Overview

This document serves to characterize the high level design of the CPU scheduler discrete event simulation (DES). A discrete event simulation is one that attempts to codify and categorize the behavior of complex systems as an ordered sequence of well-defined events. Much of this document relies on the description of the system found in the assignment and event descriptions provided. This document assumes these description as a priori knowledge. The DES system will be implemented with various goals in mind, prioritized in the following order:

Correctness The simulation should be correct. The mechanics of the simulation should run without error and should logically implement the system description. The statistics, counts, and other analysis-related data should be correctly stored and reported.

Simplicity The DES should be simple and intuitively designed. Explicit code should trump clever code. The design hierarchy should be reproducible and understood from a high level with ease. Code should be well documented and standardized (simple to understand).

Modularity The design of the system should be modular. Definitiions and functionalities of each moving part should be well-defined and not unexpectedly change. The DES will be objected oriented. Classes, data, and functional procedures should be as stand alone as possible; avoid strong coupling between elements of the class hierarchy and general flow of system.

Scalability The system should be scalable for plausible improvements or additional functionalities. Adding new moving parts should not be impossible or require significant overhaul in order to implement.

Efficiency When possible/sensible, the system should be efficient. Avoid nested looping structures or oppressively iterative computation. Vectorize when possible and carefully weigh each implementation so that speed or storage does not become an issue.

See below for a detailed description of the structures that describe the components of the DES system and further documentation regarding some of the mechanics of the components' interactions and other routines/procedures that may be of interest.

System Description

This section describes the DES system's components, their behavior and interactions, and other procdures of interest in the system. Each of these objects will have access to a global set of variables, namely several of the command line arguments entered to initialize the system:

`QUANTUM` The length of a time slice in the system.

`SWITCHTIME` The time penalty to perform a context switch.

`STOPTIME` The maximum timestep the system should run until.

Components, behavior, interactions

Below is a description of each structure/class of that comprises the DES system. The organization of these descriptions roughly starts with the objects that system **interacts with** and increases **in order of authority** downwards. It's important to note that the design of the system could be subject to change in an effort to better accomplish some subset of the goals described in the Overview section.

Process

A **Process** represents a single process in the system. This class is essentially a package of various data points that is shuffled around the system and manipulated as the simulation marches forward. An instance of a **Process** p the following attributes, initialized upon construction:

p.`type (const, string)` The process name for *p*. Passed in as argument.

p.`burst_cpu (const, int)` The length of time *p* sprints on the CPU. Passed in as argument.

p.`burst_io (const, int)` The length of time *p* takes to perform I/O. Passed in as argument.

p.`demand (int)` The current number of timesteps that *p* has left to run before exiting the system. The system will update this value appropriately as it is processed on a CPU. Passed in as argument.

p.`cpu_current int` Initialized as a null field. The current CPU on which *p* is running. If *p* stands in the ready queue, then *p*.`cpu_current` is null.

p.`arrival_time (const, int)` The time that *p* entered the system.

p.`next_arrival int` The timestep of the next arrival of a new process with the same `type`. Value is set on construction. If there `next_arrival` is -1, then a new process of such a `type` will not be initialized for the rest of the duration of the system. As *p* leaves the system (finishes being processed), a new process with the same `type` will enter an `arrival_time` of `next_arrival`

Processes will appear in many parts of the program: scheduled on **CPU**s, standing in the ready queue, referenced by **Event**s, but they will be modified/moved by a single governing source, the **Clock**.

Event

A **Event** e_t represents the event that occurs at time t. Event e_i is composed of two attributes, a **Process** p and a integral timestep t. The attribute p is necessary so that when e_t is dequeued, we can easily access and manipulate the process that is responsible for e_t. The attribute t is the priority of e_i. It is placed in the EventQueue based on t.

ProcessFactory

A **ProcessFactory** creates new processes that will be added to the ready queue. The major point of the **ProcessFactory** is that it contains an associative array `procmap` (hashmap, dictionary, or something similar) that maps process types to their respective average CPU and I/O burst times, average interarrival times, and average overall CPU demand durations. The **ProcessFactory** has single method called `observe`(*stringprocType*) that returns an instance of a `Process`, where each of its fields are initialized to values pulled from random distributions (exponential or uniform, depending on the field) with centers found in `procmap`[*procType*]. The **ProcessFactory** is also responsible for reading the procedure generation file and cataloguing the various types of processes.

Queue

The abstract base classes for the ready and event queues. Contains objects of some arbitrary type.

ReadyQueue

A simple linked list of **Process**es. Extends **Queue**. Holds and releases **Process**es for dispatch into the system.

EventQueue

The queue of **Event**s. This child of **Queue** is always in sorted order with respect to the priority of each **Event** that resides within it.

CPU

A **CPU** is an object that will directly interface with **Process**es pulled from the **ReadyQueue**. As such, the **CPU** has a member variable called `proc` that represents the current process running on this **CPU**. The **CPU** object has a single method called `update()` that looks at the current timestep of the simulation and updates various fields of `proc` accordingly. The context switch time is factored into the processing time of `proc` by considering it in the update equation for the various fields of `proc`. The **CPU** object also contains a few running counters for post processing statistics such as `time_spent_idle`, `total_context_switch`, and possibly others.

Clock

The **Clock** object drives the system. The **Clock** object contains a current time `timestep` that describes the temporal position of the simulation and runs until `timestep` reaches or exceeds STOPTIME, the program ends. The **Clock** contains the logic that dispatches **Process**es from the **ReadyQueue** to some **CPU**, the logic that generates new events based on the I/O, burst, and demand regimes of incoming **Process**es, and contains the code responsible for calling each of the CPU `update` methods. The **Clock** object will also be responsible for reporting verbose outputs.

Tracker

This is element of the system may collapse into the **Clock** object detailed above but for the sake of modularity it will be described as a separate entity. This object contains tracking statistics about the system over the course of the simulation, such as the length of the ready and event queues at time t, the total list of **Event**s/**Process**es that took place, etc.

Procedures and routines

There are two routines that will be highlighted in this section, the startup and shutdown procedures. On startup, the input arguments are first validated and sanitized. Subsequently, a **ProcessFactory** reads and catalogues the contents of the input procedure generation file. Seed processes for each type of process are generated and added to the ready/event queues. From there the system proceeds as described, instantiating new processes, dispatching to **CPU**s, recording information, etc. At shutdown, either by keyboard interrupt or normal termination, the program will write to disk some version of an ad-hoc summary of the simulation for later analysis.

Sources

- Discrete Event Simulator Description
  https://whatis.techtarget.com/definition/discrete-event-simulation-DES