P1 - Scheduler

200007413

1 Design

I tried to focus my practical on clarity of code to help as I build up the functionality from basic FIFO implementation, to priority scheduling with a time-quantum based ageing function. I began by outlining the general functionality the sched program would require.

```
1. parse config file
  1.1. foreach line in config -> create process control block
    1.1.1. parse the process information from the config line
  1.2. admit the process control blocks to the schedule
    1.2.1. run the process as a child
    1.2.2. stop the process
    1.2.3. add the process to the schedule
2. while schedule is not empty -> run cycle
  2.1. run the next process
  2.2. wait for the process to either terminate or stop
  2.3. update the schedule
    2.3.a. if the process terminates, remove it from the schedule
    2.3.b. otherwise, re-add the process to the schedule
```

1.1 Data Stuctures

I focused on creating two data structures which would be needed to manage the scheduler and it's processes:

- Scheduler (*Sched*)

  Contains an ordered list of process control blocks which are created by parsing the config.

- Process Control Blocks (*Process*)

  Contains information about a particular line in the config file allowing a child process to be created and then be maintained by the scheduler.

For the list of process control blocks *Sched* holds, I decided to use a **doubly-linked list**. My motivation for this decision was two fold:

Firstly, I looked at a typical CPU scheduler, and noted that schedulers typically are dynamically sized to allow processes to be added and removed at any time. The specification also required the scheduler to be able to setup an arbitrary number of process in a configuration file.

Secondly, I had always expected to eventually implement priority scheduling, which involves maintaining an ordered list when elements are added, as well as when priorities change due to ageing.

These requirements make a standard array less feasible, as they must be created with a strict maximum length, or require expensive realloc commands to grow and shrink. Also, using an array would require shifting lots of elements when processes are removed and added, especially when priority scheduling is used, as elements are not just added to the back of the list like with a basic FIFO scheduler.

Therefore, I chose a doubly-linked list, as they have no theoretical maximum size (except for the limitation of integer's size in C) because elements can always be added by updating pointers of elements. Moreover, adding elements to positions within the list requires only updating pointers of the new element and surrounding elements, which improves the performance of the priority scheduler over shifting many elements in an array.

To make it easier for the initial first-in-first-out (FIFO) scheduler, I chose to include in the scheduler both a link to the head of the list, and the tail. This is because elements can be popped from the head of the list, and for the FIFO scheduler, just pushed to the tail.

To keep track of the list, I addded head and tail attributes to the Sched struct. Whilst the tail attribute is not necessary, I decided it would make the initial FIFO scheduler more efficient, as elements are pushed to the back of the list, and popped from the front, and just having a head pointer would require looping through each element in the list to get to the end. However, for the priority scheduler this cannot be avoided, as elements must be placed in positions based on their priority.

With these decisions made, my *Sched* structure had the following attributes:

Attribute	Data Type
head	Process*
tail	Process*
size	int

Next I designed the linked list elements. As this data structure only needed to hold pointers to the next and previous elements, as well as the process control block itself, I just included the next and prev pointers in teh *Process* struct itself.

In reflection, I feel this decision did not best follow my primary design aim for clarity. Separating the scheduler list items from the process control blocks would have kept the scheduler functionality completely separate from the process information.

However, after making this decision, I designed the rest of the *Process* structure

As the scheduler would be managing the processes, I needed a way of keeping track of each processess state, along with the informatin about the process. Therefore, I designed an enum data structure for the process state called PROCESS_STATE, which would contain the NEW, READY, RUNNING, and TERMINATED states the process could be in at any particular time. I also added a pid attribute which would be set when the child process is created using the config information.

Attribute	Data Type
next	Process*
prev	Process*
state	PROCESS_STATE
pid	int
runCount	int
priority	int
path	char*
args	char**

For the priority scheduler, I made the decision to age processes based upon the number of run cycles ran on a particular process. Each time a run cycle runs a particular process, it's runCount increments, and when this reaches a threshold set by CYCLES_BEFORE_AGEING is reached, the process' priority decrements.

This decision to decrement the priority of frequently ran processes instead of incrementing infrequently ran processes because every cycle, only the process ran would require ageing, instead of all processes in the schedule.

