Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

SF3P: A Scheduling Framework For Fast Prototyping

A Programmer's Manual

Version 0.1 α

Andrés Gómez

gomeza@tik.ee.ethz.ch

http://www.tik.ee.ethz.ch/~euretile/scheduling/

Preface

The Scheduling Framework For Fast Prototyping (SF3P) is a scheduling framework that runs at the applications level, on top of a standard linux kernel. This project began as a part of a master thesis at ETH. The original framework of the thesis, called Extendable Server Framework (ESF), ran only two algorithms (Time Division Multiple Access and Constant Bandwidth Server) in flat topologies. SF3P was designed by taking ESF's basic principles and building a model that would work with a variety of scheduler and server algorithms. One of the new features in SF3P is the ability to hierarchically combine different scheduling algorithms.

In addition to the basic framework, there were several (command line) tools developed in order to facilitate the execution of the framework. These include an XML parser, Octave scripts to calculate different metrics, and some bash scripts to perform batch experimentation.

This manual is attempt to help developers familiar with C/C++ to use, and (hopefully) extend SF3P to their own scheduling needs.

Contents

1

Introduction

1.1 Overview

The Scheduling Framework For Fast Prototyping (SF3P) is an application-level framework that gives the developer the freedom to choose the scheduling algorithm that will run arbitrary task sets.

In general terms, all multitasking systems require a scheduler to determine what task to run whenever there is more than one active task. All tasks that arrive must be stored in a queue. The scheduler is in charge of selecting exactly one task (in a uniprocessor environment) from this queue. This task will then execute, and depending on whether the scheduling algorithm is preemptable or not, the task will continue executing until its completion. At this point, the scheduler will choose the next task to run. This process is summarized in in figure 1.1.

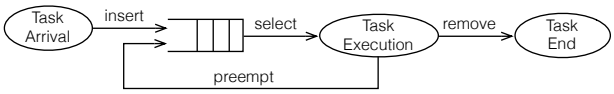

Figure 1.1: Simplified scheduling model for uniprocessor environments.

In a typical linux system, one can have several tasks which sit on top of a concurrency manager like the pthread library. Each task, or thread, can have a scheduling parameter associated to it, most commonly a priority. The concurrency manager serves as a middleman between the kernel and the tasks and ensures that the tasks' scheduling parameters are enforced. The linux kernel, shown in figure 1.2 in gray, maintains the queues of all the idle and blocked processes in the entire system. It is also in charge of handling interrupts, preempting active threads, context switching, and several other lower level functions.

SF3P introduces a new layer at the application level that implements different scheduling algorithms to be used by SF3P tasks running on top of it. This would not be possible in a standard linux system because the linux scheduler is priority-based, and has limited scheduling options. Naturally, since SF3P itself runs on top of the standard pthread library, all algorithms must be implemented using a priority-based mechanism, which will be explained in greater detail in section 2.1.

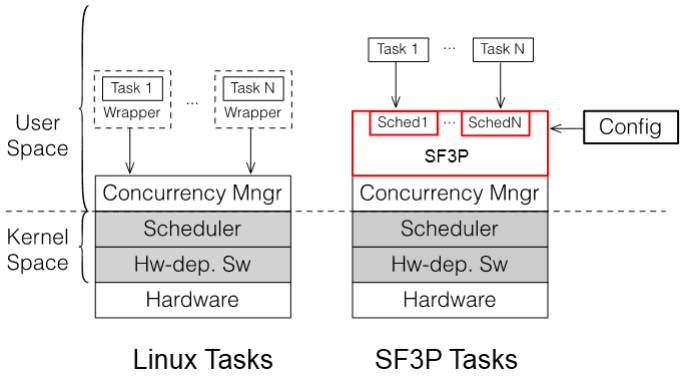

Figure 1.2: Comparison between standard linux and SF3P task sets.

1.2 Features

SF3P was designed with several features in mind. By introducing a scheduling layer at the application level, we can use standard software components such as a concurrency manager and a linux kernel. This design method ensures that SF3P has high hardware and software compatibility. The linux kernel can offer high resolution timers, so the timing accuracy in SF3P will also be high.

The main contribution of SF3P is the ability to combine classical schedulers in a hierarchical fashion. This can be achieved because of a modular, object-oriented design. In addition, the design reuses many components, with lowers the implementation costs of new scheduling algorithms.

The standard linux scheduler offers 100 priority levels. In order to execute a priority higher than 1, root privileges are needed. Since most of the linux processes run with a lower priority when SF3P executes (with a higher priority), it takes control of one cpu and fully utilizes it until the end of the execution.

