CMPS 405: OPERATING SYSTEMS

Simulations of CPU Scheduling

This material is based on the operating system books by Silberschatz and Stallings
And other Linux, system programming, and simulation books.

Objectives

- Motivations
- Invoking CPU Scheduler
- Preemptive and Non-Preemptive Scheduling
- Performance measures of interest
- Basic Scheduling Algorithms
- Other Scheduling Algorithms
- Linux scheduling related policies and system calls
- The actual Linux CPU scheduler
- Evaluation of CPU Scheduling Algorithms
- Simulation of CPU scheduling algorithms

Studying CPU Scheduling Algorithms

- Studying CPU scheduling algorithms involves calculating performance measures of interest which can be done in different ways:

  - Deterministic modeling:

  - Queueing models:

  - Simulations:

  - Implementation/Experimental:

<u>Ex.</u> *Compare and contrast the four different methods (discussed in the class) used to evaluate CPU scheduling algorithms.*

Next Topic

Simulation of

CPU scheduling algorithms

Simulation of CPU scheduling algorithms

- We are going to use Discrete-event Simulation (DES).
  - Events happen at discrete times.
  - An even list is created prior to the start of the simulation.
  - Mostly, the simulation ends when all events are served.

- Preliminary required knowledge
  - Knowledge of the simulated CPU scheduling algorithm
  - Construction of structures to model system entities.
  - Dealing with various types of data structures such as array, list, linkedlist, queue, priority queue and stack.
  - Knowledge needed to calculate performance measures of interest.
  - Knowledge of representing trace data.
  - Simulation and modeling basic concepts.
  - Moderate programming skill.

Simulation of CPU scheduling algorithms

- ❖ Development steps
  1. Identify and model main entities, their representation, and the required data structures to host them and operate on them.
  2. Supply the required data for these entities.
  3. Model the system and its resources
  4. Identify the events that might happen in the system.
  5. Model the simulation clock tick and its starting time.
  6. Decide on the termination condition(s) of the simulation
  7. Decide on the format of the trace data
  8. Decide on the calculations for the performance measures of interest.
  9. Model the scheduling policy and provide the corresponding implementation code.

Simulation of CPU scheduling algorithms

- ❖ Conducting a simulation study

1. You can use the simulation to simulate different settings of your system each is represented by a simulation scenario (an experiment).
2. Run each scenario many time (hundreds or thousands) and report the averages for all runs.
3. Study the reported results and construct appropriate plots to establish discussions on.
4. Collectively, study when varying a simulation parameter or a combination of parameters in multiple scenarios.
5. Provide your findings and comments on the overall simulation.

Simulation of a FIFO CPU scheduling

- Check the code on BB.
- Development steps
  1. Identify and model main entities, their representation, and the required data structures to host them and operate on them.
     - The main entity is a process that is represented by an id, arrival time, and CPU burst time.
     - A process is represented by a struct type in C

       ```
       struct process{
        int id;
        int arrival_time;
        int burst_time;
       };
       ```

     - Processes are hosted in an array of size N (equals to the number of required processes).

