM size = 3584 bytes.
2. Instruction ‘ alloc 100 0 ’ from process c1 requires 100 bytes from RAM for register 0, meaning 1 frame (256 bytes) from RAM is needed for this allocation. RAM size = 3328 bytes.
3. Instruction ‘ alloc 100 1 ’ from process c0 requires 100 bytes from RAM for register 1, meaning 1 frame (256 bytes) from RAM is needed for this allocation. RAM size = 3072 bytes.
4. Instruction ‘ alloc 256 1 ’ from process c1 requires 256 bytes from RAM for register 1, meaning 1 frame (256 bytes) from RAM is needed for this allocation. RAM size = 2816 bytes.
5. Instruction ‘ free 0’ from process c0 deallocates register 0. Since we cannot collect back the taken physical frames which might cause memory holes, we keep the collected storage space in a free list for further ‘alloc’ requests. RAM size = 2816 bytes.

*Example 2:*

We consider this input, where RAM is initialized with 1024 bytes (or 4 free frames), and SWP is initialized with 16 mb, with the same two processes c0 and c1 as the previous example.



Running the input, we obtain the following results:



Explanation for every memory allocation and deallocation call in execution order:

1. Instruction ‘ alloc 300 0 ’ from process c0 requires 300 bytes from RAM for register 0, meaning 2 frames (512 bytes) from RAM are needed for this allocation. RAM size = 512 bytes.
2. Instruction ‘ alloc 100 0 ’ from process c1 requires 100 bytes from RAM for register 0, meaning 1 frame (256 bytes) from RAM is needed for this allocation. RAM size = 256 bytes.
3. Instruction ‘ alloc 100 1 ’ from process c0 requires 100 bytes from RAM for register 1, meaning 1 frame (256 bytes) from RAM is needed for this allocation. RAM size = 0 bytes.
4. Instruction ‘ alloc 256 1 ’ from process c1 requires 256 bytes from RAM for register 1, meaning 1 frame (256 bytes) from RAM is needed for this allocation. Since RAM has no frames left, 1 frame from SWP is needed. RAM size = 0 bytes.
5. Instruction ‘ free 0’ from process c0 deallocates register 0. Since we cannot collect back the taken physical frames which might cause memory holes, we keep the collected storage space in a free list for further ‘alloc’ requests. RAM size = 0 bytes.

## 6. Synchronization problem:

In this section, we will look at many synchronization issues in our simple operating system. Because each process in our operating system has its own virtual memory and memory management unit, synchronization issues will not arise in this section. However, race conditions can occur in the physical memory part. Consider our memphy\_struct data type, which represents our physical device:

| struct memphy\_struct {  pthread\_mutex\_t memphy\_lock;  pthread\_mutex\_t memphy\_freefp\_lock;  pthread\_mutex\_t memphy\_usedfp\_lock;  pthread\_cond\_t wait\_cond;   /\* Basic field of data and size \*/  BYTE \*storage;  int maxsz;    /\* Sequential device fields \*/   int rdmflg;  int cursor;   /\* Management structure \*/  struct framephy\_struct \*free\_fp\_list;  struct framephy\_struct \*used\_fp\_list; }; |
| --- |

In memphy\_struct, there are three main elements that will be utilized by several processes at the same time: storage, free\_fp\_list, and used\_fp\_list. For each component, we will use a mutex lock to allow only one process to access these components at a time.

Race conditions are theoretically possible, although in our testing, it rarely happens. However, our OS has one significant synchronization issue. Consider the following example:

| custom\_test | c0 | c1 |
| --- | --- | --- |
| 2 2 2  256 16777216 0 0 0  0 c0 0  1 c1 0 | 1 6  calc  calc  calc  alloc 300 0  alloc 100 1  read 1 0 0 | 2 5  calc  calc  alloc 300 0  alloc 100 1  read 1 5 0 |

In this configuration, we have two CPUs and one frame of RAM. Process c0 will begin at timeslot 0 and process c1 will begin at timeslot 1. Both processes are willing to perform READ operations at time slot 5. As previously stated, the READ operation requires a frame in RAM before it can be performed. However, because both processes have a corresponding frame in the SWAP device, they must visit the list of used frames in RAM to obtain the victim frame in order to complete the place replacement procedure. In this case, if we used a simple mutex lock to control access to the used frame list, for example, process 1 can get the used frame but process 2 cannot, and thus cannot perform the READ operation, and our OS raised an ERROR because we assume there is always a used frame in RAM if there is no free frame in RAM.

Here we can easily observe that there is a segmentation ERROR here. Therefore we will added a condition variable in the MEMPHY\_get\_usefp() which make the process 2 (in our example above) to wait until process 1 finished it operation and then

# **III. Overall**