Summary of features:

- Portable
- Extendable
- High HW/SW compatibility
- High timing accuracy
- Low overhead
- Hierarchical scheduling

1.3 System Requirements

To Compile & Execute:

- POSIX-compliant kernel
- (High Resolution) Timers
- Pthread library
- glibc++
- libmgl7-dev
- libx11-dev
- octave
- php
- Root privileges

1.4 Installation

1. If you SF3P folder is not located in ∼/git, then please change line 3 of sf3p_paths.sh to the path of your SF3P folder

2. In the terminal, type:

   source sf3p_paths.sh

   This will set a new $SF3P variable, and add it to your $PATH variable. You can also add it to your ∼/.bashrc file, to have it load automatically. This will allow you to easily invoke any SF3P executable from any folder in the terminal.

   You need root privileges to execute sf3p. On some older systems, you might have to add the following line to your bash profile in order to inherit you PATH variable when using 'sudo':

   alias sudo='sudo env PATH=$PATH@'

3. Type:

   ./install.sh

   By this point, if all of the library requirements are met, you have successfully compiled SF3P and you may execute it from any terminal.

2

SF3P Basic Concepts

2.1 Basic scheduling mechanism

The scheduling of tasks in SF3P is done by a `scheduler` thread. This thread will always have a high priority. All tasks are performed by a `worker` thread. This thread can have either an active or an inactive priority. When active, a `worker` can execute its task; when inactive, it must wait. At any point time, there can be at most one `worker` with an active priority. It is the `scheduler` thread's job to activate and deactivate each `worker` thread it *schedules* to run.

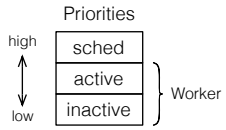

Figure 2.1: Scheduling Concept: `scheduler` thread will always have a high priority. `worker` threads can have an active priority, where they can execute their task, or an inactive priority, where they have to wait to be activated.

2.2 Components

SF3P has three components mapped to pthreads: the scheduler, the worker, and the dispatcher. The first two were introduced in the previous section. The `dispatcher` thread is in charge of simulating the job arrivals belonging to one task. SF3P supports aperiodic, periodic, and periodic tasks with jitter.

The SF3P executable accepts an XML file as input where the topology, including the scheduler and task set, is described. The SF3P parser will automatically generate

all the different pthreads according to the specifications. During the simulation, traces of different events will be buffered in memory, and after the simulation ends, they will be saved to different csv files. The component diagram can be seen in figure 2.2.

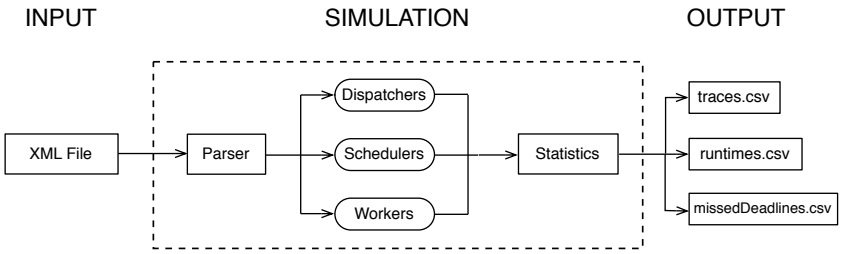

Figure 2.2: Component View. The SF3P executable has an XML file as input, and has several csv files as output.

2.3 Input File

The input file must be an XML that follows the format shown in listing 2.1.

```
<simulation name="STRING">
    <duration TIME />
    <TOPOLOGY>
</simulation>
```

Listing 2.1: Input xml file format

Where STRING can be any alphanumeric string. For the simulation name, it is recommended to choose the filename as the simulation name.

TIME is a combination of two xml attributes, **value** and **units**, that are added to certain xml nodes like: duration, period, jitter, relative_deadline, etc. **Value** must be a positive integer, and **units** can be "us", "ms", or "sec", for microseconds, milliseconds and seconds, respectively. So, for example, a 1 second simulation requires the following *duration* xml node:

<duration value="1" unit="sec" />

The TOPOLOGY node consists of at least two items: a top-level scheduler and at least one worker. Schedulers and workers are declared as *runnables* and some have specific subitems. A basic TOPOLOGY can be seen in listing 2.2.

```
<runnable type="scheduler" algorithm="ALG">
      <WORKER>
</runnable>
```

Listing 2.2: Basic TOPOLOGY

The WORKER node describes the task to be assigned to one `worker` thread, as well as its periodicity. Listing 2.3 shows how to declare a basic worker.

```
<runnable type="worker" periodicity="PER" task="TASK">
      <period TIME />
      <relative_deadline TIME />
      <CRITERIA>
</runnable>
```

Listing 2.3: Basic WORKER

Where PER is either "periodic", "aperiodic", or "periodic_jitter". When the `worker` is periodic, it must contain a *period* subitem. If it is aperiodic, it must contain a *release_time* subitem. And if it is periodic with jitter, it must contain both a *period* and *jitter* subitem.

