

# Project 2: HOST Dispatcher


Zhuohao Li ✉\*

May 22, 2022

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Lab Content</b>	<b>2</b>
2.1	Exercise11 . . . . .	2
2.1.1	API . . . . .	4
2.2	FCFS . . . . .	4
2.3	RR . . . . .	5
2.4	Feedback . . . . .	5
2.5	Resources Allocation . . . . .	6
2.5.1	Memory . . . . .	6
2.5.2	IO . . . . .	7
<b>3</b>	<b>Conclusion</b>	<b>7</b>
3.1	Compared with Real OS . . . . .	7
3.2	Achievements . . . . .	8

---

\*edith\_lzh@sjtu.edu.cn | 519021911248 |  [Project Repo](#)

# 1 Introduction

In this lab, we're gonna build a simple HOST Dispatcher, is a multi-process system. Processes in HOST are given four priorities. Level scheduler control, while being subject to limited hardware and software resources. For more details, please check more on [assignment.pdf](#) by TA.

**Zhuohao** finished [All](#).

The lab environment is **gcc (Ubuntu 7.5.0-3ubuntu1 18.04) 7.5.0 in Ubuntu 18.04**

## 2 Lab Content

We're gonna implement **FCFS, RR, 3-level Feedback, Resources Allocation**

### 2.1 Exercise11

As we only need to upload `exercise11` source code. So I just make comments on `./exercise11/src/hostd.c` as follows: (`./exercise11/src/mab.c` will be discussed in **Resource Allocation** later)

1. [Parse command line](#).

Firstly, we need to parse the arguments from the UNIX command line (generally, the command goes like `hostd [-mf -mn -mb -mw] <dispatch file>`, where `<dispatch file>` is list of process parameters as specified). `strcmp` could figure which memory allocation will be used. ( `-mx` is optional selection of memory allocation algorithm, `-mf` is First Fit (default), `-mn` is Next Fit, `-mb` is Best Fit, `-mw` is Worst Fit)

2. [Initialization](#)

Initialize `inputqueue`, `userjobqueue`, `dispatcherqueues` ([0] - real-time, [1]-[3] - feedback), `memory`, `resources`. Initialize `timer`

3. [Fill input queue from dispatch list file](#)

The process is submitted to the dispatcher `inputqueue` via `FILE *inputliststream` to read from an initial process list (`fcfs.txt` something like that) that designates the arrival time, priority, processor time required (in seconds), memory block size, and other resources requested.

4. [While there's anything in any of the queues or there is a currently running process:](#)

- [Legal check](#)

For example, illegal priorities, resource overflow for user process, illegal I/O request from real-time process. If there's an illegal stall happens, it returns an `ERROR` message.(For details, refer to source code)

- **Real time process**

Any pending Real-Time jobs are submitted for execution on a **FCFS** basis.

- **User process**

A process is "ready-to-run" when it has "arrived" and all required resources are available. After successfully passing the legal check, the process dequeues and is transferred to the appropriate priority queue within the **feedback** dispatcher unit, and the remaining resource indicators (memory list and I/O devices) updated.

If `currentprocess` is still remained, `remaincputime-=quantum`. Otherwise, if `current` is run out, `terminatePcb(currentprocess)` and `memFree(memoryblock)`, `rsrcFree(&resources, currentprocess->req)` to release the resources and terminate.

Or, if a user process and other processes are waiting in feedback queues, `suspendPcb(currentprocess)` and modify its priority by `++(currentprocess->priority)`. (Note that the priority value is up to 3). Then `enPcb` new process.

If no process currently running & queues are not empty, dequeue process from RR queue and set it as currently running process.