However, I now realize a flaw in this decision comes if processes can be added to the schedule at the same time it is running (like in the case of standard CPU schedulers). Here, low priority tasks remain low-priority, but the newly added higher-priority tasks will always be chosen before the low-priority tasks, not sufficiently tackling the problem of starvation. The high-priority tasks eventually become low-priority, and so also starve if higher-priority tasks are constantly added.

1.2 Code Structure

In writing the methods in each file, I tried to follow a linear layout, with the functions called defined directly below the functions which call them in order, to help with code readability. For example, the parser.c file begins by defining the parseSchedulerConfig method, which then calls the parseProcessConfig method with each line in the config file, which is defined directly below.

The basic spec did not require a round-robin or priority scheduler, but I wanted to extend my code to include these variations.

With this in mind, and with the desire to unit-test my code, I split my code into files corresponding to the general functionality or data structure they were associated with. I created header files for the data structure related files which contained the data structure themselves, and prototypes for the functionality associated with them.

In the case of the sched data structure, this allowed me to have the basic sched.h header file, the universal scheduler functionality in the sched.c file, and the scheduer-specific code in their own sched_fifo.c, sched_rr.c, and sched_priority.c files, which all implement the same methods which are not included in the universal sched.c file.

This meant switching between the chedulers was easy for testing and performance comparisons, as I could just switch the implementation specific import.

To determine the methods requiring designing, I looked back at my outline, and broke it down into single-function tasks, which could be implemented.

```
1 parseSchedulerConfig(config_file)
  1.1 Sched_create()
  1.2 parseProcessConfig(config_file)
      1.2.1 Process_create(priority, path, args)
  1.3 Sched_admit(sched, process)
      1.3.1 Process_setup(process)
      1.3.2 Sched_push(sched, process)
2 Sched_run(sched)
  2.1 while !Sched_isEmpty(sched) -> Sched_cycle(process)
      2.1.1 Sched_pop(sched)
      2.1.2 Process_run(process)
      2.1.3 Process_wait(process)
      2.1.4 if priority scheduler -> Process_age(process)
      2.1.4 if Process_getState(process) != TERMINATED -> Sched_push(process)
```

1.3 Additional Features

As well as the required first-in first-out scheduler, I designed and implemented a round-robin scheduler, and then a priority scheduler, so I could compare the performance differences between them, and understand how schedulers can be improved.

My round-robin design set a time-quantum, which would limit the amount of time the scheduler will wait for a particular process (in milliseconds). This prevents processes which are particularly intesive from blocking the other processes entirely, which enable the less intensive processes to complete earlier, which reduces the total wait time of the scheduler.

My design of this round-robin approach forks a child process which waits for the time-quantum, and then sends a stop signal to the scheduled process. As the main scheduler process waits for the scheduled processs to change state, the waiting then can end when either the process completes, or the time-quantum ellpases. For larger time-quantum, this notably increases performance, as it removes the unnecessary wait time for processes after they complete

As the scheduler checks whether the process has completed or was terminated, this approach intercepts signals sent from anywhere, which allows the scheduler to interact with other programs which may stop the scheduled, without breaking.

2 Implementation & Testing

As I implemented features, I created unit tests to help ensure functionality remained consistent as I extended the code from FIFO scheduling, to RR scheduling, to priority scheduling. My code structure made this process relatively simple, as the scheduler specific tests could include the relevant implementation.

For these tests, I created a simple unit-testing application, where tests are defined as functions, which are then ran and check if expressions are true to determine the success of the test.

Throughout creating methods which took parameters, I made sure to always add the edge cases of NULL parameters, as well as tests in the methods themselves which would print sensible error or warning messages to stderr.

As the schedule would hold a list of Processes, I first implemented the Process struct and it's related methods which would be used by the scheduler.

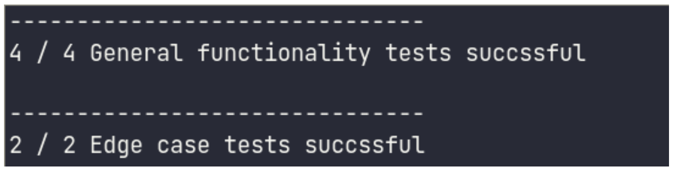
As I implemented this, I included prototypes in the process.h header file containing the process data structure, and created a test_process.c file for unit-testing.

