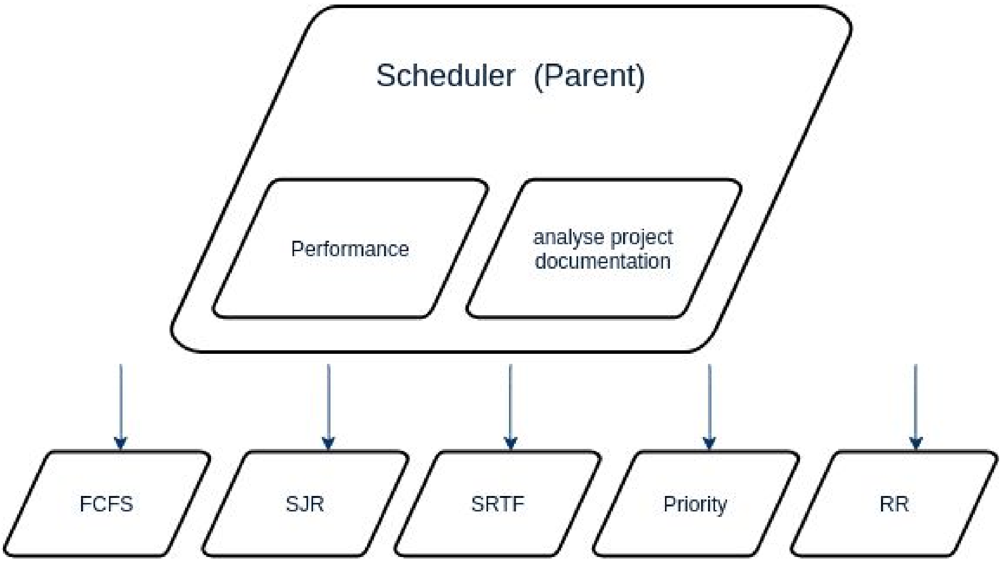
Operating System Project 2

1. 程式碼簡介

a. 簡述

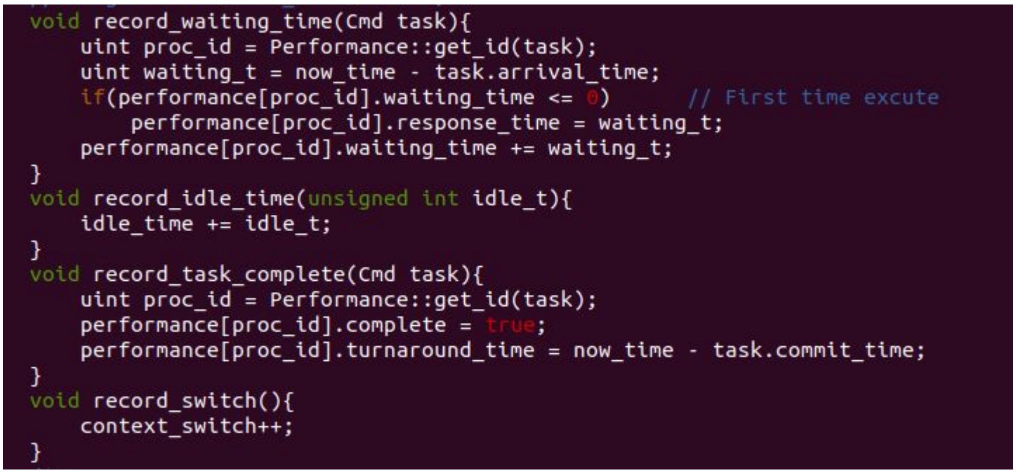
使用一個Parent Class，再繼承出其他的Child Classes。



b. Class

i. Performance：記錄各種performance的變數，例如turnaround_time、waiting_time、response_time、runtime等等。

