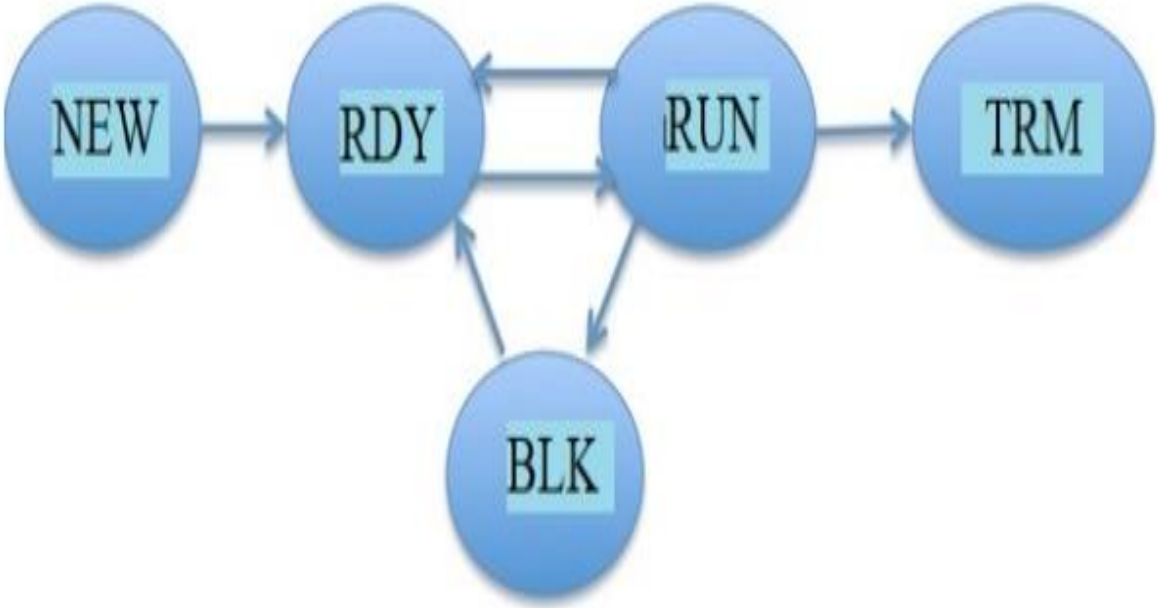
Data Structures and Algorithms



Process Scheduler

Project Requirements

Objectives:

By the end of this project, you should be able to:

- Understand unstructured, natural language problem description and derive an appropriate design.

- Intuitively modularize a design into independent components.

- Build and use data structures to implement the proposed design.

- Write a complete object-oriented G++ program that performs a non-trivial task.

Introduction:

Process scheduling is allocating resources to different processes so that they can be completed in an efficient and timely manner. The main objectives of scheduling are optimizing the use of resources, keep the CPU busy as possible, have maximum utilization of the CPU, and minimize response time and waiting time for processes being scheduled. One more objective in multiprocessor systems is to ensure load balancing among different processors.

In this project you are required to develop a program that simulates the operation of the process scheduler in ‘a multiprocessor system and calculates some related statistics in order to help improve the overall scheduling system.

Basic Definitions

Processes:

"The following information should be maintained for each process in the system."

- **PID**: a unique identifier for each process.

- **Arrival Time (AT)**: the time when the process is ready to be scheduled.

- **Response Time (RT)**: The difference between the arrival time and the time at which the

process gets the CPU for the FIRST time.

- **CPU time (CT)**: total time needed to run the process continually.

- **Termination Time (TT)**: Time when the process finishes execution.

- **Turnaround duration (TRT)**: The total time a process spends in the system from its arrival to

its termination.

$$TRT = TT - AT$$

- **Waiting Time (WT):** Total time a process spends in the system not being executed by the CPU.

$$WT = TRT - CT$$

- **Process States**: **[NEW, RDY, RUN, BLK, TRM, ORPH]**

[NEW] New: A process that has been created but not arrived yet

[RDY] Ready: A process that has arrived and was moved to the ready queue (by the scheduler). it is ready to be served by the CPU.

