CPU Scheduling Simulator (CPUSS)

Granville Barnett

April 3rd, 2008

This document applies to CPUSS version 1.0

1 Overview

This document is a set of brief notes about what CPUSS offers and how to use the CPUSS framework, and associated tools.

CPUSS is a framework that allows you to gather metrics on algorithms used to schedule the next process to utilize the CPU. CPUSS can be extended so you can implement your own algorithms and harness the metrics engine offered by CPUSS so you can perform further analysis on your algorithm.

The distribution of CPUSS includes three items of note: Cpuss.dll (core framework), Cpuss.Strategies.dll (implementations of established and research strategies), and finally cpussrg.exe, the latter of which is a command line tool used for generating reports for load generated scenarios against your strategy.

CPUSS 1.0 is built using C# 3.0 and makes extensive use of property initializers, and lambda expressions, there are no plans to support earlier versions of the language.

The actual source code contains six projects: Cpuss, Cpuss.Tests, Cpuss.Console, Cpuss.Console.Tests, Cpuss.Strategies, and Cpuss.Strategies.Tests. The test projects have a dependency on NUnit[1], if building from scratch I advise that you download the project, build it and then run all the tests in debug mode before compiling release builds. I strongly recommend using the precompiled binaries especially if you are creating plugins as described later as you will need to build against the latest release version of Cpuss.dll.

2 Using the CPUSS Framework

The CPUSS framework is very simple to use and provides many hooks for further extension.

I will only cover a very brief amount that the CPUSS framework offers as I have ensured that the CPUSS core is very well documented, so for a more comprehensive description of the API please see the compiled help file that ship's with every release.

2.1 IStrategy interface

In order to plug your algorithm into the CPUSS framework you must implement the `IStrategy` interface, this interface contains only two methods.

Figure 1: IStrategy interface

```
namespace Cpuss
{
    public interface IStrategy
    {
        void Execute(Runner runner);
```

[1] http://www.nunit.org

```
        string ToString();
    }
}
```

The more important of the two methods is the Execute method. This method should contain the logic for your algorithm.

2.2 Setting up the execution environment

If you are purely using CPUSS for research into development of your own scheduling algorithm then you need really only be familiar with the IStrategy interface and the Runner type.

The Runner type provides the black box for which all computation takes place in terms of metrics tracking of each process; it also provides supplementary functionality like events and functions to aide with rapid data analysis.

A Runner instance expects only two arguments: processes for the algorithm to schedule, and finally the algorithm to schedule those processes.

Figure 2: Runner type

```
Runner runner = new Runner(processLoad, new FirstComeFirstServed());
```

The first argument is a collection of type `ProcessLoad`, this collection is rather self explanatory it is a collection of processes, the strategy to use must implement the `IStrategy` interface.

The `Runner` type is actually overloaded, the overloaded constructor takes four parameters, the first three of which represent the number of small, medium and large processes respectively to run. These processes have varying properties that are randomized, e.g. the burst time of a medium sized process will lie between two set boundary points, the arrival time will be within two fixed points as well, and similarly the priority of the process will be randomized between the available values. This is particularly useful when wanting to gather results on your algorithm that represent a more broad and varying set of processes, it can be further enhanced by using the `RepeatRunner` type which will aggregate key data from each of a series of runs.

2.3 Invoking a simulation

In order to attain metrics for your scheduling algorithm you need to have invoked and completed the simulation, this is done by calling the Run method on a valid Runner object.

Figure 3: Invoking a simulation

```
using Cpuss;
using Cpuss.Strategies;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            // ...
            ProcessLoad processLoad = new ProcessLoad();
            Runner runner = new Runner(processLoad, new ShortestJobFirstExpert(10,
20));
            runner.Run();
            // ...
        }
    }
```

```
}
```

2.4 Data associated with each process

The following is a brief list of properties exposed by the `Process` type.

- Arrival Time
- Start Time
- Completion Time
- CPU Activity
- Id
- Priority
- Wait Time
- Response Time
- Turnaround Time

All of this data is valid after the simulation has completed.

Figure 4: Inspecting some properties of a process

```
using System;
using System.Collections.Generic;
using Cpuss;
using Cpuss.Strategies;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            Runner runner = new Runner(10, 10, 40, new ShortestJobFirstExpert(10,
20));
            runner.Run();
            foreach (KeyValuePair<int, Process> process in runner.Metrics)
            {
                Console.WriteLine("PID: {0}", process.Value.Id);
                Console.WriteLine("Wait Time: {0}ns", process.Value.WaitTime);
            }
        }
    }
}
```

2.5 Hooking into key process lifetime events

There may be times when you want to see what happens to what process and when during the simulation, or record the data for further analysis afterwards, e.g. you may want to keep note of the number of context switches that your algorithm has coerced from its processes. CPUSS offers four events, specific to processes these include:

- ProcessStarted
- ProcessPreempted

- ProcessResumed
- ProcessCompleted

These events are all exposed by the Runner which keeps track of key moments of a Process' lifetime.

Figure 5: Key process lifetime events

```
using System;
using Cpuss;
using Cpuss.Strategies;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            Runner runner = new Runner(10, 2, 13, new ShortestJobFirstExpert(1,
25));
            runner.ProcessStarted += (o, e) => Console.WriteLine("PID{0} Started",
e.Id);
            runner.ProcessPreempted += (o, e) => Console.WriteLine("PID{0}
Preempted", e.Id);
            runner.ProcessResumed += (o, e) => Console.WriteLine("PID{0} Resumed",
e.Id);
            runner.ProcessCompleted += (o, e) => Console.WriteLine("PID{0}
Completed", e.Id);
            runner.Run();
        }
    }
}
```

Events are specific to the CPUSS framework in that they are exposed as hooks for extracting further custom data for further analysis, and also as a means to provide hooks for other tooling that you may require in your research work.