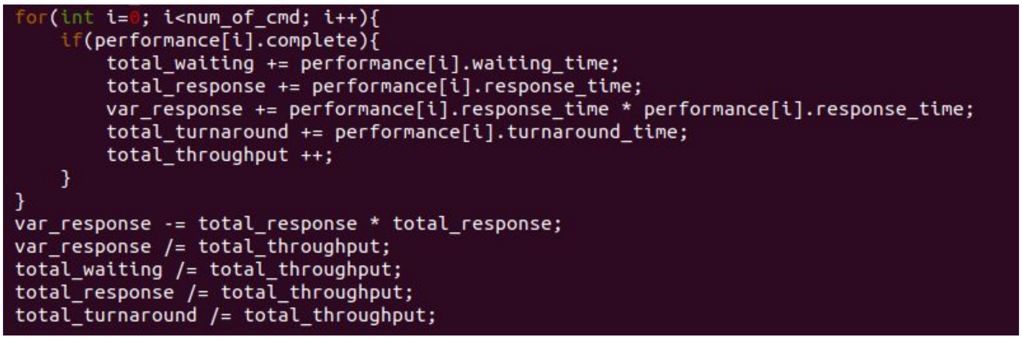
ii. Scheduler：所有scheduler的parent class，定義了cmd_queue、各種進行schedule會用到的公用變數，以及Performance陣列，為每個task紀錄，另外提供也紀錄的function，children class只要在適當時刻call這些function就能紀錄。



```
void record_waiting_time(Cmd task){
    uint proc_id = Performance::get_id(task);
    uint waiting_t = now_time - task.arrival_time;
    if(performance[proc_id].waiting_time <= 0)      // First time excute
        performance[proc_id].response_time = waiting_t;
    performance[proc_id].waiting_time += waiting_t;
}
void record_idle_time(unsigned int idle_t){
    idle_time += idle_t;
}
void record_task_complete(Cmd task){
    uint proc_id = Performance::get_id(task);
    performance[proc_id].complete = true;
    performance[proc_id].turnaround_time = now_time - task.commit_time;
}
void record_switch(){
    context_switch++;
}
```

紀錄Performance的Function

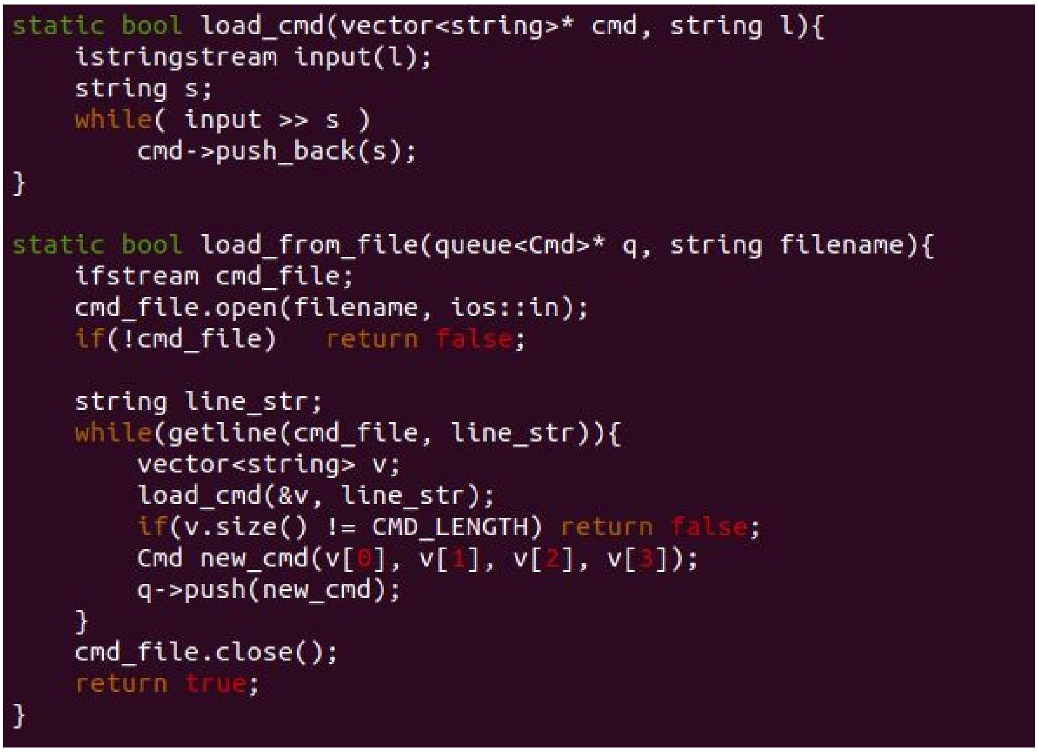
此外還會紀錄整體performance例如idle_time、context_switch的次數等等。也提供make_summary的function，方便class輸出summary並且輸出到檔案。Child Class需要實作virtual function work和finish，分別是用來執行schdule以及判斷是否已經schedule完成。



```
for(int i=0; i<num_of_cmd; i++){
    if(performance[i].complete){
        total_waiting += performance[i].waiting_time;
        total_response += performance[i].response_time;
        var_response += performance[i].response_time * performance[i].response_time;
        total_turnaround += performance[i].turnaround_time;
        total_throughput ++;
    }
}
var_response -= total_response * total_response;
var_response /= total_throughput;
total_waiting /= total_throughput;
total_response /= total_throughput;
total_turnaround /= total_throughput;
```