       ```
       struct process p[n];
       ```

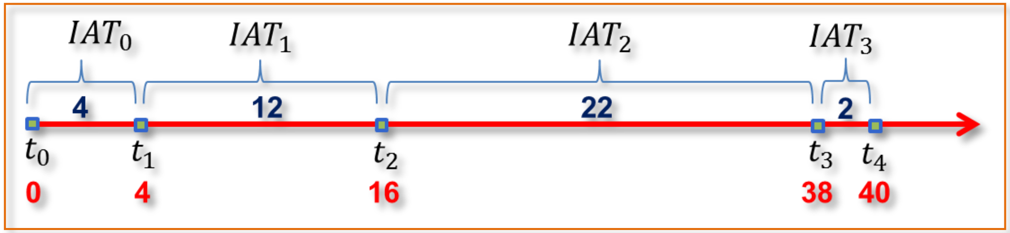
Simulation of a FIFO CPU scheduling

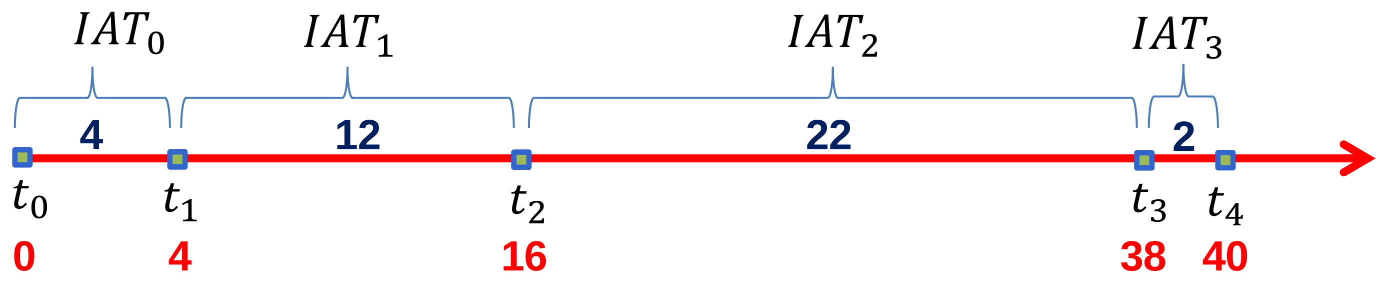
- Development steps

2. Supply the required data for these entities.

   - The data is auto-generated as follows
     - The id is a sequence number from 0 to N-1.
     - Time between arrival of jobs is a random number in the interval [0,10] ms.
     - Required CPU burst time is a random number in the interval [1,7] ms.
     - The corresponding code is

```
int previous_arrival_time=0;
int i=0;
for(i=0;i<n;i++){
        p[i].id=i;
        p[i].arrival_time=rand()%10+previous_arrival_time;
        previous_arrival_time=p[i].arrival_time;
        p[i].burst_time=1+rand()%7;
}
```



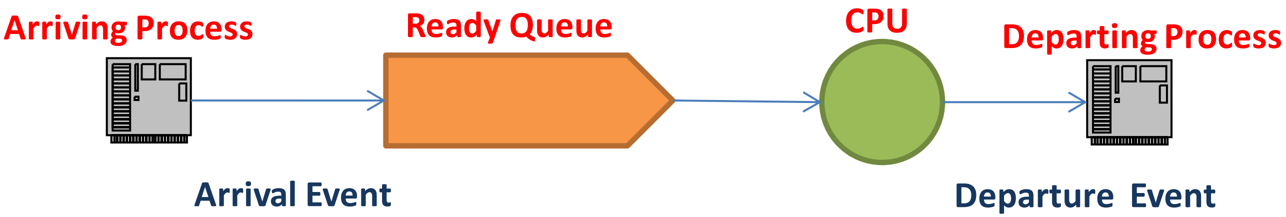

IAT_0
IAT_1
IAT_2
IAT_3
4
12
22
2
t_0
t_1
t_2
t_3
t_4
0
4
16
38
40

Simulation of a FIFO CPU scheduling

Ready Queue

- Development steps

3. Model the system resources
   - The system consists of one ready queue and one CPU.



   - For this specific CPU simulation (FIFO), We can use the same created list of the processes (the array process) to model the ready queue since the processes are fully served once they are selected by the algorithm.
   - The CPU is modeled as a Boolean variable with two states
     - True (1): indicating that it is idle
     - False (0): indicating that it is busy
   - The corresponding implementation of the CPU is

```
bool cpu_idle=true;
```

Simulation of a FIFO CPU scheduling

- Development steps

  4. Identify the events that might happen in the system.
     - We have two types of events
       - Arrival event: happens on arrival of a process to the ready queue.
       - Departure event: happens when an process finishes execution.
     - For this specific simulation (FIFO), these events happen only on time for each process and therefore, we do not need to represent them instead we only need to present the code to check when they are happening and to take the appropriate act for that event.
     - In most of simulation cases we need to define a struct for the event that has information about the type and time of the event, and encapsulates the process affected. All of these events are saved in an ordered list of events to be retrieved and executed by the simulation logic.