3 CPUSS Report Generator (CPUSSRG)

CPUSSRG allows you to perform a load test scenario using a specified strategy. The test can be repeated *n* times.

The strategies available to use with CPUSSRG are located in the `Plugins` subfolder. All valid algorithms are those that implement the `IStrategy` interface as described before. In order to use CPUSSRG with your strategies all you need to do is place the assembly that defines your algorithms into the `Plugins` folder and they will be loaded at runtime and available for selection.

3.1 Valid Flags

Below is a list of flags, and the expected argument types each flag expects.

Table 1: CPUSSRG flags

Flag	Argument
/strategy	Strategy Name<Type>,arg1<Int32>,arg2<Int32>,...,argN<Int32>
/report	String<name of file>
/small	Int32<number of small processes>
/medium	Int32<number of medium processes>

```
/large          Int32<number of large processes>
/repeat         Int32<number of times to repeat simulation>
/outdir         String<name of output directory>
/window         Int32<window timeframe to use for throughput>
```

Figure 6: Example CPUSSRG usage

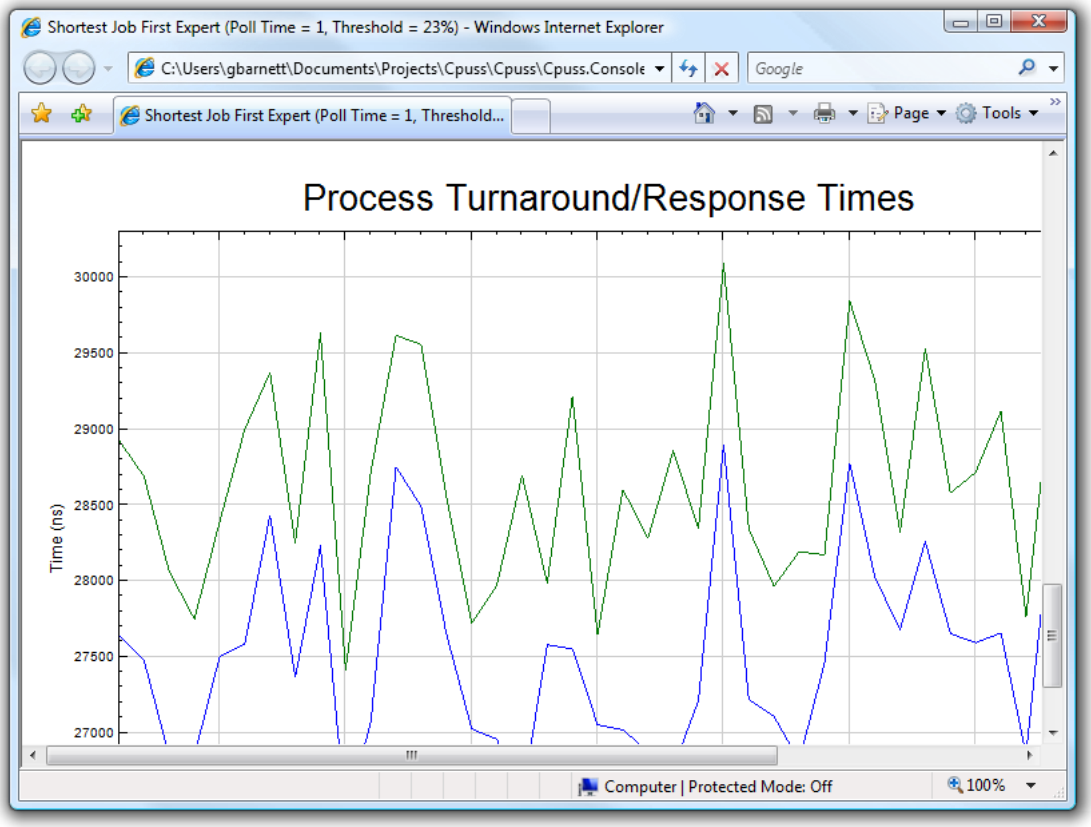
```
> cpussrg /strategy:ShortestJobFirstExpert,1,25 /report:sjfe /repeat:25 /small:33
/large:15
```

Figure 6 shows an example where the strategy specified requires two parameters for its constructor, both of which are `Int32`'s (`Int32`'s are the only type supported from CPUSSRG, after research I have found that no strategies require a more complex integer data type). The parameter values are associated with its respective constructor parameters in a left-to-right manner, that is if the `ShortestJobFirstExpert` constructor had two parameters, the first of which was called `pollTime`, the second `threshold` then the value of `pollTime` would be `1`, and the value of `threshold` would be `25` respectively. Strategies with default, parameter less constructors need not bother with the additional syntax; all that is required is the short type name of the strategy.

3.2 Generated Reports

CPUSSRG presents the data collected from the simulation runs as a HTML file containing tabular data and visualizations of that data in the form of graphs.

Figure 7: Report



CPUSSRG is not the only tool that can create reports; if you are using the CPUSS framework directly you can create a report by using the `HtmlReport` type.