CSE 489 Final Report: Analogous CPU

MATTHEW C. LINDEMAN,
New Mexico Institute of Mining and Technology, USA

This project is meant to be an environment from which the user can test scheduling algorithms to deduce logical statements about them in a reasonable manner. A reasonable manner in this instance, is a way in which the only interface for the user to interact with is the implementation of the scheduling algorithm itself in native C code. In this manner, roughly, scheduling algorithms can be judged based off of performance with respect to a particular feature. For example, scheduling algorithm X may be better at handling IO processes in short bursts. This project is the collection of code that represents the base unit of execution, process list in a manner from which data can generically be piped in and out, and the data collection elements bassed of the base unit of execution. Additionally, as a proof of concept, I have implemented the lottery scheduling algorithm and used the system to collect data over it which I will present as an example of what the system can do with respect to analysis.

Please see the source code as it was intended to be veiwed by the developer (complete with automatic markdown reader) by accessing https://github.com/millipedes/Scheduling-Simulator with a modern non-text based browser.

Additional Key Words and Phrases: CPU, Operating System Scheduling, Process Scheduling

ACM Reference Format:
Matthew C. Lindeman. 2022. CSE 489 Final Report: Analogous CPU. *J. ACM* 1, 1, Article 1 (March 2022), **??** pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 OVERVIEW OF THE PROJECT

For a visual reference please see the uml document hosted under the source project directory `documentation/plant/uml_figure.png`.

1.1 Fundamental Structure

For the system to be opaque to the user, a scheduling algorithm must be well defined. Thus we will define it using three variants: α, β, and γ. Which are defined respectively to be the following:

(1) α - the hardware environment which the algorithm runs on (analogous to the base unit of execution in the project as work is measured as a scalar not a vector).
(2) β - The characteristics that the scheduling algorithm associates with each process (this can be the empty set and in

Author's address: Matthew C. Lindeman, matthew.lindeman@student.nmt.edu, New Mexico Institute of Mining and Technology, 801 Leroy Pl, Socorro, New Mexico, USA, 87801.

0004-5411/2022/3-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

the code is analogous to the process and process_list data objects). The architecture of this project allows for dynamic β. The way that this was implemented was using the relationship between process and proc_list. Each of these structures have generic ports to custom data types. This is where work quantity being a scalar not a vector becomes useful as regardless of the data types required by the algorithm, the end quantity of work being performed (what is recorded by each process) can be ported to this scalar work quantity which is universal.

(3) γ - The scheduling algorithm itself (this is the primarily interesting variant, however to show the functionality of the system I have implemented an instance of the lottery scheduling algorithm with the project infrastructure).

Using these variants it becomes clear that a system capable of handling a varying α, β, and γ in extensible C code must have the following:

- Have a process tracking infrastructure capable of communicating with a generic interface, both storing and writing data.
- The units of the process tracking system must be capable of having varying attributes.
- A generic and universal way of representing work.

The uml diagram contains a full accounting of the project, but some of the more fundamental and high level objects can be used to explain the functionality of the system.

Some of the genericity of the following subsystems have yet to be implemented in the project, as currently the lottery scheduling algorithm has been hard coded with respect to types in the source code. These can be replaced by void pointers instead to achieve this genericity and the code functions in the same manner.

cpu_t - This is the interface with the direct work of the system (i.e. how many and or to what degree have processes been completed for a given time quantum). The genericity is achieved here by having a generic unit of execution. I.e. at runtime it can be assigned to be an ARM architecture processor, x86 architecture processor, etc. as its attribute of execution is user defined.

process - This structure in the code can handle the data provided by the scheduling algorithm in a manner suitable for the particular algorithm. See later in the paper for this explained for lottery scheduling.

proc_list - This structure contains a generic way to interact with the scheduling algorithm's data structures.

lottery/base - The particular γ which was implemented for the project as a proof of concept.

base - In the description of the implementation of the lottery scheduling algorithm, the researchers who wrote the paper mentioned a particular data structure which was used to more efficiently track tickets. This structure involved having pointers from the threads to the currently executing process's tickets which in turn had pointers to processes with pointers to tickets and this process is repeated until the process is pointing the the base structure of currency.

