CS29206 Systems Programming Laboratory
Spring 2024
Lab Assignment: 6
Date: 21–Mar–2024

Programming in gawk

A single-CPU computer in Fooragpur Informatics Center (FIC) runs processes from three types of users.

- **Government users:** Processes from these users have the highest priority (1).
- **Paid users:** Processes from these users have intermediate priority (2).
- **Free users:** Processes from these users have the least priority (3).

A process starts with some computations in the CPU, then does some IO, again does some computations in the CPU, then some IO, and so on. Each computation period in the CPU is called a *CPU burst*, whereas each IO operation is called an *IO burst*. A process starts with a CPU burst. Subsequently, IO and CPU bursts alternate. We assume that each process ends with a CPU burst. Each process has a name (unique), a priority (1, 2, or 3), an arrival time, and a sequence of CPU and IO bursts. In reality, the arrival times and the burst times (also the numbers of bursts) of processes are not known beforehand. For the sake of simplicity, assume that these quantities are supplied to you in a file `proc.txt`. Each line in this file stores the details of the process as follows.

$$\texttt{Name,Priority,Arrival,}\texttt{CPU}_1\texttt{,}\texttt{IO}_1\texttt{,}\texttt{CPU}_2\texttt{,}\texttt{IO}_2\texttt{,}\texttt{CPU}_3\texttt{,}\texttt{IO}_3\texttt{,...,}\texttt{CPU}_k$$

Here, `Name` is the name of the process (a string, distinct from one process to another), `Priority` is an integer among 1, 2, 3, `Arrival` is the time (a non-negative integer) when the process arrives at the system. This is followed by the would-be burst times (positive integers) of the process in sequence. Comma is the field separator, and no space appears anywhere in each line (process names do not contain spaces).

When a process does IO, it cannot use the CPU. Other processes that are ready to run (see below) can be scheduled to use the CPU during that period. The FIC machine uses a priority-based non-preemptive round-robin scheduling algorithm. These three terms are explained below. A process is called *ready to run* if it has arrived, has not terminated, and is not doing IO. When a ready process is scheduled, it runs for its next CPU burst time, and then either terminates (if that was its last CPU burst) or starts IO after relinquishing the CPU for other processes.

> **Priority-based:** When the CPU is free to schedule the next process, it picks a ready process which has the highest priority among all the ready processes at that moment. For example, a priority-2 process is scheduled only when no priority-1 processes are ready to run, and a priority-3 process is scheduled only when no priority-1 or priority-2 processes are ready to run.
>
> **Non-preemptive:** When a process is scheduled to run, it is allowed to complete its next CPU burst. Arrival or IO completion of a higher-priority process during that period will not stop the running process. The process stops only at the end of its current CPU burst.
>
> **Round-robin:** When multiple processes of the same priority level are ready to run, the scheduler schedules them in the round-robin fashion. We maintain three FIFO queues, one for the ready processes at each priority level. The process that is picked from a queue (the highest-priority non-empty queue) is the one at the front of that queue. When a process arrives or completes an IO burst, it is added to the back of the queue at its priority level.

With this understanding of the system, write a gawk program to show all the scheduling and running of the processes in `proc.txt`, and to compute the average waiting times of processes at the three priority levels. Your program should use an algorithm described now.

First, read the records from the input file `proc.txt`, and store the information in appropriate (global) arrays. Since process names are unique, you can use the process name as an index in the arrays. But then, the arrays will be associative arrays, but that does not matter, because the access syntax for associative arrays is identical to that for integer-indexed arrays (in gawk).

Maintain three FIFO queues, one for the ready processes at each level, that are waiting for the CPU. The data structure that you use for these queues is left to you. That need not be time- or space-efficient, but should correctly maintain the FIFO order.