Simulation of a FIFO CPU scheduling

- Development steps

  5. Model the simulation clock tick and its starting time.
     - The simulation clock starting time is the time of the first arrival process to the system.
     - The tick of the simulation is a constant increment (constant one unit of time)
       - In most cases instead, the tick length is the time between events which varies. Why?
         - There isn't a strong point requiring recording the system state if it is not changing. Recall that the state of the system is changed whenever an event happens otherwise, it isn't changing.
     - Corresponding code

```
int start_time=p[0].arrival_time;
for(t=start_time;;t++)
```

Simulation of a FIFO CPU scheduling

- Development steps

6. Decide on the termination condition(s) of the simulation
    - The simulation finishes when all jobs are served.
    - Since jobs are served sequentially and their id is sequential, the simulation terminates if the id of the job to be served has reached N.
        - Do not forget that we are serving N jobs with ids 0 to N-1.
    - Corresponding code

```
if(id==n) break;
```

Simulation of a FIFO CPU scheduling

❖ Development steps

7. Decide on the format of the trace data

    - We are displaying on the screen the information related to the process under execution in the CPU with time. If the CPU is idle we display that with the time.
        - To do so a function to display a process information is coded and called when necessary and single simple printf statements will do when the CPU is idle.
    - The corresponding code is

```
void display_process(int t,struct process p){
        printf("%-8d P%d,%d\n", t, p.id,p.burst_time);
}


display_process(t,p[id]);
printf("%-8d idle\n",t);
```

    - In most cases, the trace data is recorded in a file with a predefined format . The recording is done only when there is a change in the state of the system.
    - The trace file can be used later on study the system behavior and calculate performance measures or even for visualization purposes.

Simulation of a FIFO CPU scheduling

- Development steps
  - Step 8 and 9 are specific to the simulated system and collected performance measures.
    - These are explained with the code of FIFO simulation.

8. Decide on the calculations for the performance measures of interest.
   - For the FIFO simulation example, the collected performance measures are average waiting time and CPU utilization. Therefore, we need to record the waiting times of the processes and CPU idle times.
   - You can one variable to accumulate waiting times for all processes and another to accumulate the CPU idle times. In this simulation instead, we define an array to save the waiting time for each process separately.
     - Waiting time for a process is calculated when it is picked up from the queue for service by the CPU.
     - CPU utilization and average queueing (waiting) time are calculated after executing all jobs.

Simulation of a FIFO CPU scheduling

- Corresponding code

```
int  waiting_time[n],  idle_time=0;



int sum=0;
for(i=0;i<n;i++){
        printf("%-8d %-8d\n",i,waiting_time[i]);
        sum+=waiting_time[i];
}
float avrwt = (float)sum/n;

float util = (float)(t-start_time-idle_time)*100/(t-start_time);
```

Simulation of a FIFO CPU scheduling

```
for(t=start_time;;t++){
        if(id==n) break;
        if(!cpu_idle){
                p[id].burst_time--;
                if(p[id].burst_time==0){
                        cpu_idle=true;
                        id++;
                }
                else
                        display_process(t,p[id]);
        }
        if(id<n && p[id].arrival_time<=t && cpu_idle){
                cpu_idle=false;
                waiting_time[id]=t-p[id].arrival_time;
                display_process(t,p[id]);
        }
        if(cpu_idle){
                idle_time++;
                printf("%-8d idle\n",t);
        }
 }
```

Are we done?

The CPU is busy

serve the process one unit of time then check if by that it will finish then depart it and mark the CPU to be idle and the id next process to be served is id+1 otherwise display the process in CPU.

If Idle and we have a process to serve

Change the CPU to busy, calculate the queueing time and display the process.

If after all of that the CPU is still idle

Accumulate the CPU idle time measure and display the CPU idle status.

Simulation of CPU scheduling

- The next exercise is to simulate RR, SJF, Priority, SRTF, and Preemptive Priority queueing.
  - In these simulations you need to:
    - Define the structure event as described earlier.
    - You need to use an ordered linkedlist of events entries called the event list.
      - This list is ordered based on time then event type where, departures have priority on arrivals if they are happening at the same time.
      - The tick of the simulation's clock is controlled by the time of the event happening.

- Rework the previous simulations with the features:
  - Calculating the rest of performance measures.
  - Rearranging the trace in tabular format and writing it to a file.