The CRITERIA node describes some additional scheduling parameters associated to one `worker` . If, for example, a fixed-priority scheduler is used, one must specify the priority of each worker. The xml node describing one such sample TOPOLOGY with two tasks can be seen in listing 2.4.

```
<runnable type="scheduler" algorithm="FixedPriority">
      <runnable type="worker" periodicity="periodic" task="TASK">
            <period value="10" units="ms />
            <criteria type="inclusive">
                  <priority value="10" />
            </criteria>
      </runnable>
      <runnable type="worker" periodicity="periodic" task="TASK">
            <period value="5" units="ms />
            <criteria type="inclusive">
                  <priority value="5" />
            </criteria>
      </runnable>
</runnable>
```

Listing 2.4: A simple fixed-priority topology

Currently, SF3P support only two types of tasks: "busy_wait", and "video". The first is a simple busy-wait loop, which requires a *wcet* subitem. The second is a mjpeg video decoder, where one job decodes one video frame.

2.4 Output Files

traces.csv

This file contains traces from all SF3P threads. Table 2.4 describes all events traced and their corresponding thread. Listing 2.5 shows the trace row format. The timestamp is the number of microseconds since the beginning of the simulation.

```
<timestamp> , <thread\_id> , <event>
```

.

Listing 2.5: traces.csv row format

Event	Description	Registered by
TASK_ARRIVAL	The arrival of a new job arrival	`dispatcher`
SCHED_START	The `scheduler` has activated a `worker`	`scheduler`
SCHED_END	The `scheduler` has deactivated a `worker`	`scheduler`
TASK_START	The `worker` has begun the execution of a job	`worker`
TASK_END	The `worker` has finished the execution of a job	`worker`
DEADLINE_MET	The `worker` met the deadline of its last job	`worker`
DEADLINE_MISSED	The `worker` missed the deadline of its last job	`worker`

Table 2.1: Events traced by SF3P.

runtimes.csv

This file contains the runtimes of all SF3P threads. Listing 2.6 shows the runtime row format. The runtime is the total number of microseconds that the thread ran for. Thread type can be one of the following: Idle, Scheduler, Dispatcher, or Worker.

```
<thread_type> , <thread\_id> , <runtime>
```

.

Listing 2.6: runtimes.csv row format

missedDeadlines.csv

This file lists any and all missed deadlines during the simulation. Listing 2.7 shows the row format. All times (including deadline) are in microseconds relative to the beginning of the simulation.

```
<thread_id> , <start_time> , <deadline>, <finish_time>
.
```

Listing 2.7: missedDeadlines.csv row format

2.5 Analysis

SF3P includes some tools to analyze the results from the framework execution. Octave is used to calculate the metrics described in table 2.5.

Metric	Description
Execution Time	The amount of time spent by the `worker` in order to complete a job
Response TIme	FInish Time - Arrival Time
Utilization	$\sum$Execution Times / Simulation Time
Resource Allocation Cost	The execution time of all schedulers
Total System Cost	Resource Allocation Cost + time spent simulating all job arrivals
Worker Cost	Ratio between a worker's traced runtime and the pthread_clock runtime
Throughput	Number of jobs completed in simulation / simulation time
MIssed Deadlines	Percentage of missed deadlines per `worker`

Table 2.2: Metrics calculated by SF3P tools.

3

SF3P Source Code

3.1 File Structure

The SF3P project is organized into 7 main folders, explained in the following table:

Folder	Content
bin	All executable binaries
doc	Doxygen generatd documentation files
examples	Sample xml simulation files
lib	C++ libraries such as the mjpeg decoder, and mathgl dependent code
obj	Non-executable object files for all SF3P tools and libraries
scripts	Auxiliary scripts in php, bash, matlab, and octave
src	SF3P source code

Table 3.1: Main SF3P folders.

The SRC folder

The src folder contains the main SF3P code developed in C++. It contains all of the C++ main functions for all SF3P executables. It is also divided into 10 subfolders that group relevant SF3P classes together. They are explained in table 3.1.