make_summary計算各種performance

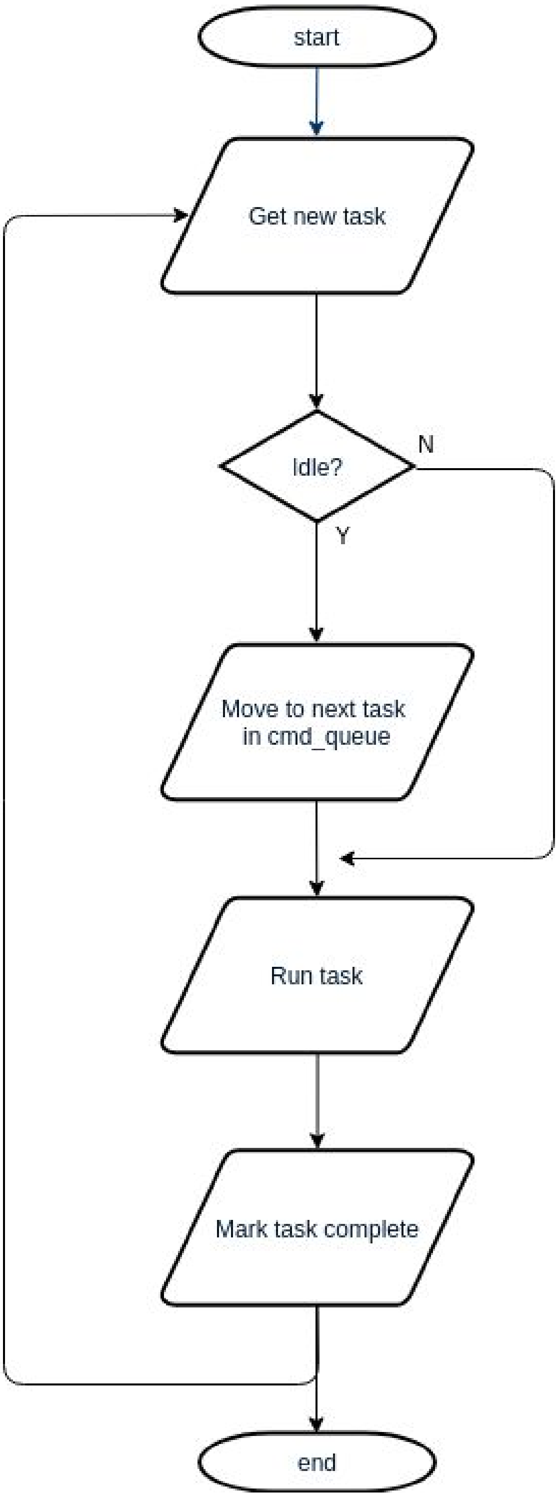
iii. Command：基本Task的資料結構，包含arrival_time(進入ready_queue的時間)、runtime(總執行時間)、commit_time(task submit的時間)、priroity以及基本的load from file的function。



```
static bool load_cmd(vector<string>* cmd, string l){
    istringstream input(l);
    string s;
    while( input >> s )
        cmd->push_back(s);
}

static bool load_from_file(queue<Cmd>* q, string filename){
    ifstream cmd_file;
    cmd_file.open(filename, ios::in);
    if(!cmd_file)   return false;

    string line_str;
    while(getline(cmd_file, line_str)){
        vector<string> v;
        load_cmd(&v, line_str);
        if(v.size() != CMD_LENGTH) return false;
        Cmd new_cmd(v[0], v[1], v[2], v[3]);
        q->push(new_cmd);
    }
    cmd_file.close();
    return true;
}
```

