Department of Computer Science and Engineering
The Chinese University of Hong Kong

CSCI/CENG 3150: Introduction to Operating Systems

Tutorial Two: Simulate FIFO Scheduling

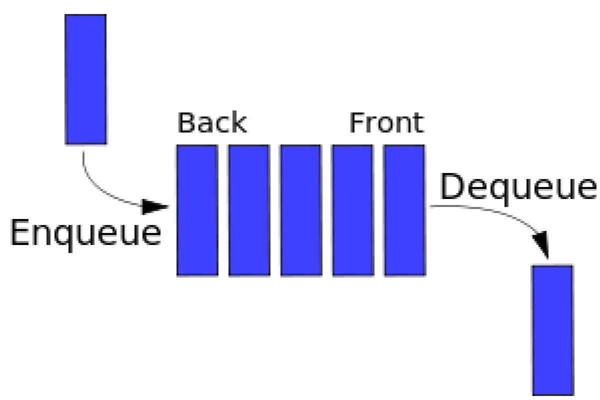
Objectives:
(1) Understand basic data structure: queue, and know well about the properties.
(2) Practice to simulate FIFO Scheduling with queue.

1. Queue

(1) **Concept:** A Queue is a linear collection in which the entities in the collection are kept in order and there are only two operations on the collection:

- **enqueue**: addition of entities to the rear terminal position,
- **dequeue**: removal of entities from the front terminal position).

This makes the queue a FIFO (First-In-First-Out) data structure, which means the first element added to the queue will be the first one to be removed.



(Representation of a FIFO queue)

There are several implementation methods of FIFO queue:

- Fixed length arrays: limited in capacity.
- Linked list: a regular singly linked list has efficient insertion and deletion at one end; if keep a pointer to the last node in addition to the first one, will enable it to implement an efficient queue.

(2) **Implementation**: Now, we will use the Linked List we provide to implement Queue. For the linked list, we have implemented these methods:

- `LinkedList* Create();`

  Create a new void Linked List. A void node will be created as the linked list head.

- `LinkedList* AddTail(LinkedList* Llist, Eletype value);`

  Add new node at tail.

- `LinkedList* AddHead(LinkedList* Llist, Eletype value);`

  Add new node at head.

- `LinkedList* Add(LinkedList* Llist, Eletype value, int position);`

  A method that can do add operation at all positions of linked list.

- `LinkedList* DeleteTail(LinkedList* Llist);`

  Delete the tail node.

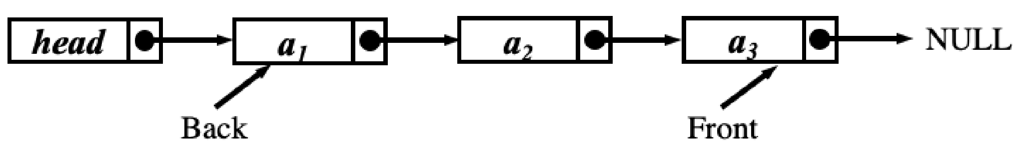
- `LinkedList* DeleteHead(LinkedList* Llist);`

  Delete the head node.

- `LinkedList* Delete(LinkedList* Llist, int position);`

  A method that can do delete operation at all positions of linked list.

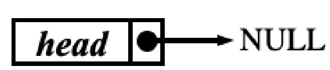
At first, we need to include Linked List head file *linkedlist.h*.



(The tail of Linked List is the Front end of Queue,
and the head of Linked List is the Back end of Queue.)

For Queue, we will also have several functions. The first one we want to implement is create a new queue, the return result just like the right figure:

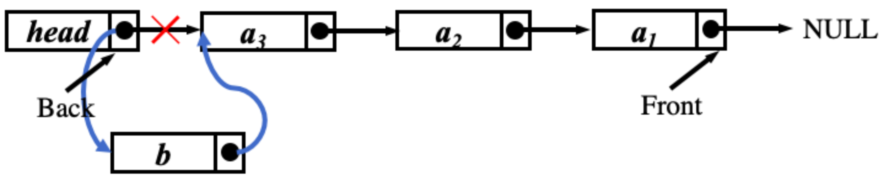
```
LinkedQueue* InitQueue(){
    return Create();
}
```