In reality, the working of the system (involving the CPU, the scheduler, and the processes) is governed by the occurrence of events (typically triggered by interrupts to the CPU). In this assignment, you *simulate* the events. Take the following approach. Maintain an *event queue* EQ to store the forthcoming events that change the system configuration. Between two consecutive events, nothing significant happens. For example, when a scheduled process is in its current CPU burst, nothing happens in the simulation until one of the following events happens: (1) a new process arrives, (2) an existing process in its IO burst completes the IO, and (3) the CPU burst of the running process ends. In fact, these are the only events that you need to handle. Note that the events must be handled chronologically, that is, you must maintain a current time. The occurrence of the next event changes the current time to the time of occurrence of that event. As mentioned earlier, nothing of concern happens between two consecutive events. For the working of the algorithm, you must know the precise times at which the events will occur. Place an event in EQ as soon as its time of occurrence can be precisely determined.

Initially, only the arrival times of the processes are known from the input file `proc.txt`. Before running the simulation, you do not know when each CPU or IO burst of a process will start or finish. However, if the start time of a burst is determined, its finish time is known immediately, because in the FIC system, both CPU and IO bursts run without any interruption. What is not known beforehand is when each CPU burst of each process is going to be scheduled, because that depends on the existence of other processes at the same or other priority levels.

A pseudocode of this event-driven simulation algorithm is presented on the next page. It is evident that EQ needs to be implemented as a priority queue. You may implement EQ as a (binary) heap or a sorted array (the choice is yours). Each event is a 3-tuple consisting of a process name, the type of the event, and the time when this event occurs. Note the following precise ordering of two events:

> Events that happen earlier appear earlier in EQ.
> For events happening at the same time:
> > Use the ordering of the event type:
> > > ARRIVAL < END-OF-IO-BURST < END-OF-CPU-BURST
> > If that too does not break a tie, use the process names under dictionary ordering

Since no two processes can have the same name, this specifies a unique ordering of the events. Note that priorities of the processes do not matter in sequencing the events (priorities even do not need to be stored in an event). Processes of different priorities are already distinguished by the three separate ready queues, and the scheduler is programmed to respect this ordering.

Waiting time of a process

The waiting time of a process is the total time it spends in its ready queue. This is the sum of all the amounts of time between entering the ready queue and getting scheduled for its next IO burst. Waiting for the completion of an IO request is not to be considered in the sum.

Sample

A sample will be provided in the course web site. Follow that style strictly in your program's output.

The event-handler algorithm

Initialize EQ to contain the arrival events of all the processes, and the state of the CPU to IDLE.
So long as EQ is not empty, repeat:
Extract the next event *e* from EQ. Let *P* be the process that caused *e*.
Based upon the type of the event, do the following:
Case 1: *e* is an ARRIVAL event
Based on the priority of *P*, enqueue *P* to the back of the appropriate ready queue (the first burst of each process is a CPU burst).
Case 2: *e* is an END-OF-IO-BURST event
Based on the priority of *P*, enqueue *P* to the back of the appropriate ready queue (*P* is now ready to do its next CPU burst).
Case 3: *e* is an END-OF-CPU-BURST event
If that was the last CPU burst of *P*, the process terminates,
else do the following:
- *P* will (immediately) go to its next IO burst, and the time when this IO burst will finish can now be computed. Insert a suitable END-OF-IO-BURST event to EQ.
- Mark CPU as IDLE.

If the CPU is IDLE, do the following:
If there are no ready processes in any of the three ready queues,
the CPU will stay IDLE until the next event occurs,
else do:
- Pick the highest-priority non-empty queue.
- Extract the process *N* from the front of that queue.
- Schedule *N* to finish its next CPU burst. Compute the time when this CPU burst will end, and add an appropriate END-OF-CPU-BURST event to EQ.
- Mark the CPU as BUSY.

What to submit

Submit a single gawk executable file (with an appropriate hash-bang notation at the first line). In your file, write the functions that you use (you are free to write suitable functions), and then the BEGIN block, a single reading block, and finally the END block.

Tips on gawk programming

- Different processes may have different numbers of bursts. The built-in variable `NF` tells you the exact number. Run a variable `i` over the burst-list indices (in the current record), and access the burst times as `$i`. By gawk, `i` will be evaluated first, and then `$i`.

- Elements of a two-dimensional array can be accessed as `A[x,y]`. Here, each of `x` and `y` can be either a positive integer or a string.

- The built-in function `sprintf` works as follows:

```
string = sprintf(format_string, list of expressions)
```

  This is unlike C, where `string` is specified as the first argument of `sprintf`.