Problem Set 3 - Scheduling

1. Reminders

A scheduling policy allows to choose from a set of ready tasks that request access to the processor. Some policies assign a priority to each task, others don't.
With batch processing, a running task cannot be stopped by the arrival of another task nor by a clock interrupt (*eg FCFS, SJN*). Preemptive scheduling interrupts the running task if a new task with a higher priority arrives in the system (*eg PSJN*).
In a time-sharing system, the elected task acquires the processor for (at most) a quantum of time.
A good scheduling algorithm must prevent **famine** at all costs. Famine occurs when the election criteria of the policy prohibits the election of at least one of the tasks and therefore blocks its execution.

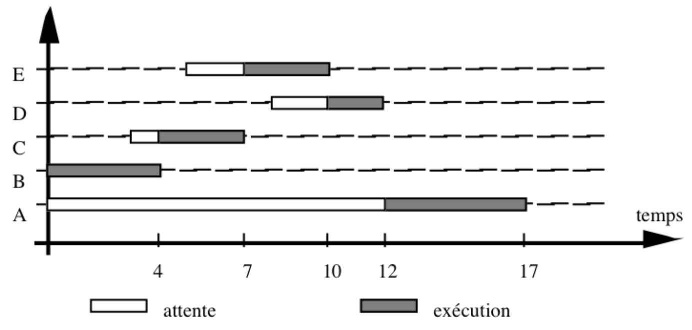
2. Batch Processing

Consider the following set of tasks:

Task	A	B	C	D	E
Time of insertion	0	0	3	8	5
CPU time	5	4	3	2	3

2.1.

The system scheduler enforces SJN (Shortest Job Next). Represent the GANTT diagram of this run. For each task, give its response time (time of completion - time of insertion) and its penalty rate (response time / CPU time). Is there a risk of famine?



Response time

A : 17 B : 4 C : 4 D : 4 E : 5

NB : Response time may not be a good measure of performance, because it does not account for utilization. For instance, tasks B and D incur the same response time; yet B got to use twice as much CPU time as D did. A somewhat better measure is the penalty rate.

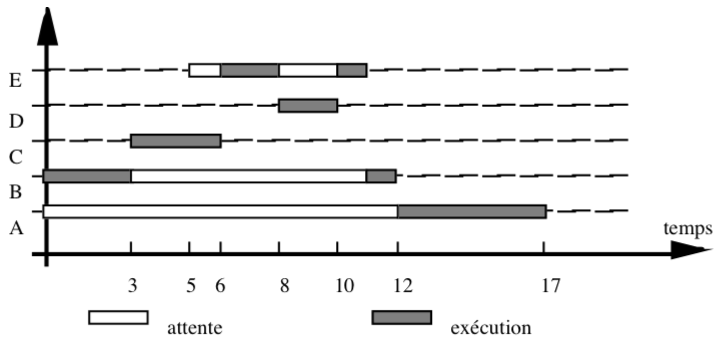
Penalty rate

A : 3.4 B : 1 C : 1.33 D : 2 E : 1.66

Giving priority to shorter tasks strongly discriminates against longer tasks: it breeds famine.

2.2.

Repeat question (2.1) with the PSJN (Preemptive SJN) strategy.



Response time

A : 17 B : 12 C : 3 D : 2 E : 6

Penalty rate

A : 3.4 B : 3 C : 1 D : 1 E : 2

Extending the priority to shorter tasks with preemption pushes the discrimination against longer tasks even further: more famine on the horizon!

3. Timesharing

Round-robin is a circular scheduling policy with a single queue. Any new task gets inserted at the end of the queue. The processor gets allocated to the **first** task from the queue that is **ready**. The elected task gets evicted when it requests an input/output or when it has exhausted its quantum: it is then reinserted at the end of the queue. If a task that requested an I/O reaches the front of the queue before the awaited response, the scheduler leaves it there and looks further down the queue for a task that is ready.
Consider a computer with an exchange unit that allows I/Os in parallel with the CPU.