Each folder has an associated makefile target for compilation. When modifying files belonging to a single folder, one can simply recompile the folder, and relink using the other preexisting object files; this great reduces the total compilation time.

Subfolder	Content
core	High-level abstract classes and the Simulation class
criteria	Criteria subclasses. For the moment, only InclusiveCriteria exists
dispatchers	All `dispatcher` subclasses (Periodic, Aperiodic, etc)
pthread	Pthread dependent classes.
queues	RunnableQueue subclasses (PriorityQueue, DeadlineQueue, etc)
results	Classes that gather statistics and produce the output files
schedulers	Scheduler classes
servers	Server classes (under development)
tasks	Task subclasses
until	Time-keeping functions, enumerations, and other utilities

Table 3.2: SRC subfolders.

3.2 MakeFile

SF3P includes several different executable files that not only run the framework, but also perform some analysis and facilitate the use of the framework. All of the SF3P executables are explained in table 3.2.

SF3P Executables

Executable	Description
sf3p	SF3P framework executable that performs one simulation and produces the output csv files
calculate	SF3P tool that serves as a wrapper to call octave scripts which calculate metrics and saves them to additional csv files
show	SF3P tool that prints the csv metrics files with some statistics in the command line
simulate	SF3P executable that automatically runs the sf3p executable and calculate tool
simfig	SF3P tool that generates a graphical representation of the simulation
publish	SF3P tool that generates an HTML file with all of the simulation statistics

Table 3.3: SF3P Executables.

The main makefile targets used to compile all SF3P executables are explained in table 3.2.

Target	Description
all	Compiles all libraries and sf3p classes. Generates all SF3P executables
sf3p	Compiles only sf3p classes and relinks it with existing library object files
tools	Compiles only the SF3P tools simulate, calculate, show, publish, and simfig
clean	erases all binary files (executable and object files).

Table 3.4: Makefile Targets.

3.3 Compiling SF3P

In the terminal, when standing in the $SF3P folder, you must type:

make TARGET

Where TARGET is one of the targets mentioned in table 3.2.

4

The SF3P Executables

This chapter shows how to run all of the SF3P executables.

4.1 SF3P

SF3P is the main executable that receives the XML file as input, and generates CSV files as output.

To run SF3P, type in your terminal:

sudo sf3p <FILE>

Where <FILE> is the name of XML file in your current directory. You can omit the .xml and just use the file's prefix. When you run the edf example provided in the SF3P folder, you obtain the following:

```
$ ls
edf.xml
$ sudo sf3p edf

***    Loaded 'edf'       ***
***    Simulating         ***
***    Done               ***
***    Results Saved!     ***

$ ls
edf_missed_deadlines.csv  edf_runtimes.csv  edf_traces.csv  edf.xml
```

As the messages indicate, the file is first parsed and loaded, then it simulates and saves the results. After running SF3P, you obtain the three output files mentioned in section 2.4.

4.2 CALCULATE

At this point, with the tree output files, you can calculate any of the metrics mentioned in table 2.5. In the command line, there is a shorthand for each metric, as shown in the following table:

Metric	Shorthand
Execution Time	exe
Response TIme	reap
Utilization	util
Resource Allocation Cost	alloc
Total System Cost	sys
Worker Cost	worker
Throughput	throughput
MIssed Deadlines	missed
All metrics	all

Table 4.1: Metrics calculated by SF3P tools.

You can run the calculate tool with the following format:

calculate <metric_shorthand> ∗ <file_prefix>

Where you can indicate (in any order) the file prefix (simulation name), as well as one or more metrics that you wish to calculate. If no metric is provided, all metrics are calculated.

Following our example, this is how we would calculate the response times:

```
$ ls
edf_missed_deadlines.csv  edf_runtimes.csv  edf_traces.csv  edf.xml
$ calculate edf resp

+++  Simulation: edf  +++

Calculating Response Times...

$ ls
edf_missed_deadlines.csv  edf_resp_ms.csv  edf_runtimes.csv
edf_traces.csv  edf.xml
```

The calculate tool does not show the results, it merely calculates them, and saves them in different csv files. In our example, the new file is edf_resp_ms.csv (numbers are in milliseconds).

4.3 SHOW

The show tool reads the csv file generated by the calculate tool and prints some statistics in the command line, using the same syntax as the calculate tool. You must first calculate a metric before running the show command.