Command Class提供load from file的功能

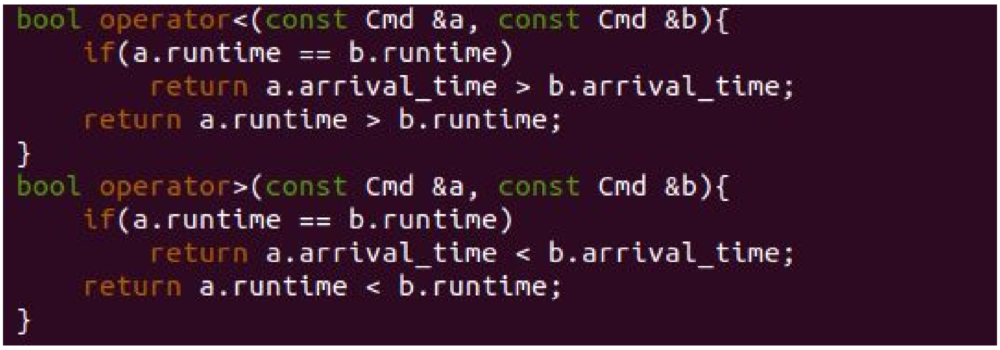
iv. Scheduler_fcfs：繼承自scheduler，實作單純的first in first out queue。



Work Flow

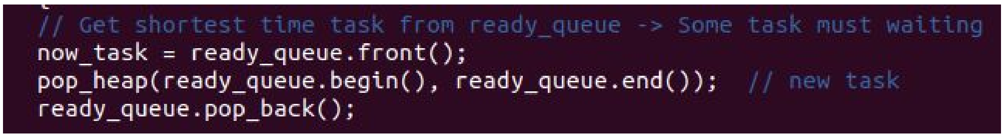
v. Scheduler_sjr

使用Algorithm 及Vector Library實作Heap堆積的ready_queue，以達到排序的效果，為了在Command間製造一個大小比較關係，必須overload >、<運算子先以runtime長度決定大小，若rutntime相同，才以arrival_time作為比較。



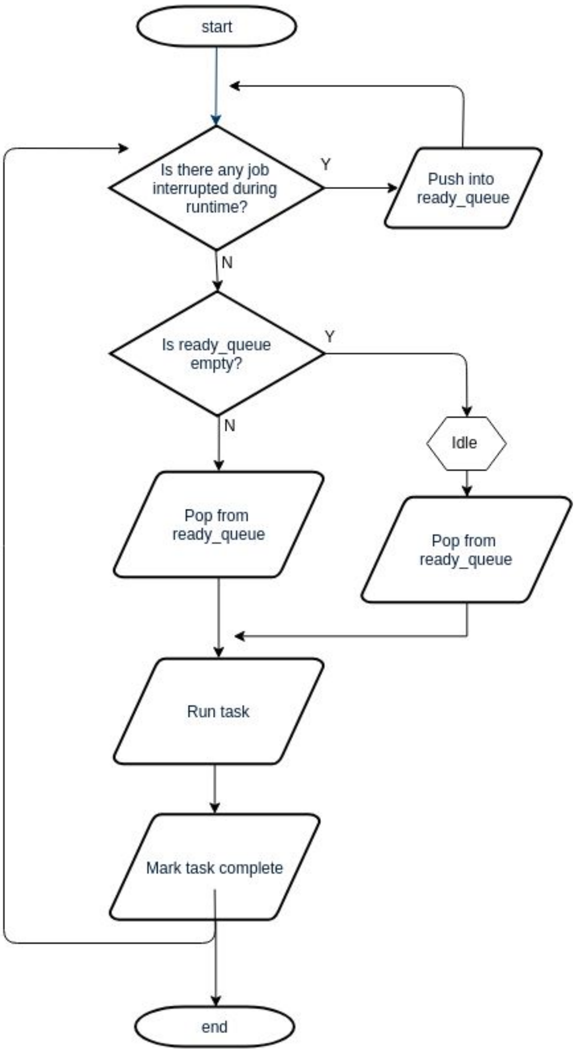
```
bool operator<(const Cmd &a, const Cmd &b){
    if(a.runtime == b.runtime)
        return a.arrival_time > b.arrival_time;
    return a.runtime > b.runtime;
}
bool operator>(const Cmd &a, const Cmd &b){
    if(a.runtime == b.runtime)
        return a.arrival_time < b.arrival_time;
    return a.runtime < b.runtime;
}
```