Let the quantum Q = 100ms.

Consider the following tasks:

	CPU time	I/O	Δ I/O
T1	300ms	none	
T2	30 ms	every 10 ms	250 ms
T3	to 200 ms	none	
T4	40 ms	every 20 ms	180 ms

Initially, the tasks get inserted in the following order: T1, T2, T3, T4.

At t = 30 (t = 40 resp.) Task T2 (T4 resp.) requests an I/O before ending.

3.1

Write a detailed log of the scheduling events in the table below.
The following events can occur: I/O request, I/O response, switch, end of task. Several events can occur at the same date.
Note: it is assumed that the EU manages two disks, and that the I/Os for tasks 2 and 4 can be carried out concurrently on different disks.

Date	Event(s)	Elected task	State of the queue
0	load T1	T1	1r 2r 3r 4r
100	switch	T2	2r 3r 4r 1r
110	I/O T2, switch	T3	3r 4r 1r 2b
210	switch	T4	4r 1r 2b 3r
230	I/O T4, switch	T1	1r 2b 3r 4b
330	switch	T3	2b 3r 4b 1r
360	end of I/O T2	T3	2r 3r 4b 1r
410	end of I/O T4	T3	2r 3r 4r 1r
430	end of T3 switch	T2	2r 4r 1r
440	I/O T2, switch	T4	4r 1r 2b
460	I/O T4, switch	T1	1r 2b 4b
560	end of T1, switch	none	2b 4b
640	end of I/O T4, end of T4	none	2b
690	end of I/O T2	T2	2r
700	I/O T2, switch	none	2b
950	end of I/O T2, end of T2	none	empty

3.2.

What are the advantages and disadvantages of this scheduling mechanism?
Is there a risk of famine? Why?

Advantage: simplicity
Disadvantage: does not account for priority
Risk of famine? No. As soon as a task is in the ready queue, it will get elected after a finite delay. This is guaranteed because the number of tasks in the system is finite, and no task can jump the queue.

4. Timesharing with static priorities

This type of policy associates a priority with each task upon its creation. The priority of a task remains the same until its termination.

4.1.

Consider a system that enforces round-robin on a multilevel queue with 4 priority levels numbered from 0 to 3 (0 being the lowest).

Repeat question 3.1 with this new policy for the following tasks:

	CPU time	I / O	Δ I/O	Priority
T1	300 ms	none		0
T2	30 ms	every 10 ms	250 ms	3
T3	200 ms	none		1
T4	40 ms	every 20 ms	180 ms	3
T5	300 ms	none		1

Date	Event(s)	Elected task	State of the queue
0	load T2	T2	2r 4r / / 3r 5p / 1r
10	I/O T2, switch	T4	4r 2b / / 3r 5p / 1r
30	I/O T4, switch	T3	2b 4b / / 3r 5p / 1r
130	switch	T5	2b 4b / / 5p 3r / 1r
210	end of I/O T4	T5	2b 4r / / 5p 3r / 1r
230	switch	T4	4r 2b / / 3r 5p / 1r
250	I/O T4, switch	T3	2b 4b / / 3r 5p / 1r
260	end of I/O T2	T3	2r 4b / / 3r 5p / 1r
350	switch, end of T3	T2	2r 4b / / 5p 3r / 1r
360	I/O T2, switch	T5	4b 2b / / 5p / 1r
430	end of I/O T4, end of T4	T5	2b / / 5p / 1r
460	switch	T5	2b / / 5p / 1r
560	End of T5, switch	T1	2b / / / 1r
610	end of I/O T2	T1	2r / / 5p / 1r
660	switch	T2	2r / / / 1r
670	I/O T2, switch	T1	2b / / / 1r
770	switch	T1	2b / / / 1r
870	end of T1, switch	none	2b / / /
920	end of I/O T2, end of T2	none	empty

4.2.