I used malloc to create both the Sched and Process variables, as they are used throughout the whole execution of the scheduler, and so need to remain accessible.

In my testing of implementing the Process_create method, my testing showed that I could not just set the process path argument to the variable passed into the function. This is because it may be the case that the pointer would be to the argument itself, and so all created processes would point to the same string, meaning every path would be set to the last set value.

I fixed this by using the *strdup* method from the standard library string.h. This makes a unique copy in memory of the string so each process has it's own path variable.

The results of running the process tests on my final submission are below.

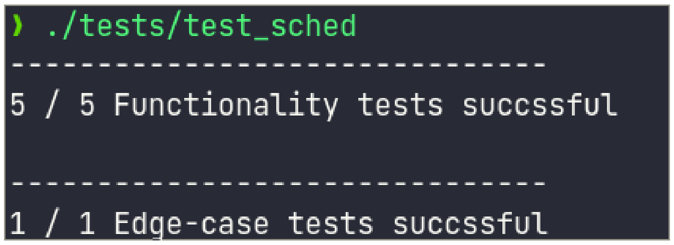


```
------------------------------
4 / 4 General functionality tests succssful

------------------------------
2 / 2 Edge case tests succssful
```

The rest of the implementation was relatively straight forwards, and after creating tests, I moved on to create the schedule.

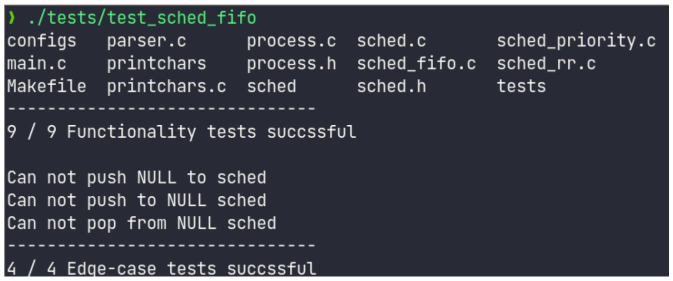
For this, I first created and tested the basic functionality in the sched.c file, before tackling the implementation specific functionality.

Test results for the basic scheduler functionality:



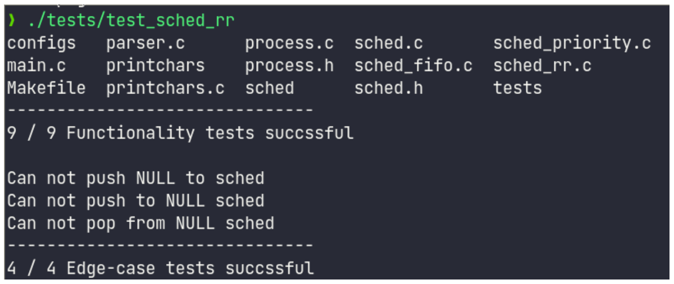
```
❯ ./tests/test_sched
------------------------------
5 / 5 Functionality tests succssful

------------------------------
1 / 1 Edge-case tests succssful
```

Test results for the FIFO scheduler implementation:



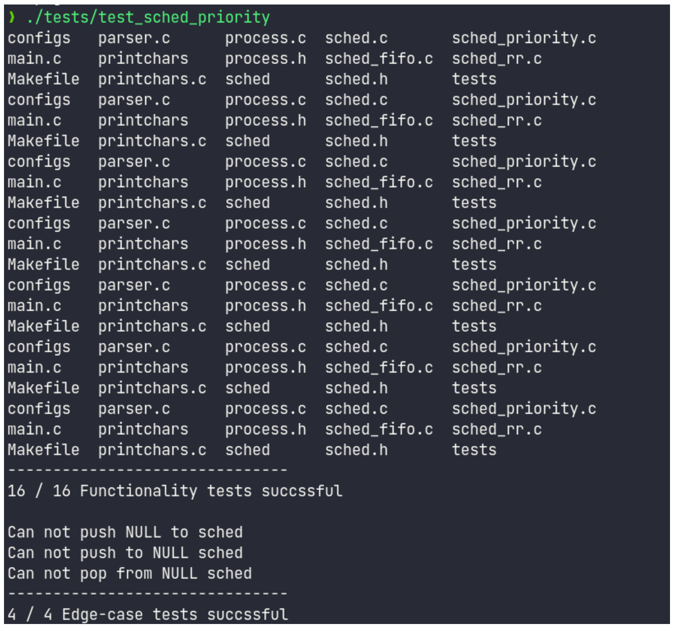
```
❯ ./tests/test_sched_fifo
configs   parser.c      process.c  sched.c       sched_priority.c
main.c    printchars    process.h  sched_fifo.c  sched_rr.c
Makefile  printchars.c  sched      sched.h       tests
------------------------------
9 / 9 Functionality tests succssful

Can not push NULL to sched
Can not push to NULL sched
Can not pop from NULL sched
------------------------------
4 / 4 Edge-case tests succssful
```