[RUN] Running: A process being served by the CPU.

[BLK] Blocked: If a process requires /O resources during execution, it changes from run to blocked state. The process should wait till the required I/0 resources become available. It then completes its I/ operations then advances back to the ready state.

[TRM] Terminated: A process that has finished execution.

[ORPH] Orphan: more about orphan state in "Process Scheduling Criteria" below

- **I/O Request Time (IO_R)**: The execution interval after which the process requests for **I/O** resources.

For example, if a process request **I/O** after 5 timesteps, this means that it requests **I/O** after being executed by CPU for 5 timesteps. These 5 timesteps of execution may be continuous or not depending on the scheduling algorithm of the processor executing it.

- **I/O Duration (IO_D)**: Time duration required for a process to perform some **I/O** task. Notice that a process may require other resources. But for simplicity in our project, we assume that the process may require **I/O** resources only.

Processors:

"Processors are responsible for process execution. The system in our project is a multiprocessor system.

We will adopt the following rules in our project:

1. Each processor has its own ready queue.
2. Each processor has its own scheduling algorithm to decide which process to excuse next.

In this project the words CPU and processor are used exchange ably to express the same thing.

A processor may be in one of 2 states:

1. BUSY: executing a process
2. IDLE: Has no process to execute (and it read queue is empty - No process to pick)

Processor Load (%)

In multiprocessor systems, one main objective of the scheduler is to balance load among different processors. To measure Load balance, we calculate the load on each processor as follows:

$$Pload = \frac{Total\ Processor\ Busy\ Time}{Total\ turnaround\ Time\ for\ Allprocesses}$$

Processor utilization (%)

Processor utilization is the percentage of time the processor is executing processes. Ideally, it should be 100% of the time. However, GPU may be idle for some time:

$$PUtil = \frac{Total\ Processor\ Busy\ Time}{Total\ Processor\ Busy\ Time + Total\ processor\ IDLE\ time}$$

Scheduler:

'The scheduler's job is to decide of which process to execute next depending on the scheduling algorithm (see below)

Scheduling algorithms:

'There are many scheduling algorithms. In this project we are concerned with the following algorithms only:

- First Come First Serve:

FCFS is considered to be the simplest of all operating system scheduling algorithms. First come first serve scheduling algorithm states that the process that requests the CPU first Is allocated the CPU first

- Shortest Job First(SJF):

'Shortest job first (SJF) is a scheduling algorithm that selects the waiting process with the smallest remaining GPU time (CT) to execute next.

- Round Robin (RR):

Round Robin is the preemptive version of FCFS algorithm where a running process (in RUN state) is assigned a fixed **Time Slice** to execute part of the process. Then stop the process at hand (put it in RDY state) and get another RDY process and RUN it giving it a fixed time slice too then move to a third process and 5o on.

In all of the above algorithms, If a running process requests IO at any time, it should be moved to BLK list until it finishes its IO. Then it should join the shortest RDY list. This implies that a process in the SF RDY list for example can move to BLK list then move to RR RDY list or FCFS RDY list. So, the same process may encounter different scheduling algorithms during its lifetime.

Processes Scheduling Criteria

When a new process arrives, the **scheduler** should move it to the ready queue of the processor with the **shortest queue**. The shortest queue doesn't mean the lowest number of processes. Rather, it means the queue that is expected to finish earlier. This means that you must maintain a timer for each processor queue about the time it is expected to finish all its processes.