5. `sleep(quantum)`
6. `timer += quantum`
7. `return step4`

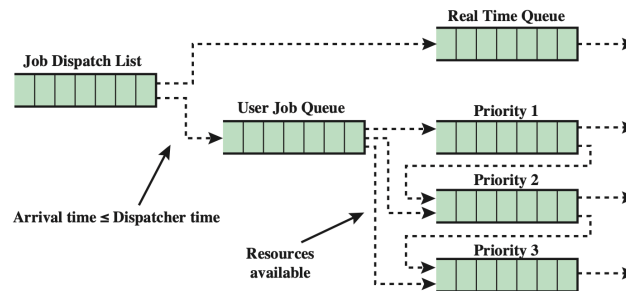


Figure 1: Dispatcher Logic Flow

### 2.1.1 API

- Maintain a `dispatcherqueues` data structure. It includes 4 index of use-ful queues.
- `dispatcherqueues[0]` refers to real-time-queue
- `dispatcherqueues[1, 2, 3]` 3 user-job-queues.
- `enPcb(dispatcherqueues[i], process)` refers to adding process to `dispatcherqueues[i]`
  - `suspendPcb(p)` refers to suspending p
  - `startPcb(p)` refers to start p
  - `deqPcb(PcbPtr* headofQ)` take Pcb from "head" of queue and returns PcbPtr if dequeued
  - `currentprocess` helps to refer as the current process.
  - `memFree(block)` refers to freeing block from memory
  - `rsrcFree(&rsrc, req)` refers to moving `&rsrc` out of `req` list of resources

### 2.2 FCFS

First-come-first-service. Just simply descent `currentprocess->remainingcputime` by `QUANTUM` until  $\leq 0$ , then free all resources and terminate Pcb. Next, fetch the head-of inputqueue if it is available.(Like line7-12 below)

---

```
1 //While there's anything in the queue or there is a currently
   //    running process:
2 //    i. If a process is currently running;
3 if (currentprocess) {
4 //        a. Decrement process remainingcputime;
5     currentprocess->remainingcputime -= QUANTUM;
6 //        b. If times up:
7     if (currentprocess->remainingcputime <= 0) {
8 //        A. Send SIGINT to the process to terminate it;
9         terminatePcb(currentprocess);
10 //        B. Free up process structure memory
11         free(currentprocess);
12         currentprocess = NULL;
13     }
14 }
15 //    ii. If no process now currently running, dispatcher
   //        queue is not empty, arrivaltime of process at head of
   //        queue is <= dispatcher timer:
```

```

16 if (!currentprocess && inputqueue && inputqueue->arrivaltime
    <= timer) {
17 //      a. Dequeue process and start it (fork & exec)
18 //      b. Set it as currently running process;
19     currentprocess = deqPcb( &inputqueue);
20     startPcb( currentprocess);
21 }
22 //      iii. sleep for one second;
23 sleep(QUANTUM);
24 //      iv. Increment dispatcher timer;
25 timer += QUANTUM;
26 //      v. Go back to 4.

```

---

## 2.3 RR

Maintain `rrqueue` for RR. If other processes are waiting in RR queue, `suspendPcb` it and enqueue it into `rrqueue`. (Like line4 and line 6 below)

```

1 else if (rrqueue)
2 {
3     //      A. Send SIGTSTP to suspend it;
4     suspendPcb(currentprocess);
5     //      B. Enqueue it back on RR queue;
6     rrqueue = enqPcb(rrqueue, currentprocess);
7     currentprocess = NULL;
8 }

```

---

## 2.4 Feedback

Feedback is the general situation for RR. Maintain a `fbqueue` for feedback. When a process ran out of its time period, `suspendPcb` it and `++priority`. (Like line4-9 below)

```

1 else if (CheckQueues(fbqueue) >= 0)
2 {
3     //      A. Send SIGTSTP to suspend it;
4     suspendPcb(currentprocess);
5     //      B. Reduce the priority of the process (if
        possible) and enqueue it on
6     //      the appropriate feedback queue;;
7     if (++(currentprocess->priority) >= N_FB_QUEUES)
8         currentprocess->priority = N_FB_QUEUES - 1;
9     fbqueue[currentprocess->priority] = enqPcb(fbqueue[
        currentprocess->priority],currentprocess);

```

```
10     currentprocess = NULL;
11 }
```

---

## 2.5 Resources Allocation

In this section, we're gonna implement **Resource Allocation** .