Overload Relationship Operator



```
// Get shortest time task from ready_queue -> Some task must waiting
now_task = ready_queue.front();
pop_heap(ready_queue.begin(), ready_queue.end());  // new task
ready_queue.pop_back();
```

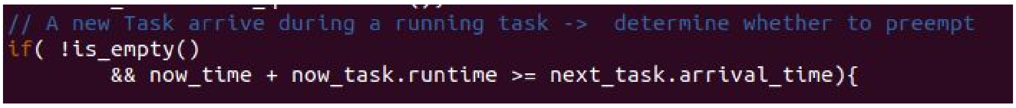
每次都從ready_queue選runtime最小的task



work flow

vi. Scheduler_srtf

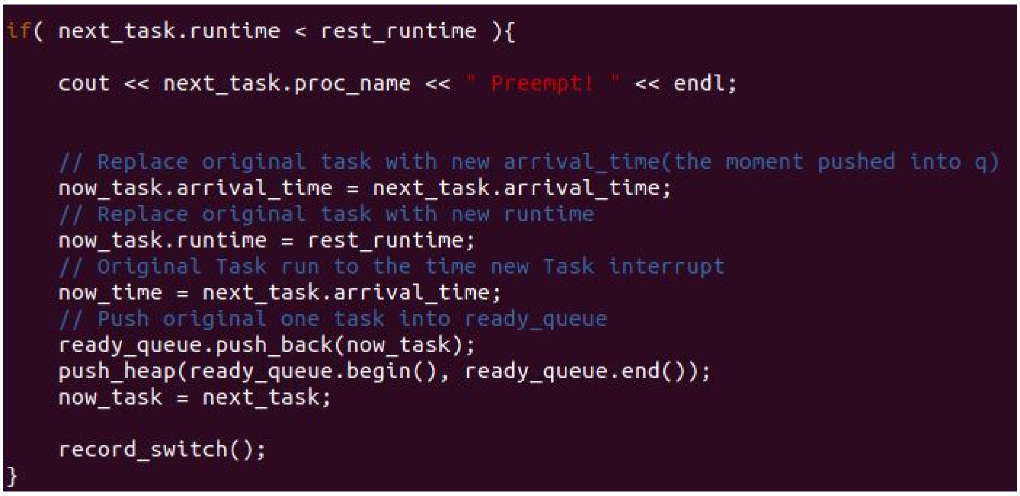
在run task之前檢查是不是會有新的task interrupt進來，並且決定是否要Preept，否則的話就push到ready_queue裏面。



```
// A new Task arrive during a running task ->  determine whether to preempt
if( !is_empty()
        && now_time + now_task.runtime >= next_task.arrival_time){
```