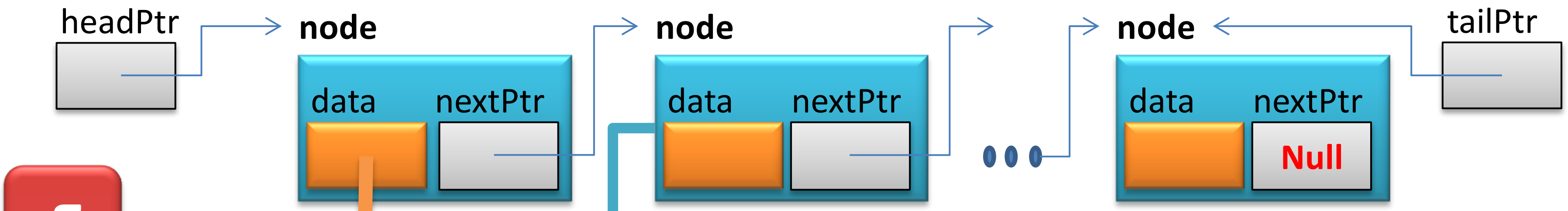
- Try more advanced simulations:
  - Linux scheduler with one CPU and two queues (active and expired)

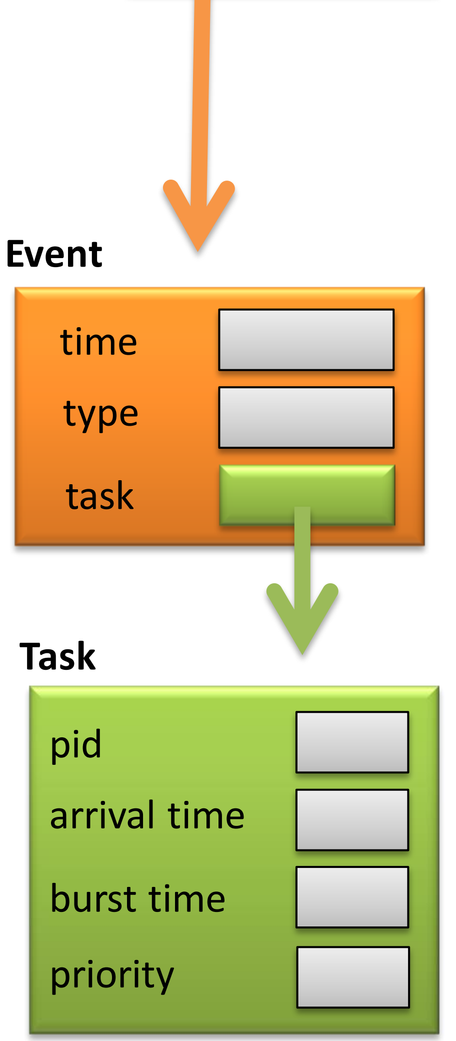
- Check the code on BB.

Proper DES Simulation for FIFO Scheduling

- Check the code on BB.
- In DES the simulation executes events from an event list.
  - The event list is a linked list
    - The data part of a node in this list is an event.
  - The nodes in the event list are ordered by time then event type.
    - An event type can be either arrival or departure.
    - Nodes are ordered first by event time and then by event type.
      - If the time of event A is less than the time of event B then Event A should be ordered before event B.
      - If the time of event A equals the time of event B and they are of different types (arrival departure) then the departure event should be ordered before the arrival event.
    - An event structure has type, time, and a task structure.
    - A task structure has information about the task/process such as pid, arrival time, CPU burst time, priority, …etc.