2 LIST OF SUPPORTED VARIANTS

The following are the variants that can be changed that would be of use to the user. Additionally, the structural extensibility of the project is discussed. Any all-caps reference is a definition under the source project file: `src/constants_macros/include/constants.h`.

2.1 α

Structural Extensibility:

- The pointer between cpu_t and what is currently thread can be modified to change the way in which tasks are handled (i.e. a cpu could point to another cpu changing the way it executes the scheduling algorithm).

Data Variants:

- The number of threads that the given CPU is utilizing. (THREAD_NO).
- The scalar quantity of work that each thread is executing per time quantum (THREAD_WORK).

2.2 β

Structural Extensibility:

(1) There exists a pointer from proc_list to process which allows for the downward flow of data.
(2) proc_list and process both void pointer data ports to receive data from a scheduling algorithm.

Data Variants:

(1) The total number of processes that the process list can have (P_LIST_INITIAL_SIZE).
(2) The total proportion of I/O processes to memory related processes (MEM_PROP).
(3) The maximum number of processes generated by the system for each time quantum. It is the maximum as all process quantity generation is random (MAX_NG_PROCS).
(4) The maximum quantity of work that a process requires to be completed. Again, is the maximum as all process work quantity generation is random (MAX_PROC_WORK).

2.3 γ

Structural Extensibility:

- The algorithm implementation can be changed dynamically under the condition it feeds data into the proper ports with the surrounding system.
- The algorithm data structures can be changed dynamically under the condition that it feeds data into the proper ports of the surrounding system.

Due to the nature of β there should exist no variance in the data related to γ.

3 LOTTERY SCHEDULING SYSTEM

As previously mentioned, as a proof of concept of the system itself I implemented the lottery scheduling algorithm as a sample γ (note that the uml diagram is reflective of this). Below I will describe the manner in which α, β, and γ were realized in the system.

3.1 α Implementation

This is done via CPU with threads that have direct work quantity that they fulfil each time quantum.

3.2 β Implementation

This is where this project best illustrates how it is extensible. In order to discuss why, first we must review the way in which the lottery scheduling algorithm determines where its currencies are backed from.

The way in which this was implemented for the lottery scheduling algorithm was by using a base currency structure which interfaces with proc_list via pointers and ticket_bundles which interact with process replicating (with a few extra explicit edges in the representative graph of the data structure) the process described in the lottery scheduling algorithm paper.

3.3 γ Implementation

The entirety of the lottery scheduling interface itself can be found in Appendix figure 1. The process list can be used to alter and get data, but under the hood it is the user defined base and ticket_bundle structures that fundamentally store the data.

4 EXAMPLE EXPERIMENT USING THE SUBSYSTEM

4.1 Premise

Suppose a user wanted to know the approximate threshold at which the quantity of processes will either be completed per each time quantum by the processor and the threshold at which the number of processes tends towards infinity. From the system documentation we can see that there are several relevant parameters, but this particular user is interested only in the process's data impacts on these thresholds. This narrows it down to two vartiants:

- Process Data Variant - MAX_NG_PROCS (the max number of processes that can be generated given a particular time quantum).

- Process Data Variant - MAX_PROC_WORK (the maximum amount of work that it will require to complete a generated process).