決定是否有new task

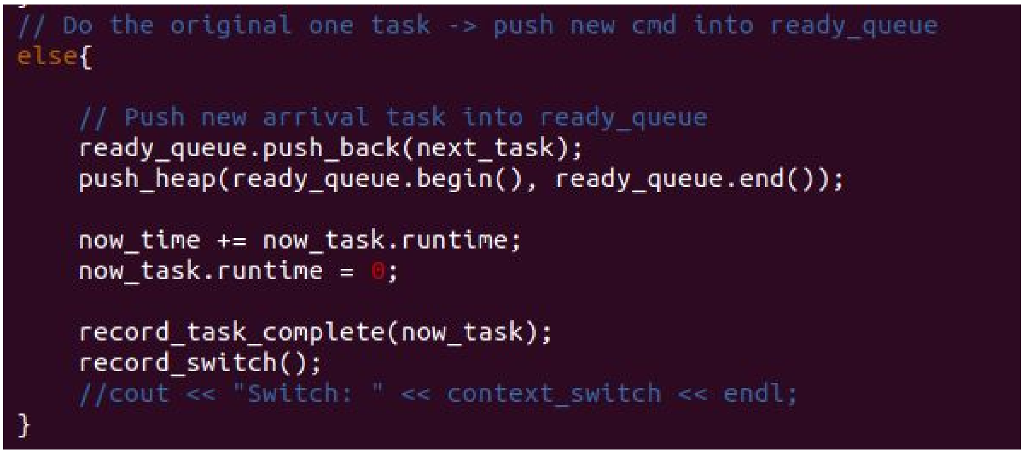
利用比較rest_runtime決定是否要preempt，若要preempt就context switch並把原本的task push 到ready_queue。



```
if( next_task.runtime < rest_runtime ){

    cout << next_task.proc_name << " Preempt! " << endl;


    // Replace original task with new arrival_time(the moment pushed into q)
    now_task.arrival_time = next_task.arrival_time;
    // Replace original task with new runtime
    now_task.runtime = rest_runtime;
    // Original Task run to the time new Task interrupt
    now_time = next_task.arrival_time;
    // Push original one task into ready_queue
    ready_queue.push_back(now_task);
    push_heap(ready_queue.begin(), ready_queue.end());
    now_task = next_task;

    record_switch();
}
```