Event List Design and Implementation



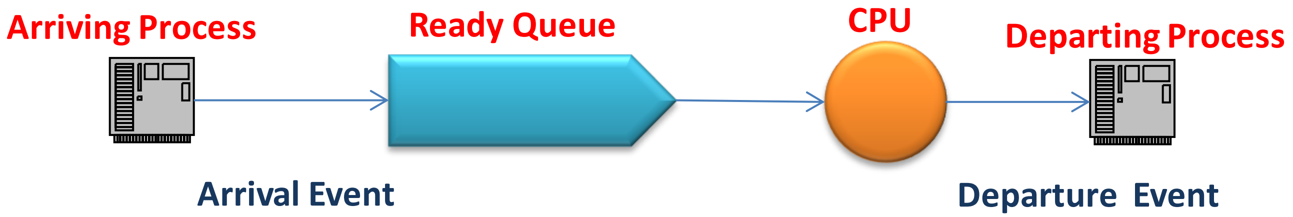


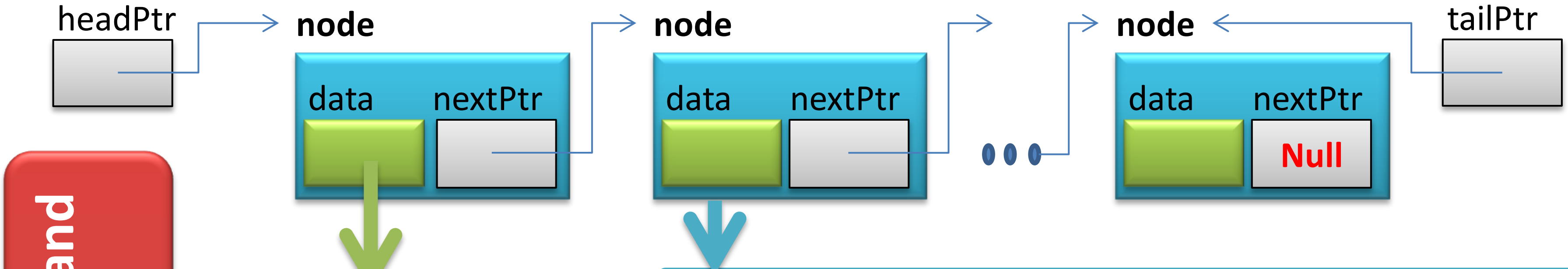
```
struct eventQnode{//an node in the events list
                Event data;//the event
                struct eventQnode *nextPtr;
};
typedef struct eventQnode EventQnode;
typedef EventQnode *EventQnodePtr;
```

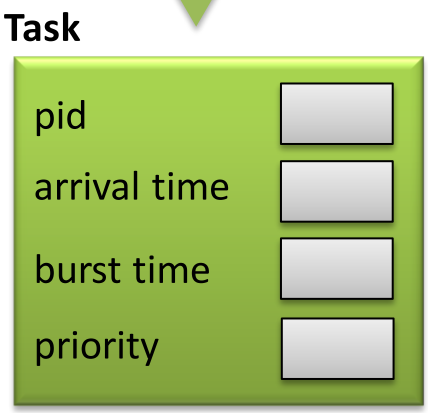
```
struct event{//an event event
                int type;//event type 0:arrival, 1: departure
                int time;//event time
                Task task;//the process
};
typedef struct event Event;
typedef Event *EventPtr;
```

```
struct task{//process
                int id;//pid
                int bt;//burst time
                int at;//arrival time
                int pr;//priority
};
typedef struct task Task;
typedef Task *TaskPtr;
```

Run Queue Design and Implementation







```
struct qnode{//a node in the run/ready queue
                Task data;//process
                struct qnode *nextPtr;
};
typedef struct qnode Qnode;
typedef Qnode *QnodePtr;
```

```
QnodePtr rqheadPtr=NULL, rqtailPtr=NULL;//the run queue
EventQnodePtr eventsQheadPtr=NULL, eventsQtailPtr=NULL;//the event queue/list
```

Arrival Events

- All arrival events are generated prior to the executing the simulation scheduler.
  - Generate the data for a task
    - Set the number of tasks, and other information such as Max IAT (inter-arrival time), Max burst time, the rage of priorities, …etc.
    - Use rand with bounded intervals to generate the data.
  - Each time you create a task
    - Encapsulate the task in an event
      - The type of the event should be set to arrival
      - The time of the event should be set to the arrival time generated for the task
    - Insert the task in the events list
  - Once finished, you will have a time ordered list of all arrival events.

Arrival Events

```
EventQnodePtr eventsQheadPtr=NULL, eventsQtailPtr=NULL;//the event queue/list
Task task;//the process structure
Event event;//the event structure
int prevat = 0, i;//set the previous arrival time to zero

for(i=0;i<MAXTASKS;i++){//generate all arrivals and insert them in the event list
        //fill up the info of the process structure
        task.id=i;
        task.at=rand()%IAT+prevat;
        prevat=task.at;
        task.bt=rand()%MAXBURSTTIME+1;
        //fill up the info of the event structure
        event.type=0;//event type is 0:arrival
        event.time=task.at;//event time
        event.task=task;//note that the process is encapsulated in an event structure
        enqueueevent(&eventsQheadPtr,&eventsQtailPtr,event);//insert in the events list
}
```