The time we are referring to here is the processes CT(so processes IO time is not taken into account)

In our system, some processors serve processes based on FCFS algorithm. Others use SJF and others use RR algorithms. No matter what the algorithm used by the processor is, the schedule always picks the processor with the shortest queue as described above,

Process migration (applicable for process in RUN state only)

Processes can migrate for one processor to another based on the following criteria:

- RR process should migrate to a SJF queue if that process has a remaining time to finish less than a certain threshold **RTF**. RTF should be loaded from the input file. (See File format section below)
- FCFC process should migrate to a RR queue if the process total waiting time so far is greater than a certain threshold **MaxW** (to be loaded from then input file too)

I calculate the waiting time as this waitingTime=CurrentTimeStep-AT-CurrentCPU

In both cases, when the processor picks a process to run, it first checks if it should migrate or not. If so, it should initiate the migration procedure. Otherwise, it runs normally. More than one process can migrate from the same processor at the same timestep.

Work stealing:

With a work-stealing approach, the shortest ready queue in the system looks at the longest ready queue to see how full it is. If the longest queue is (notably) more full, the shortest one will 'steal" one or more process from it to help balance load

Stealing Rules:

- The shortest/longest queue is measured according to the expected finish time of the queue.
- The initiate it, the Steal Limit must be at least 40%. This can be calculated as:

$$\boldsymbol{StealLimit}(\%) = \frac{\boldsymbol{LQF - SQF}}{\boldsymbol{LQF}} > \mathbf{40\%}$$

Where **LQF** is longest queue finish time, **SQF** is shortest queue finish time.

- The process to be stolen are the ones at the top of the longest ready queue.
- The short queue steals one process then recalculates Steal Limit and if it is still greater than 40%, it steals another and so on fill the limit goes beyond 40%.
- This stealing action is performed each STL time (to be loaded from the input file).

Process Migration and Work Stealing is NOT applicable for forked processes. will be handled later

Killing a process: **(applicable for FCFS scheduling only)**

- A process should be killed if it receives a SIGKILL (kill signal)
- This is applicable for processes in **RDY or RUN states** ONLY and scheduled by **FCFS** algorithm.
- When a process is killed it should be moved to the TRM list immediately.
- The time at which SIGKILL is sent to some processes is loaded from the input file.
- 1If the process to be Killed by SIGKILL is not in a FCFS RDY/RUN list, ignore the signal.

Process Forking: **Parent and Child relationship.**

(Applicable for FCFS scheduling only)

A process can **fork** (create) another process. The forking process is called the parent process and the forked one is called the child.

Rules related to forking:

- Only a process in RUN state and being scheduled by FCFS algorithm can fork
- For simplicity, a process can fork only once during its lifetime.
- When a process asks its processor to fork a child process at time T, the child process should be created with new unique ID and with **AT = T and CT_{Chiled}=Remaining CT_{Parent}**

- The processor then asks the scheduler to create a new process. The scheduler should create it and add it to the shortest RDY queues of FCFS processors ONLY.

- Any child process can be scheduled by FCFS algorithm only.
- A child process can never ask for 1/0 and hence it can never be moved to BLK list.
- The child process itself has the chance to fork another one too.
- You should maintain a list for each parent and its child and grandchildren.
- The fork probability is loaded from the input file. At each time step, you should generate a random number for each individual running process and if it falls within that probability, initiate the fork operation.

Orphan Process:

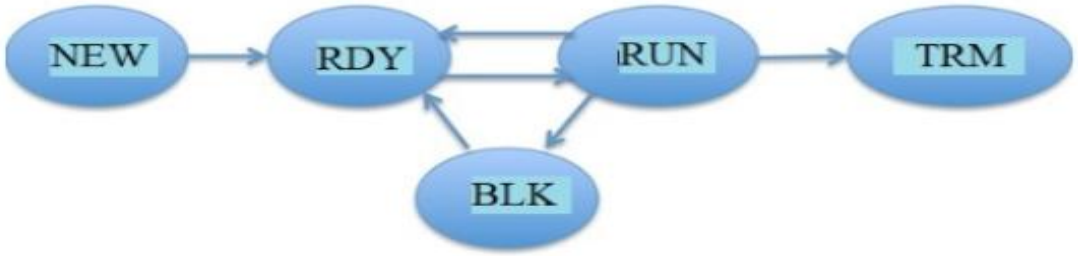
When a child process is forked by its parent, it is moved to the RDY list to be handled later by the processor. There Is a chance that the parent process terminates before its child process. In this case, the child process (and all grandchildren) is considered as **ORPHAN** processes [ORPH]. All orphan processes must be killed immediately and moved to the TRM list.