Then, it is also important to know whether a queue is empty:

```
int IsEmptyQueue(LinkedQueue* LQueue){
    return IsEmpty(LQueue);
}
```

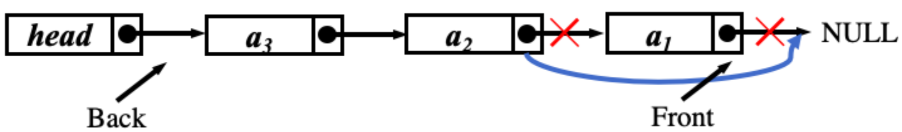
Next, we need to implement the most important functions, EnQueue, DeQueue and Front Queue:



(Enqueue new element b.)

```
LinkedQueue* EnQueue(LinkedQueue* LQueue, Eletype value){
    return AddHead(LQueue, value);
}
```

In this EnQueue function, we will return the queue.



(Dequeue operation will remove the element at the front end of queue.)

```
Eletype DeQueue(LinkedQueue* LQueue){
    if (IsEmpty(LQueue)){
        printf("Delete Error! Empty!");
        return -1;
    }else{
        LinkedQueue* pt = LQueue;
        Node tmp;
        while (pt->next){
            if (pt->next->next == NULL){
                tmp = pt->next;
                pt->next = NULL;
                break;
            }
            pt = pt->next;
        }
        return tmp->num;
    }
}
```

In the DeQueue function, we will return the value of removed element.

Then, we need a function to get the value of the element in the front end of queue:

```
Eletype FrontQueue(LinkedQueue* LQueue){
    if (IsEmpty(LQueue)){
        printf("Error! Empty!");
        return -1;
    }else{
        LinkedQueue* pt = LQueue;
        Node tmp;
        while (pt->next){
            if (pt->next->next == NULL){
                tmp = pt->next;
                break;
            }
            pt = pt->next;
        }
        return tmp->num;
    }
}
```

As for other functions like Length, please refer to the code file *queue.c* to check the implementation.

2. Simulate FIFO Scheduling

Download and copy tutorial-02.zip in your current directory and then unzip files as follows:

unzip ./tutorial-02.zip

In this zip, there will be 2 .h header files, 3 .c code file, process.file, queue.cfg and a Makefile. These files respectively are implemented linked list (linkedlist.h & linkedlist.c), implemented queue (queue.h & queue.c), **Simulation C file (csci3150_tut2.c, which will simulate the FIFO scheduling)**, process input file (process.file) and queue configuration file (queue.cfg).

As we know that Queue structure also have the property of FIFO. To simulate FIFO Scheduling, it is appropriate to use Queue (provided as **linkedlist.h** and **queue.h**).

Data Structure:

1. One process has several metrics, including pid, arrival time, service time, execution time, waiting time turnaround time and completion time. The structure of process is shown as follows:

```
typedef struct process {
    int process_id;
    int arrival_time;
    int service_time;
    int execution_time;
    int waiting_time;
    int turnaround_time;
    int completion_time;
} Process;
```

2. For the Queue we will use, the original element we stored should be changed to be Process. Just as the following code shows:

```
typedef struct node {
    Process proc;
    struct node *next;
} LinkedList, *Node;

typedef LinkedList LinkedQueue;
```

3. For more details of the implementation and functions of Queue we will use, please refer to the source codes we provided.

Input files: There are 2 input files **process.file** and **queue.cfg.**

1. **process.file** contains the process information and its format is as follows:

```
ProcessNum  N
pid:X1, arrival_time:X11, execution_time:X12
pid:X2, arrival_time:X21, execution_time:X22
…
pid:XN, arrival_time:XN1, execution_time:XN2
```