The Scheduler

```
while(!isEmptyEQ(eventsQheadPtr)){
        currentEvent = dequeueevent(&eventsQheadPtr,&eventsQtailPtr);//get an event
        clock=currentEvent.time;//
```

Departure

```
        if(currentEvent.type==1){//Departure logic
                idle=1;
                task = currentEvent.task;
                printf("\nTime: %d : Departure : task(id:%d,bt:%d) has finished.\n", clock,
                        task.id,task.bt);
        }
```

Arrival

```
        if(currentEvent.type==0){//Arrival logic
                task = currentEvent.task;
                printf("\nTime: %d : Arrival : task(id:%d,bt:%d).\n", clock, task.id,task.bt);
                enqueue(&rqheadPtr,&rqtailPtr,task);
        }
```

Service

```
        if(idle==1 && !isEmpty(rqheadPtr)){//Service logic
                idle = 0;
                 currentEvent.task=dequeue(&rqheadPtr,&rqtailPtr);
                 wt+=clock-currentEvent.task.at;
                 currentEvent.type=1;
                 currentEvent.time=clock+currentEvent.task.bt;
                 enqueueevent(&eventsQheadPtr,&eventsQtailPtr,currentEvent);
                 task = currentEvent.task;
                 printf("\nTime: %d : Serving : task(id:%d,bt:%d).\n", clock,task.id,task.bt);
        }
}
```

Simulation of CPU scheduling

- The next exercise is to simulate RR, SJF, Priority, SRTF, and Preemptive Priority queueing.

- Rework the previous simulations with the features:
  - Calculating the rest of performance measures.
  - Rearranging the trace in tabular format and writing it to a file.

- Try more advanced simulations:
  - Linux scheduler with one CPU and two queues (active and expired)
  - Linux scheduler with N CPUs and two queues (active and expired) for each CPU.

- Check the code on BB.

Deriving Priority sim from FIFO sim

1. Generate data for the priority in the task structure.

2. Change is the enqueue function of the run queue to a priority enqueue based on task priority.
   - This can be achieved by only changing the code used to find the appropriate insertion position, i.e., the code of the while loop, with the following code:

```
while(current!=NULL && task.pr>=(current->data).pr){
```

- Having that, you do not need to change the logic of the simulator.

Deriving SJF sim from FIFO sim

1. Change is the enqueue function of the run queue to a priority enqueue based on task burst time.
   - This can be achieved by only changing the code used to find the approperiate insertion position, i.e., the code of the while loop, with the following code:

```
while(current!=NULL && task.bt>=(current->data).bt){
    prev = current;
    current = current->nextPtr;
}
```

- Having that, you do not need to change the logic of the simulator.

Deriving RR sim from FIFO sim

- We reuse the code of FIFO simulation to construct RR simulation in this manner:

  1. Before the main, define a constant to represent Quantum time.

     ```
     const int QT = 20;
     ```

  2. Change the departure logic since in RR a departure can also represent enqueue of the task if it was not finished.

```
if(currentEvent.type==1){//Departure logic
    idle=1;
    task = currentEvent.task;
    if(task.bt>0){
        task.at=clock;
        enqueue(&rqheadPtr,&rqtailPtr,task);
        printf("\nTime: %d : Departure : task(id:%d,bt:%d) back
                to run queue.\n", clock, task.id,task.bt);
    }
    else
        printf("\nTime: %d : Departure : task(id:%d,bt:%d) has
                finished.\n", clock, task.id,task.bt);
}
```

Deriving RR sim from FIFO sim

3. Change the service logic such that we recalculate the remaining burst time for the task after service.

```
if(idle==1 && !isEmpty(rqheadPtr)){//Service logic
        idle = 0;
        currentEvent.task=dequeue(&rqheadPtr,&rqtailPtr);
        task = currentEvent.task;
        printf("\nTime: %d : Serving : task(id:%d,bt:%d).\n",
                  clock,task.id,task.bt);
        wt+=clock-currentEvent.task.at;
        currentEvent.type=1;
        int st;
        if(currentEvent.task.bt<QT)
            st=currentEvent.task.bt;
        else
            st=QT;
        currentEvent.time=clock+st;
        currentEvent.task.bt-=st;
        enqueueevent(&eventsQheadPtr,&eventsQtailPtr,currentEvent);
}
```

Can you?