Notice that according to what is described above, you should find orphan processes in RDY or RUN lists of FCFS processors only.

Simulation Assumptions

- You will use incremental timesteps. Simulate the changes in the system every 1 timesteps.
- A processor can execute only one process at a time.
- Part of a process can be executed by one processor while other parts may be executed by other processors. This situation may occur when a running process is executed by a certain processor, requests I/O and moves to the BLK queue. Then after O it returns to the RDY queue of another processor because Its RDY queue is the shortest at that time. It may also occur due to process migration or work stealing.
- The above point implies that the same process may encounter different scheduling algorithms during its execution.
- Only one process can perform I/O at a time. Other processes requesting IO should wait in
- BLK state till I/O is available.
- 1O requests are handled on FCFS basis.
- A process may request for /O more than one time during its lifetime.
- Process state transition MUST follow the arrows shown on the below diagram:

For example, a process can never go from RDY state to BLK state directly o from BLK state to RUN state directly.



- A process can encounter more than one state transition in the same timestep.

Input/Output File Formats

Your program should receive all information to be simulated from an input file and produce an output file that contains some information and statistics. This section describes the format of both input and output files and gives a sample for each.

The Input File

- First line contains three integers. Each integer represents the total number of processors of each scheduling algo.
    - **NF:** number of FCFS processors.
    - **NS:** number of SJF processors.
    - **NR**: number of RR processors.

- Then a line with one number representing the time slice for RR scheduling.
- Then a line with four integers **RTF, MaxW, STL and the fork probability**

  (Refer to "Process scheduling criteria" section above)

- The next line contains a number **M** which represents the number of processes following this line.
- Then the input file contains M lines (one line for each process).
  NOTE: The input lines of all processes are **sorted by the process arrival time in ascending order.**
  Each process line has the following information:
    - **AT :** Arrival time.
    - **PID:** Process ID.
    - **CT :** CPU time.
    - **N :** number of time the process requests 1/0
    - **(IO_R, I0_D):** pairs of I0_R, 10_D for the process.
- The remaining lines till end of the file shows the times at which SIGKILL is sent.

Sample Input file:

5	**4**	**2**		no. of processors of each type (FCFS, SIF, RR)
3				Time slice for RR.
4	**50**	**100**	**10**	RTF=4, MaxW=20, STL=100, and fork probability = 10%
500	**0**			no. of processes
31	**200**	**2**		**(1,20),(84,17)**
32	**56**	**5**		**(12,7),(34,13),(44,2),(52,5),(56,10)**
53	**134**	**0**		
... and so on for the 500 lines				
SIGKILL Times //Sorted ascendingly by kill time.				
9	**34**			= Kill process with PID=34 at time 9
50	**8**			= Kill process with PID=8 at time 50

The Output File ;

The output file you are required to produce should contain M output lines (a line for each terminated process) of the following format:

TT PID AT CT IO_D WT RT TRT

which means that the process identified by id=**PID** has arrived at **AT**. The line also shows time required for the process on CPU and total 10_D. It also shows the process total waiting, response and turnaround time. (Refer to the "Basic Definitions Section "above)

The output lines **must be sorted** by **TT** in ascending order. If more than one process has the same TT **print them in any order.**

At the bottom of the file, the following statistics should be shown:

1. Total number of processes.
2. Average waiting time, response time, turnaround time for all processes.
3. Percentage of process migration due to RTF and due to MaxW.
4. Percentage of process moved by work steal.
5. Percentage of process fork and kill.
6. Total number of processors and number of processors for each schedule.
7. For each processor, print processor load and utilization.
8. Average utilization for all processors.