Following our example, this is how we would show the response times of the EDF simulation:

```
$ ls
edf_missed_deadlines.csv  edf_resp_ms.csv  edf_runtimes.csv
edf_traces.csv  edf.xml
$ show edf resp

+++   Simulation: edf   +++

  Response Times (ms):

        ID      N_Jobs         MIN         AVG         MAX       TOTAL
         1          14      20.219      20.247      20.326     283.452
         2          14      40.408      40.442      40.501     566.181
         3          14      60.601      60.645      60.712     849.037
```

Note that your numbers might be a little different, but they should be close to 20, 40, and 60.

4.4 SIMFIG

The simfig tool generates a figure representing the execution of tasks. You can run the simfig tool with the following format:

simfig <file_prefix>

Following our example, this is how we would generate a figure for our EDF simulation:

```
$ ls
edf_missed_deadlines.csv  edf_resp_ms.csv  edf_runtimes.csv
edf_traces.csv  edf.xml
$ simfig edf
```

```
***    Saved EPS Figure          ***
***    Saved SVG Figure          ***

$ ls
edf_figure.eps   edf_missed_deadlines.csv   edf_runtimes.csv   edf.xml
edf_figure.svg   edf_resp_ms.csv            edf_traces.csv
```

Note that two figures are generated, one in EPS format, and another in SVG format. When you open the figure, you will obtain something like this:

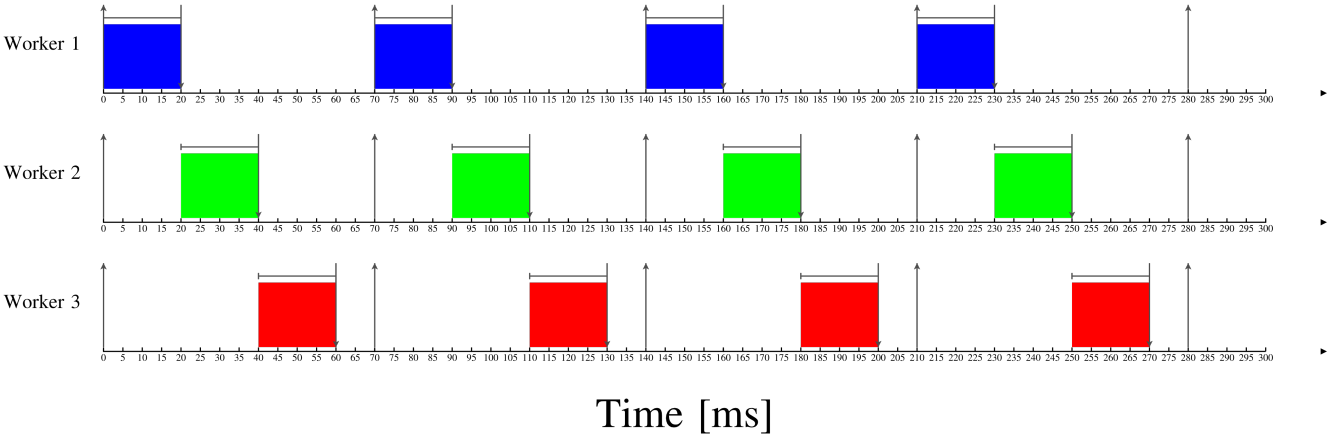

Figure 4.1: Output of the simfig tool

4.5 PUBLISH

The publish tool is meant to be used when all metrics need to be calculated and it intends to facilitate the review of all metrics for large simulations. It hides all auxiliary CSV files in a folder and generates an HTML page with the simulation results.

You can run the publish tool with the same syntax as the simfig tool. Following our example, this is how you can publish the results of the EDF simulation:

```
$ calculate all edf

+++    Simulation: edf    +++

***    Calculating all metrics!    ***

Calculating Execution Times...
```

```
Calculating Response Times...

Calculating Throughput...

Calculating Utilization...

Calculating Resource Allocation Cost...

Calculating System Cost...

Calculating Worker Cost...

Calculating Missed Deadlines...

$ ls
edf_alloc_cost_us.csv  edf_figure.svg            edf_runtimes.csv
edf_traces.csv       edf.xml
edf_exec_ms.csv        edf_missed_deadlines.csv  edf_sys_cost_us.csv
edf_utilization.csv
edf_figure.eps         edf_resp_ms.csv           edf_throughput.csv
edf_workerCost.csv

$ publish edf

***   Published: edf.hmtl       ***

$ ls
edf_data  edf.html  edf.xml
```

Note that all of the auxiliary files are stored in the folder "edf_data". You can now open the edf.html file to explore the results using an easy to use web interface.