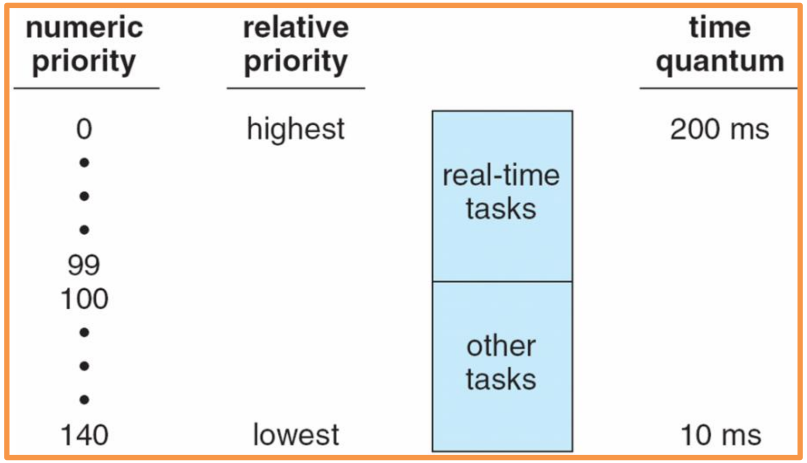
Code the simulation for Preemptive-Priority reusing the code of Priority simulation and code the simulation for SRTF reusing the code of SJF simulation.

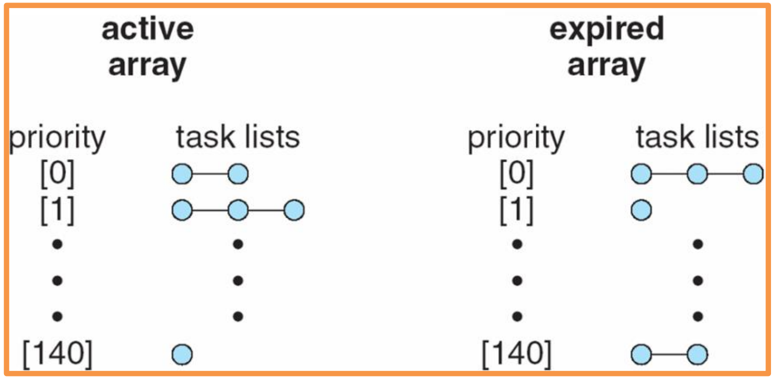
Hint

- Always save the event of the task currently executing in CPU, say cpuevent.
  - Change the logic of arrival to include the logic of preemption when applicable, which involves:
    - Checking for possible preemption,
    - If there is a preemption of the task of cpuevent, then calculate the left over burst time, and enqueu the task of the cpuevent in the run queue with arrival time equals to current clock time.
  - Change the logic of service such that it ends up with setting the cpuevent to currentevent;
  - No change on the logic of departure.

Linux Scheduler

- An overview of Linux scheduler
  - There are two priority ranges: Real-time range from 0 to 99 and nice value from 100 to 140.
  - Linux assigns higher-priority tasks longer time quanta and lower-priority tasks shorter time quanta.
  - Every processor has its own runqueue which contains two priority arrays: active and expired arrays.
  - Every processor services tasks from its own active priority array.
  - When all tasks have exhausted their time slices (that is, the active array is empty), the two priority arrays are exchanged; the expired array becomes the active array, and vice versa.





Linux Scheduler

- Linux implements real-time scheduling as defined by POSIX.1b.
  - A deeper overview of Linux scheduler priorities
    - Real-time tasks are assigned static priorities. All other tasks have dynamic priorities that are based on their nice values plus or minus the value 5.
    - A task's dynamic priority is recalculated when the task has exhausted its time quantum and is to be moved to the expired array.
    - The interactivity of a task determines whether the value 5 will be added to or subtracted from the nice value.

A task's interactivity is determined by how long it has been sleeping while waiting for I/O.

→ Tasks that are more interactive typically have longer sleep times and therefore are more likely to have adjustments closer to −5, as the scheduler favors interactive tasks. The result of such adjustments will be higher priorities for these tasks.

→ Conversely, tasks with shorter sleep times are often more CPU-bound and thus will have their priorities lowered.