Sample Output File

The following numbers are just for clarification and are not produced by actual calculations.

```
TT    PID  AT    CT    IO_D      WT   RT   TRT
50    1    3     20    17        27   2    47
50    10   9     22    10        19   6    41
61  ..................................................... an so on

Processes: 612
Avg WT = 34,      Avg RT = 13,        Avg TRT=78
Migration %:      RTF= 11% ,          MaxW = 23%
Work Steal%: 3%
Forked Process: 5%
Killed Process: 2%

Processors: 5 [2 FCFS, 2 SJF, 1 RR]
Processors Load
p1=12%,   p2=27%,    p3=15%,    p4=23%,   p5=23%

Processors Utiliz
p1=72%,   p2=97%,    p3=80%,    p4=84%,   p5=85%
Avg utilization = 83.6%
```

Appendix A – Guidelines for Project Code

The main classes of the project should be Scheduler, process, processor, and UI (User Interface). In addition, you need lists of appropriate types to hold processes, processors,..etc.

Process Class

Should have data members to store all info of a process. In addition to appropriate member functions. Each process should contain a list to store IO_R, IO_D pairs. Think about an appropriate list type for that.

Processor Class

There are many types of processors in the system:(e.g. FSCF, SJF,... etc). You should create a base class called "Processor" that stores data common to all processors and common functionality. It should be an abstract class with a pure virtual function "**ScheduleAlgo**". The logic of the **ScheduleAlgo** function depends on the processor type.
Each processor should have its own RDY list. From this list, function **ScheduleAlgo** picks the next process to run according to the scheduling algorithm of the processor. So this function handles moving processes from RDY to RUN lists and vice versa.
What about the SIGKILL list, in which class should you declare it as a member?

Scheduler Class

This class is the maestro class that manages the system.
It should have an appropriate ***list of processors***. Also it should have appropriate lists to hold processes in NEW, BLK, TRM states.
Among its functionality, it should have member functions to:

1- At program startup, open the input file and load all processes data into NEW list.
2- At each timestep,
   a. It should handle moving processes between different lists according to the rules and criteria mentioned in "Processes Scheduling Criteria" section.
      i. Move processes that has arrived at this timestep from NEW list to the appropriate RDY list
      ii. Move processes requesting I/O from RUN to BLK list.
      iii. Move processes that finish I/O from BLK to the appropriate RDY list.
      iv. Move processes that finish execution from RUN to TRM list.

b. Handle process migrations and work stealing
c. Place forked processes at the appropriate RDY lists.
d. Handle process killing and orphan processes kill.
e. Collect statistics that are needed to create output file
f. Calls UI class functions to print details on the output screen

3- Produce the output file at the end of simulation

UI Class

This should be the class responsible for reading any inputs from the user and printing any information on the console. It contains the input and output functions that you should use to show the status of the system at each timestep.
At the end of each timestep, this class should print the output screen (see Program Interface section) showing what has happened during that step.
The is the only class whose functions can have input and output (cin and cout) lines
Input and output lines must NOT appear anywhere else in the program
This class is NOT responsible for files read/write

How to print the output screen described in "Program Interface" section Above

In each data structure you will use (e.g. queue, priority queue, linked list … etc) , you are allowed to add a member function that loops on its data and prints it.

You may also need to override << operator on classes process and processor to make it print their IDs. You may use friendship here.

Then in UI class, add a member function that takes as parameters pointers/references to these lists. This function prints the output in the required format. When it needs to print and of the lists passed to it, it should just call the print function of each list.

Note:

You are allowed to add a tail pointer to the linked list. You are also allowed to add a counter data member in different lists to be able to print their count to the O/P window.