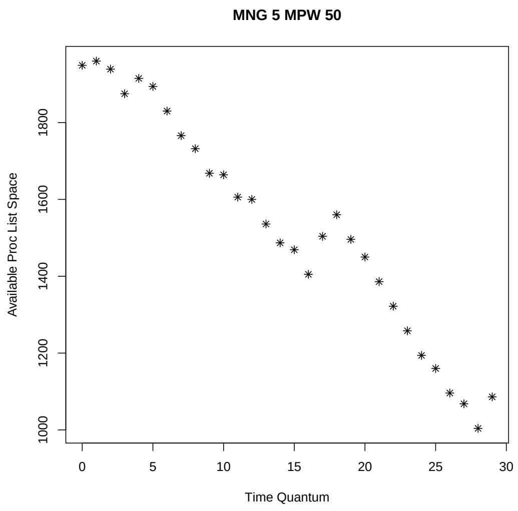
Now that the variants are decided upon, the user will chose which aspect(s) of the system the wish to examine in relationship to these variants. For this example, we will observe the affect of these variants on the proc_list across time quanta.

As it would be standard to do (and this project makes trivial) we must state our assumptions:

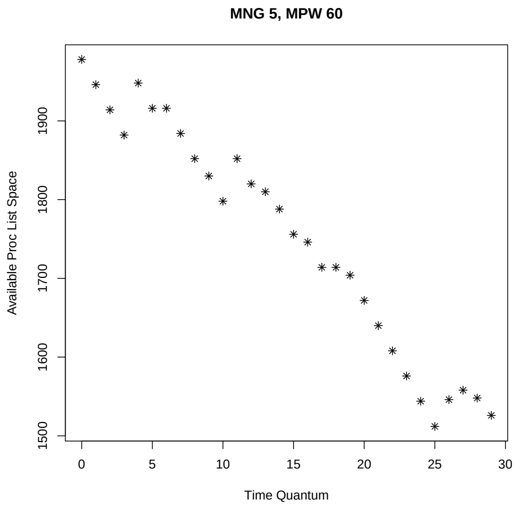
(1) Local system (i.e. single CPU)
(2) 8 Threads
(3) 2000 Max process list size
(4) Threads Perform 30 Work per Time Quantum

Let us conduct the following experiments with the above assumptions:

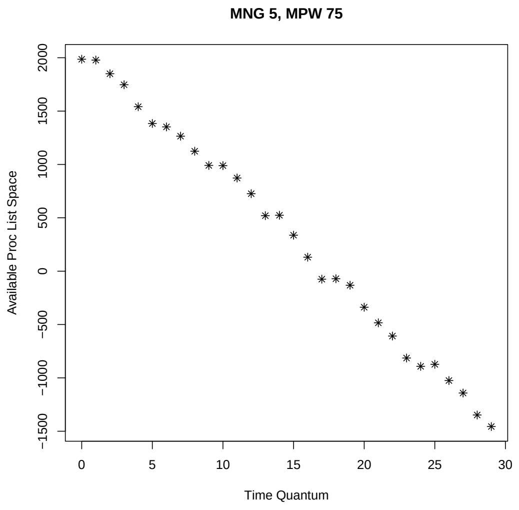
Test #	1	2	3	4	5
Max New Gen Procs	5	5	5	10	15
Max New Proc Work	50	60	75	75	75



This is the first test for which processes are introduced with low maximum potential work to be completed, and low maximum number of processes generated each generation. The user can see that this is unsustainable over time and that the available space will tend to 0.

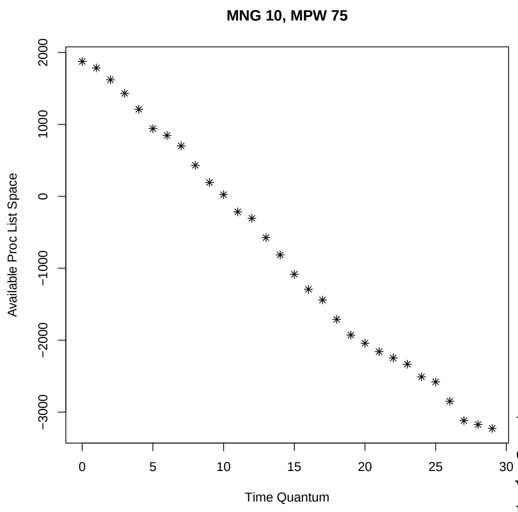


This is the second test for which processes are introduced with slightly higher number of processes being generated each time quantum. The user can see that this is unsustainable over time and that the available space will tend to 0 even faster then when the potential work was lower.