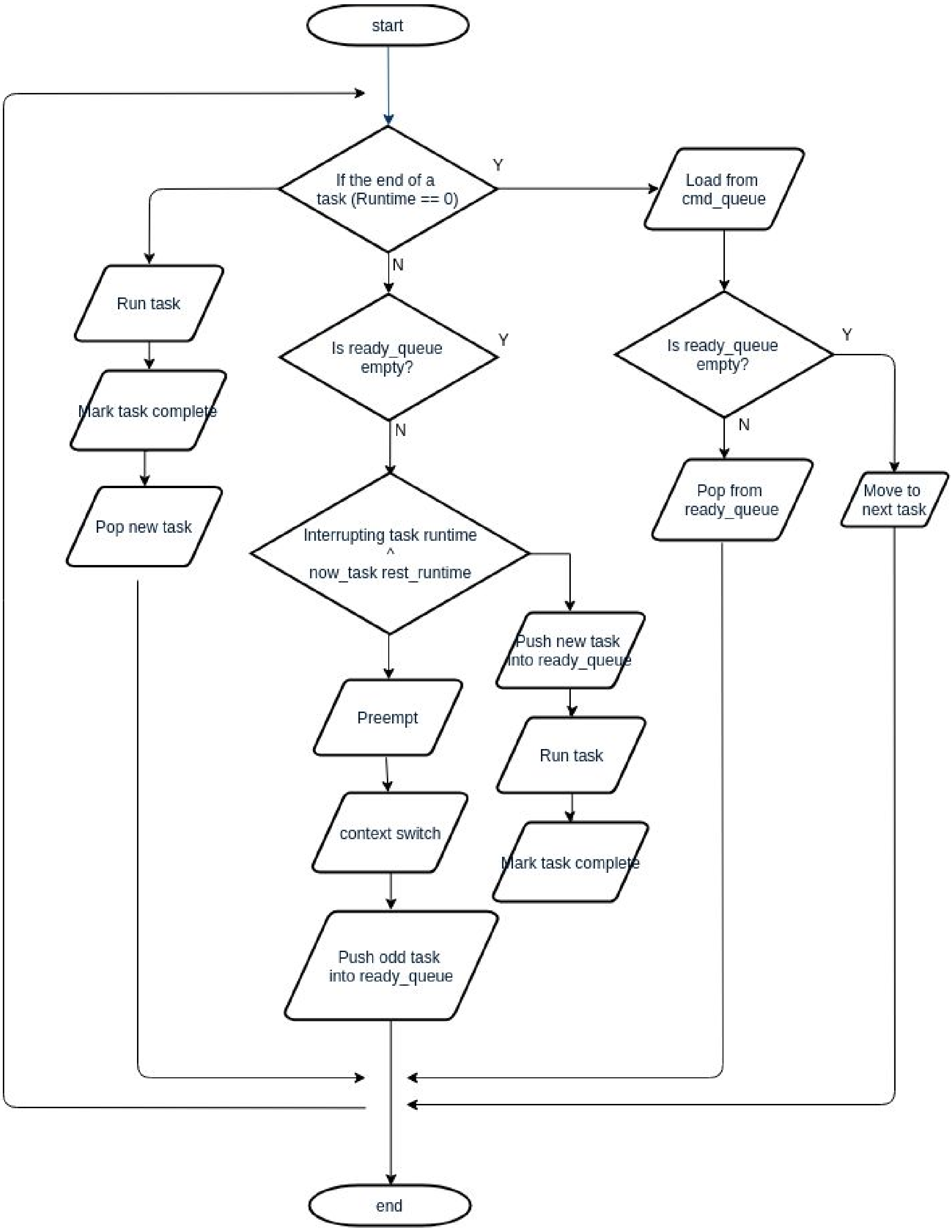
決定是否要preempt

如果不要preempt就把剩下的runtime做完。



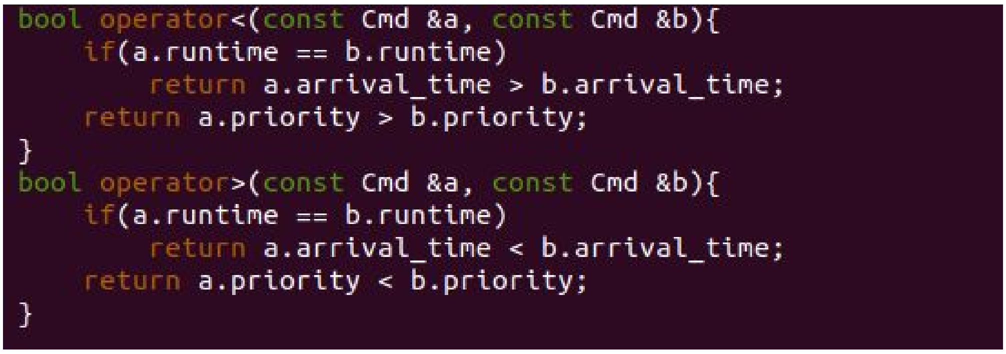
```
// Do the original one task -> push new cmd into ready_queue
else{

    // Push new arrival task into ready_queue
    ready_queue.push_back(next_task);
    push_heap(ready_queue.begin(), ready_queue.end());

    now_time += now_task.runtime;
    now_task.runtime = 0;

    record_task_complete(now_task);
    record_switch();
    //cout << "Switch: " << context_switch << endl;
}
```

把剩下的工作做完

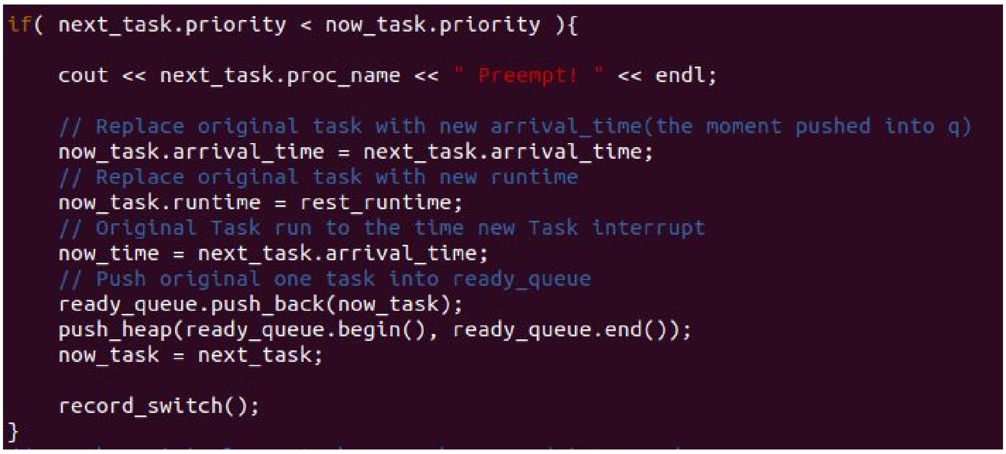

start
If the end of a task (Runtime == 0)
Y
N
Run task
Mark task complete
Pop new task
Is ready_queue empty?
Y
N
Interrupting task runtime ^ now_task rest_runtime
Preempt
context switch
Push odd task into ready_queue
Push new task into ready_queue
Run task
Mark task complete
Load from cmd_queue
Is ready_queue empty?
Y
N
Pop from ready_queue
Move to next task
end