Tests for the round-robin scheduler implementation:



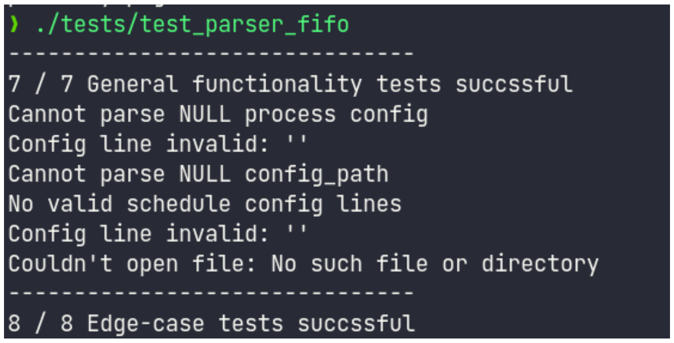
```
❯ ./tests/test_sched_rr
configs   parser.c      process.c  sched.c       sched_priority.c
main.c    printchars    process.h  sched_fifo.c  sched_rr.c
Makefile  printchars.c  sched      sched.h       tests
------------------------------
9 / 9 Functionality tests succssful

Can not push NULL to sched
Can not push to NULL sched
Can not pop from NULL sched
------------------------------
4 / 4 Edge-case tests succssful
```

Tests for the priority scheduler implementation:



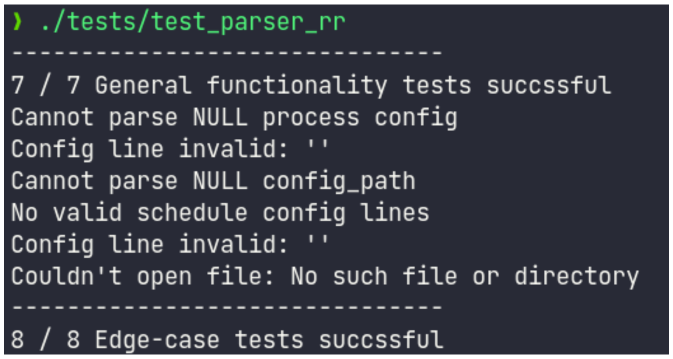
```
❯ ./tests/test_sched_priority
configs   parser.c      process.c  sched.c       sched_priority.c
main.c    printchars    process.h  sched_fifo.c  sched_rr.c
Makefile  printchars.c  sched      sched.h       tests
configs   parser.c      process.c  sched.c       sched_priority.c
main.c    printchars    process.h  sched_fifo.c  sched_rr.c
Makefile  printchars.c  sched      sched.h       tests
configs   parser.c      process.c  sched.c       sched_priority.c
main.c    printchars    process.h  sched_fifo.c  sched_rr.c
Makefile  printchars.c  sched      sched.h       tests
configs   parser.c      process.c  sched.c       sched_priority.c
main.c    printchars    process.h  sched_fifo.c  sched_rr.c
Makefile  printchars.c  sched      sched.h       tests
configs   parser.c      process.c  sched.c       sched_priority.c
main.c    printchars    process.h  sched_fifo.c  sched_rr.c
Makefile  printchars.c  sched      sched.h       tests
configs   parser.c      process.c  sched.c       sched_priority.c
main.c    printchars    process.h  sched_fifo.c  sched_rr.c
Makefile  printchars.c  sched      sched.h       tests
configs   parser.c      process.c  sched.c       sched_priority.c
main.c    printchars    process.h  sched_fifo.c  sched_rr.c
Makefile  printchars.c  sched      sched.h       tests
------------------------------
16 / 16 Functionality tests succssful

Can not push NULL to sched
Can not push to NULL sched
Can not pop from NULL sched
------------------------------
4 / 4 Edge-case tests succssful
```