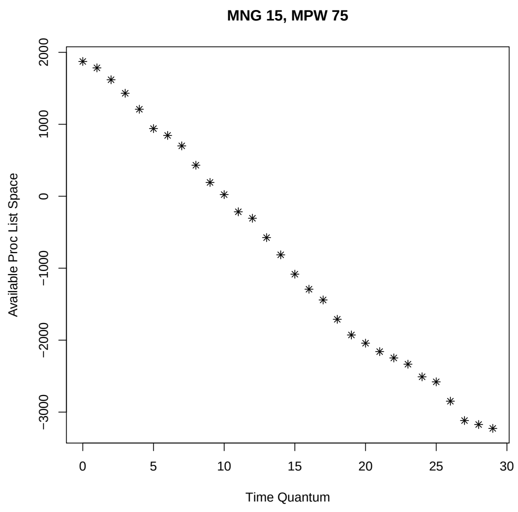


This is the third test for which processes are introduced with higher new generation process generation. The user can see that this is unsustainable over time and that the available space will tend to 0 even faster then when the potential work was lower.

Please note that available space becoming negative was a design oversight, as programmed, negative process list space is indicative of processes happening faster than the CPU can process them.



The fourth test. We can see that with a high number of process being generated each time quantum and a high potential work to complete each process that the CPU will be overwhelmed quickly.



The fifth test. Similarly to the fourth can see that with a high number of process being generated each time quantum and a high potential work to complete each process that the CPU will be overwhelmed even quicker.

5 CONCLUSION

The results from the test experiment seem reasonable. Additionally, the interface between γ and the system is just a matter of coding the implementation of the scheduling algorithm and its respective data structures in C then modifying conduct_test(). For these reasons, I would argue that this tool accomplishes, at least at a base level, its goal.

6 PROJECT NEXT STEPS

The next step would be 100% implementing the genericity to the clients that handle the work execution (i.e. tweaking the way that the cpu interacts with the scheduling algorithm).

Additionally, implementing other algorithms to give some comparison among γ would be a high priority for this project in the future.

7 FINAL SCOPE OF PROJECT

- 1350 lines of C code

8 REFERENCES

(1) Waldspurger, Weihl, Lottery Schedduling: Flexible Proportional-Share Management

(2) Moring, Schultz, Uetz, Approximation of stochastic scheduling: the power of LP-based piority policies.
(3) Scharbrodt, Schickingera, Steger, A new average analysis for completion time scheduling.
(4) Rothkopf, Scheduling with random service times.
(5) Weiss, Approximation results in parallel machines stochastic scheduling
(6) Megow, Uetz, Vredeveld, Models for stochastic online scheduling.
(7) Skutella, Uetz, stochastic Machine Scheduling with precedence constraints.
(8) Chandy, Renolds, Scheduling partially ordered tasts with posibilistic execution times.
(9) Graham, Lawler, Lenstra, Kan, Optimization and approximation in deterministic sequencing and scheduling: a servey
(10) Allahverdi, Gupta, Aldowaisan, A reveiw of scheduling reasearch involving setup consideration.

9 APPENDIX

```
void host_lottery(thread * t, proc_list * pl) {
        time_t ti;
        srand((unsigned) time(&ti));
        int windex = (pl− >b− >total_space - pl− >b− >available_space);
        int winner = 0;
        if(windex > 0) {
                winner = rand() % windex;
        }
        int reduction_index = find_ticket_partition_process_index(pl, winner);
        if(pl− >b− >bid− >size > 0 && pl− >p_list[reduction_index]− >tb) {
                reduce_bundle(pl, t− >work_qty, pl− >p_list[reduction_index]− >tb− >id);
        }
}
```

Figure 1