What are the advantages and disadvantages of this scheduling policy?
Is there a risk of famine? Why?
Advantage : extends RR with priorities
Disadvantage: introduces famine, because a low priority task will never get elected if higher priority tasks keep arriving in the system.

5. Timesharing with dynamic priorities

This type of policy reassesses priorities as the system evolves. There are many possible priority reassessment strategies. Let us consider a multilevel feedback queue strategy where every new task gets inserted at the end of the queue of highest priority. Upon eviction, a task gets inserted at the back of the queue that is one priority level lower than the queue it got elected from.

Consider the following task insertion model:

	CPU time	Insertion rate
T1	15 ms	every 150 ms
T2	200 ms	every 300 ms
T3	1000 ms	-

Assume that at date d = 0, tasks T1.0, T2.0, and T3.0 enter the system in that order. Whenever a task T1.x enters the system at the same date as a task T2.y, T1.x always precedes T2.y in the queue. Let the quantum Q = 100ms.

5.1.

Repeat question (3.1) with your new policy from date d = 0 ms to date d = 900 ms

Date	Event(s)	Elected task	State of the queue
0	load 1.0	1.0	1.0 2.0 3.0
15	end of 1.0, switch	2.0	2.0 3.0
115	switch	3.0	3.0 / 2.0
150	new (1.1)	3.0	3.0 1.1 / 2.0
215	switch	1.1	1.1 / 2.0 3.0
230	end of 1.1, switch	2.0	/ 2.0 3.0
300	new (1.2 ; 2.1)	2.0	1.2 2.1 / 2.0 3.0
330	end of 2.0, switch	1.2	1.2 2.1 / 3.0
345	end of 1.2, switch	2.1	2.1 / 3.0
445	switch	3.0	/ 3.0 2.1
450	new (1.3)	3.0	1.3 / 3.0 2.1
545	switch	1.3	1.3 / 2.1 / 3.0
560	end of 1.3, switch	2.1	/ 2.1 / 3.0
600	new (1.4 ; 2.2)	2.1	1.4 2.2 / 2.1 / 3.0
660	end of 2.1, switch	1.4	1.4 2.2 / / 3.0
675	end of 1.4, switch	2.2	2.2 / / 3.0
750	new (1.5)	2.2	2.2 1.5 / / 3.0
775	end of 2.2, switch	1.5	1.5 / 2.2 / 3.0
790	end of 1.5, switch	2.2	/ 2.2 / 3.0
890	switch	3.0	/ / 3.0

5.2.

Will T3.0 ever terminate?
Will this always be the case, regardless of the insertion model? Give an example.
Propose a scheduling algorithm which (a) favors tasks that spend all their time quantum and (b) prevents famine.

T3 is always ready, so the processor is never idle. During a cycle period of 600ms, T1 tasks and T2 tasks consume 60ms and 400ms of CPU time respectively. This leaves 140ms of running time for T3. T3 will therefore terminate, and we can determine that its date of termination is bounded by 1000 / 140 * 600 ms ~ 4.3 s.

Once T3 reaches the 3rd priority queue, it will always get overtaken by T1 tasks (which always remain at the highest priority level) and by T2 tasks (which never get demoted lower than the second highest priority level). Thus, if there is always at least one T1 or T2 task ready, T3 will never get elected. For instance, this can happen if T1 tasks consume 50ms of CPU time instead of 15.

New strategy:

(a) Add a counter to keep track of the CPU time consumed by a task within its quantum, and charge it to the task upon its next election. For example, if a task consumes only 30ms out of a 100ms quantum, it will only get a 70ms quantum the next time it gets elected.
(b) Only demote a task when it consumes a whole quantum, otherwise it creates famine for interactive tasks. Add graceful ageing: limit the number of priority levels, and promote tasks back to the highest priority when they've consumed their quantum at the lowest priority level.
(c) Periodically (very large period) return all tasks to the highest priority to make sure no task gets stuck in the lower priority levels when the insertion rate is high.