### 2.5.1 Memory

In `mab.c` and `mab.h`, pay attention to the API used. Here're some important APIs:

---

```
1 MabPtr memChk(MabPtr, int); // check whether it is available
2 int     memChkMax(int); // check if overflow the max
3 MabPtr memSplit(MabPtr, int); // divide a MabPtr memory block
      into 2, and the 1st has int size
4
5 MabPtr memAlloc(MabPtr arena, int size) // what we care about
```

---

- **First Fit**

Just scan `arena` to find the first one which is **available & larger than size**

- **Best Fit**

Find the smallest available memory block, to implement this, using `MabPtr m` to help restore the temporary current memory block and iterally check if the next legal one is **smaller**. This dispatching method will lead to a low-efficiency allocation.

- **Next Fit**

Find the next legal block from the last pointer points at. To implement this, using `MabPtr m` to help restore the temporary current memory block and start from there each time in the future.

- **Worst Fit**

Almost the same logic as **Best Fit**, just modify the part to find the next legal one which is **larger** and flush it.

In our design, best-fit or worst-fit may lead to less memory fragments, which is the better option.

### 2.5.2 IO

In `rsrc.c` and `rsrc.h`, no need to modify them. Just note that the APIs we're gonna use later

---

```
1 struct rsrc {
2     int printers; // printers
3     int scanners; //scanners
4     int modems; //modems
5     int cds; //storage(CD)
6 };
```

---

In `rsrc.c`, the logic is simple. For example, here's `rsrcAlloc()`

---

```
1 int rsrcAlloc(RsrcPtr available, Rsrc claim){
2     if (!rsrcChk(available, claim)) return FALSE;
3     available->printers -= claim.printers;
4     available->scanners -= claim.scanners;
5     available->modems -= claim.modems;
6     available->cds -= claim.cds;
7     return TRUE;
8 }
```

---

From line2-6 simply shows the idea of descending resources by request process asks for. The other functions are similar and easy to understand.

In `hostd.c`, we calls I/O resources by the process we defined before. For example, we can use `process->req.printers` to refer how many printers the process are asking for.

## 3 Conclusion

### 3.1 Compared with Real OS

Reference: [CRF](#), textbook edition10

- Linux uses `task_struct` to implement a process
- [time record](#)

In Linux, there's no concepts like **time period**, in comparison, the dispatcher will record every process. To implement this, Linux proposes **vruntime**, which could calculate how long it will take for this process based on overall operations. Higher **nice**(something like priority) will be distributed more time to execute.

- [process selecting](#)

Linux constructs a **Red-Black Tree** to sort list of process by **vruntime**. The least **vruntime** is preferred, this is just like greedy algorithm, so many optimizations are behind it.

- [suspend & waking](#)

Linux did these operations both on RB tree, processing interrupt and maintain a waiting queue.

- Some other schedulers like **Stop**(with highest priority), **Deadline**(using RB tree), **RT**(real-time scheduler).
- **HOST** is a simple process scheduler, so classic schedule algorithm(FCFS, RR, Feedback) is enough. Additionally, the time period is also used in this dispatcher.

The constraints in our design is mainly about:

- For Round Robin, urgent tasks are slow to respond. In UNIX, we could utilize **priority** + **time slice** to improve the efficiency. For time slice selection, if it's too small, frequent interruptions and process context switching will occur, increasing system overhead, but conducive to short jobs. If it's too large to degenerate into FCFS. The time slice should be slightly larger than the time of a typical interaction.
- For real-time process, which uses priority preemptive FCFS, it'll add more cost to the system.
- We could set authorities for IO to make sure it's safety and reasonable scheduled more.

## 3.2 Achievements

In this lab, I manage C coding and debug skills. Besides, I constructed a better understandings of process scheduling. By implementing FCFS, RR, Feedback. The common schedule method is settled and familiar with. I also learned deep in memory allocation strategies.

Thanks for TAs for discussing and resolutions!