Here, the first line denotes there are N processes to be scheduled (there is at least one space character between ProcessNum and N), and from the second line to the (N+1)th line, each shows the pid (process id), arrival time and execution time of a process with the format like "pid:X1, arrival_time:X11, execution_time:X12" (separated by ","). An example is given below.

```
ProcessNum 10
pidnum:2, arrival_time:10, execution_time:10
pidnum:1, arrival_time:10, execution_time:10
pidnum:3, arrival_time:10, execution_time:10
pidnum:4, arrival_time:1, execution_time:11
pidnum:0, arrival_time:100, execution_time:10
pidnum:5, arrival_time:70, execution_time:30
pidnum:6, arrival_time:30, execution_time:46
pidnum:7, arrival_time:120, execution_time:15
pidnum:8, arrival_time:130, execution_time:19
pidnum:9, arrival_time:163, execution_time:20
```

2. **queue.cfg** contains the queue information and in this tutorial, we use only one queue, the example is given below:

```
QueueNum 1
Time_Slice_Q1 10
```

Here, the first line denotes the number of queues we will use in this scheduler, the next line denotes the time slice for Queue 1.

In cici3150_tut2.c file, there are mainly **3 parts** used to simulate the whole simulating procedure:

1. The first part is to deal with the input files.

   a) To deal with process.file, we have these Global variable and functions:

```
Process* proc_tmp;
int ReadProcessFile();
int min(int x, int y);
Process MinProc(Process x, Process y);
void SortedProcess(Process* p, int num);
```

   - `proc_tmp` is used to store processes gotten from process.file.
   - `int ReadProcessFile();` This function is used to parse process.file and return the number of process in this file.
   - `void SortedProcess(Process* p, int num);` This function is used to merge sort process according to their arrival time and pid. The auxiliary function are `int min(int x, int y)` and `Process MinProc(Process x, Processy )`, which return the smaller int and process with earlier arrival time and smaller pid respectively.

   b) To deal with queue.cfg, we have these functions:

```
int GetQueueNum();
void ReadQueueCfg(LinkedQueue** LQueue, int num);
```

   - `int GetQueueNum()` is used to get the number of queues in this file.
   - `Void ReadQueueCfg(LinkedQueue** LQueue, int num);` To call this function, we need a `LinkedQueue**` type variable and the number get in `GetQueueNum()`. After call this function, it will store queue information in parameter `LQueue`.

For above functions, if you are interested in the implementation, you can look up the source code for details.

2. The second part is to calculate the waiting time, service time, turnaround time and completion time for every process we get in the first part.

   - `void Calculate(Process* proc, int n);`

In this function, we will calculate waiting time, turnaround time and completion time for every process we get in first part. For our given process.file example, this function will output like this:

Process	Execution Time	Arrival Time	Waiting Time	TurnAround Time	Completion Time	Service time
4	11	1	0	11	12	1
1	10	10	2	12	22	12
2	10	10	12	22	32	22
3	10	10	22	32	42	32
6	46	30	12	58	88	42
5	30	70	18	48	118	88
0	10	100	18	28	128	118
7	15	120	8	23	143	128
8	19	130	13	32	162	143
9	20	163	0	20	183	163

Average waiting time: 10.500000
Average turnaround time: 28.600000

For an process, the service time means the time when the process begins to run. We have:

1) $service\ time_i = \begin{cases} servce\ time_{i-1} + execution\ time_{i-1} & , arrival\ time_i \le stt_{i-1} \\ arrival\ time_i, & arrival\ time_i > stt_{i-1} \end{cases}$

   where $stt_{i-1} = servce\ time_{i-1} + execution\ time_{i-1}$

2) $waiting\ time_i = service\ time_i - arrival\ time_i$

3) $turnaround\ time_i = execution\ time_i + waiting\ time_i$

4) $completion\ time_i = turnaround\ time_i + arrival\ time_i$

5) $average\ waiting\ time = \frac{\sum waiting\ time}{process\ num}$

6) $average\ turnaround\ time = \frac{\sum turnaround\ time}{process\ num}$

3. The third part is to use the queues and processes we get before to simulate FIFO scheduling. Actually, this is the **most important part**.