vii. Scheduler_priority

Priority和srtf類似，只是把Command比較的運算子做修改，改為指定的比較方式



```
bool operator<(const Cmd &a, const Cmd &b){
    if(a.runtime == b.runtime)
        return a.arrival_time > b.arrival_time;
    return a.priority > b.priority;
}
bool operator>(const Cmd &a, const Cmd &b){
    if(a.runtime == b.runtime)
        return a.arrival_time < b.arrival_time;
    return a.priority < b.priority;
}
```



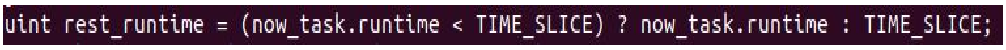
```
if( next_task.priority < now_task.priority ){

    cout << next_task.proc_name << " Preempt! " << endl;

    // Replace original task with new arrival_time(the moment pushed into q)
    now_task.arrival_time = next_task.arrival_time;
    // Replace original task with new runtime
    now_task.runtime = rest_runtime;
    // Original Task run to the time new Task interrupt
    now_time = next_task.arrival_time;
    // Push original one task into ready_queue
    ready_queue.push_back(now_task);
    push_heap(ready_queue.begin(), ready_queue.end());
    now_task = next_task;

    record_switch();
}
```

重新改寫比較運算子

註：其實應該要再寫一個Priorit的Class然後從這個class繼承出srtf才對，但後來沒有時間再改XD

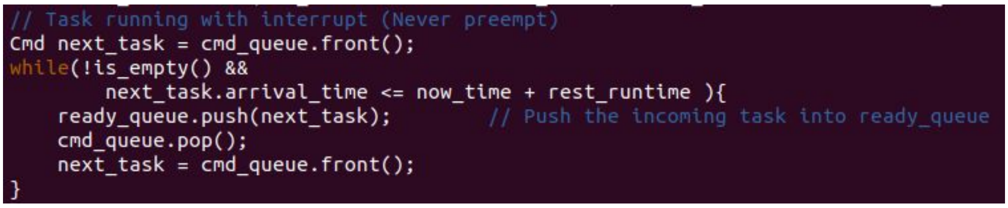
viii. Scheduler_rr

和FCFS極為相似，只是將runtime的上限改為time quantum，並且判斷是否會有task interrupt。



```
uint rest_runtime = (now_task.runtime < TIME_SLICE) ? now_task.runtime : TIME_SLICE;
```

計算rest_runtime



```
// Task running with interrupt (Never preempt)
Cmd next_task = cmd_queue.front();
while(!is_empty() &&
        next_task.arrival_time <= now_time + rest_runtime ){
    ready_queue.push(next_task);        // Push the incoming task into ready_queue
    cmd_queue.pop();
    next_task = cmd_queue.front();
}
```

檢查是否有new task interrupt

2. Performance

a. data_1

	time_used	throughput	idle	waiting	avg_response	variance_response	turnaround	context_switch

FCFS	110230	9999	27	5044	5044	83092	5055	10000
Priority	110230	9830	27	7747	4245	140782	5707	6458
RR	110228	9989	69	7642	2885	197585	7653	26083
SJR	99980	9425	27	333	333	310511	343	9426
SRTF	110204	9998	27	4490	31			

b. data_2

	time_used	throughput	idle	waiting	avg_response	variance_response	turnaround	context_switch
FCFS	2042	999	1	519	519	1378960	521	1000
Priority	2039	958	1	829	527	3344449	569	427
RR	2040	997	3	518	518	3684568	520	1003
SJR	991	665	1	40	40	5366538	42	665
SRTF	2042	999	1	466	318	2405582	320	1000

c. data_3

	time_used	throughput	idle	waiting	avg_response	variance_response	turnaround	context_switch
FCFS	4992	999	1	1985	1985	1210914	1990	1000
Priority	4992	998	1	2941	1953	1783898	1975	227
RR	4990	998	1	2374	1669	2588850	2379	1375
SJR	1003	346	1	71	71	10650871	74	346
SRTF	4992	999	1	2152	1442	935581	1447	1012