- `void Schedule (Process* proc, LinkedQueue* ProcessQueue, int proc_num);`

In this function, we can use a while loop to represent time. In every loop the process array proc[] will be travelled and if it is the time for some processes to arrive or to complete, we will do EnQueue and DeQueue operation respectively. If there exists DeQueue operation, set flag equals to 1.

```
int flag = 0;
for (int i=0; i<proc_num; i++){
    if (tmp_time == proc[i].arrival_time){
        if (IsEmptyQueue(ProcessQueue[0]))
            slice_offset = tmp_time % time_slice;
            ProcessQueue[0] = EnQueue(ProcessQueue[0], proc[i]);
    }
    if (tmp_time == proc[i].completion_time){
        de_proc = DeQueue(ProcessQueue[0]);
        flag = 1;
    }
}
```

For every past time slice, we will output a line. If some process finishes without using up one time slice, we will just output the information of this time slot.

```
if (flag == 0){
    if(tmp_time % time_slice == slice_offset && tmp_time != 0){
        if(Length(ProcessQueue[0])){
            Process front_proc = FrontQueue(ProcessQueue[0]);
            if (front_proc.arrival_time == tmp_time){}
            else {
                outprint(tmp_time-time_slice, tmp_time,
                 front_proc.process_id, front_proc.arrival_time,
                 front_proc.completion_time-tmp_time);
            }
        }
    }
}else if (flag == 1) {
    if(tmp_time % time_slice == slice_offset){
        outprint(tmp_time-time_slice, tmp_time,
              de_proc.process_id, de_proc.arrival_time,
              de_proc.completion_time-tmp_time);
    }else {
        slice_used = (tmp_time - slice_offset) % time_slice;
        slice_offset = (slice_used + slice_offset) % time_slice;
       outprint(tmp_time-slice_used, tmp_time,
               de_proc.process_id, de_proc.arrival_time,
               de_proc.completion_time-tmp_time);
    }
    flag = 0;
}
```

There are 3 situations we need to output:

1) Just use up the time of one time slice and no process finishes, output current process's information;
2) Finish one process and use up the time of one time slice at the same time, output finished process's information.
3) Finish one process without using up the time of one time slice, we need to calculate how much time are used during this time slice (slice_used) and the offset in one time slice (slice_offset). Then just output finished process's information, the time slot should be tmp_time-slice_used to tmp_time, where tmp_time is the time.

4. Main function.

```
int main(){
    int proc_num = ReadProcessFile();
    Process proc[proc_num];
    for (int i = 0;i < proc_num; i++){
        proc[i].process_id = proc_tmp[i].process_id;
        proc[i].arrival_time = proc_tmp[i].arrival_time;
        proc[i].execution_time = proc_tmp[i].execution_time;
    }
    SortProcess(proc, proc_num);

    int queue_num = GetQueueNum();
    LinkedQueue** ProcessQueue = (LinkedQueue**)malloc\
      (sizeof(LinkedQueue*) * queue_num);
    ReadQueueCfg(ProcessQueue, queue_num);

    Calculate(proc, proc_num);
    InitOutputFile();
    Schedule(proc, ProcessQueue, proc_num);

    return 0;
}
```

At first, call function `ReadProcessFile()` to parse process.file and store process information into Global Variable `Process* proc_tmp`. Then, we store process information into Local Variable process array `proc[]` by using a for loop. Next, use `SortProcess(proc, proc_num)` to sort processes according to their arrival time and pid.

Then, `queue_num` and `GetQueueNum()` are used to get queue number in queue.cfg. `ProcessQueue` will initialize first and store queue information by calling `ReadQueueCfg(ProcessQueue, queue_num)`.

At last, we call `Calculate(proc, proc_num)` and `Schedule(proc, ProcessQueue, proc_num)` to finish the whole simulation procedure.

Output: the program will output the schedule result to the file **output.log.**

The format in output.log is as follows:

Time_slot:x-y, pid:x1, arrival-time:x2, remaining_time:x3

Here, Time_slot: x-y denotes the time interval starting at time x and end at time y with the smallest time slice among all queues, pid:x1 denotes the corresponding process with pid x1 is scheduled in this time interval, arrival-time:x2 and remaining_time:x3 denote the arrival time and remaining time of the process are x2 and x3, respectively. There is at least one space character between them. An example is shown below:

```
Time_slot:1-11, pid:4, arrival-time:1, remaining_time:1
Time_slot:11-12, pid:4, arrival-time:1, remaining_time:0
Time_slot:12-22, pid:1, arrival-time:10, remaining_time:0
Time_slot:22-32, pid:2, arrival-time:10, remaining_time:0
Time_slot:32-42, pid:3, arrival-time:10, remaining_time:0
Time_slot:42-52, pid:6, arrival-time:30, remaining_time:36
Time_slot:52-62, pid:6, arrival-time:30, remaining_time:26
Time_slot:62-72, pid:6, arrival-time:30, remaining_time:16
Time_slot:72-82, pid:6, arrival-time:30, remaining_time:6
Time_slot:82-88, pid:6, arrival-time:30, remaining_time:0
Time_slot:88-98, pid:5, arrival-time:70, remaining_time:20
Time_slot:98-108, pid:5, arrival-time:70, remaining_time:10
Time_slot:108-118, pid:5, arrival-time:70, remaining_time:0
Time_slot:118-128, pid:0, arrival-time:100, remaining_time:0
Time_slot:128-138, pid:7, arrival-time:120, remaining_time:5
Time_slot:138-143, pid:7, arrival-time:120, remaining_time:0
Time_slot:143-153, pid:8, arrival-time:130, remaining_time:9
Time_slot:153-162, pid:8, arrival-time:130, remaining_time:0
Time_slot:163-173, pid:9, arrival-time:163, remaining_time:10
Time_slot:173-183, pid:9, arrival-time:163, remaining_time:0
```

We have two functions to do this work:

- `void InitOutputFile();`

In this function, we initialize the file output.log. If this file already exists, we will clear it; if it does not exists, make a new file named output.log.

- `void outprint(int time_x, int time_y, int pid, int arrival_time, int remaining_time);`

The parameters of this function are respectively x, y, x1, x2 and x3 we mentioned above. It will output a formatted line into output.log.

Compile and run after you download and unzip tutorial-02.zip from blackboard.

make

./FIFOScheduler

Then you can find output like this in terminal:

```
Process Execution Time  Arrival Time    Waiting Time    TurnAround Time Completion Time  Service time
4           11           1          0          11         12         1
1           10           10         2          12         22         12
2           10           10         12         22         32         22
3           10           10         22         32         42         32
6           46           30         12         58         88         42
5           30           70         18         48         118        88
0           10           100        18         28         128        118
7           15           120        8          23         143        128
8           19           130        13         32         162        143
9           20           163        0          20         183        163
Average waiting time: 10.500000
Average turnaround time: 28.600000
```

and in output.log, you should find:

```
Time_slot:1-11, pid:4, arrival-time:1, remaining_time:1
Time_slot:11-12, pid:4, arrival-time:1, remaining_time:0
Time_slot:12-22, pid:1, arrival-time:10, remaining_time:0
Time_slot:22-32, pid:2, arrival-time:10, remaining_time:0
Time_slot:32-42, pid:3, arrival-time:10, remaining_time:0
Time_slot:42-52, pid:6, arrival-time:30, remaining_time:36
Time_slot:52-62, pid:6, arrival-time:30, remaining_time:26
Time_slot:62-72, pid:6, arrival-time:30, remaining_time:16
Time_slot:72-82, pid:6, arrival-time:30, remaining_time:6
Time_slot:82-88, pid:6, arrival-time:30, remaining_time:0
Time_slot:88-98, pid:5, arrival-time:70, remaining_time:20
Time_slot:98-108, pid:5, arrival-time:70, remaining_time:10
Time_slot:108-118, pid:5, arrival-time:70, remaining_time:0
Time_slot:118-128, pid:0, arrival-time:100, remaining_time:0
Time_slot:128-138, pid:7, arrival-time:120, remaining_time:5
Time_slot:138-143, pid:7, arrival-time:120, remaining_time:0
Time_slot:143-153, pid:8, arrival-time:130, remaining_time:9
Time_slot:153-162, pid:8, arrival-time:130, remaining_time:0
Time_slot:163-173, pid:9, arrival-time:163, remaining_time:10
Time_slot:173-183, pid:9, arrival-time:163, remaining_time:0
```

Note:

In this tutorial, you do not need to submit any files. But please make sure you really understand FIFO scheduling and know how to deal with the input files and output file, because they will also be parts of Bonus Assignment 3.