Test results for the parser using the FIFO scheduler implementation:



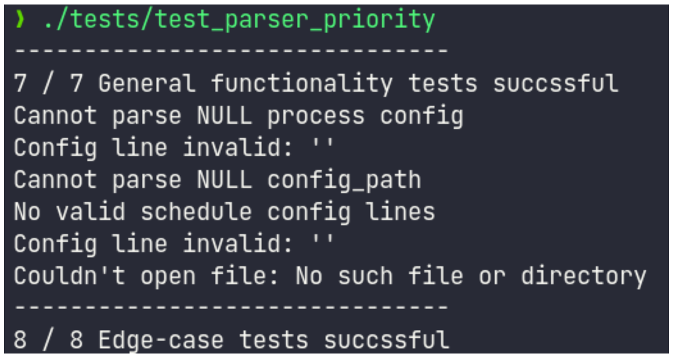
```
❯ ./tests/test_parser_fifo
------------------------------
7 / 7 General functionality tests succssful
Cannot parse NULL process config
Config line invalid: ''
Cannot parse NULL config_path
No valid schedule config lines
Config line invalid: ''
Couldn't open file: No such file or directory
------------------------------
8 / 8 Edge-case tests succssful
```

Test results for the parser using the round-robin scheduler implementation:



```
❯ ./tests/test_parser_rr
------------------------------
7 / 7 General functionality tests succssful
Cannot parse NULL process config
Config line invalid: ''
Cannot parse NULL config_path
No valid schedule config lines
Config line invalid: ''
Couldn't open file: No such file or directory
------------------------------
8 / 8 Edge-case tests succssful
```

Test results for the parser using the priority scheduler implementation:

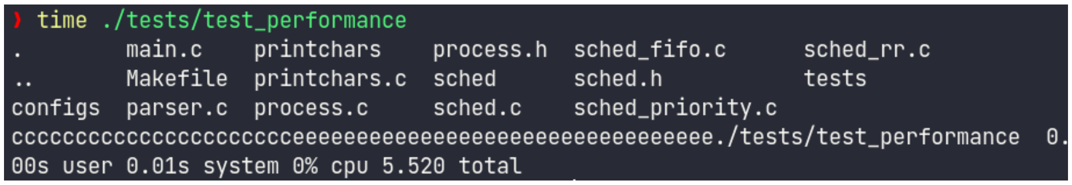


```
❯ ./tests/test_parser_priority
------------------------------
7 / 7 General functionality tests succssful
Cannot parse NULL process config
Config line invalid: ''
Cannot parse NULL config_path
No valid schedule config lines
Config line invalid: ''
Couldn't open file: No such file or directory
------------------------------
8 / 8 Edge-case tests succssful
```

2.1 Performance

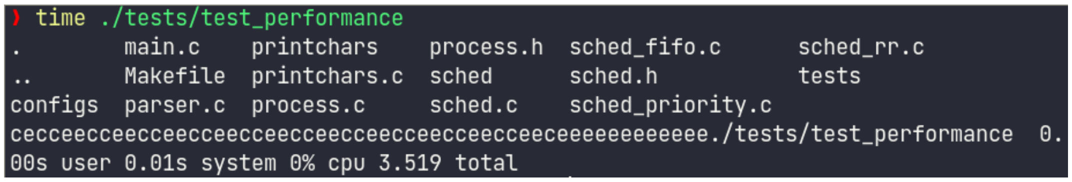
I ran a few different example configuration files on each of the three scheduling types that I implemented, and timed the total time it took to complete using the UNIX *time* command, and re-ran them, including each of the different scheduler implementations for comparion.

- FIFO Scheduler



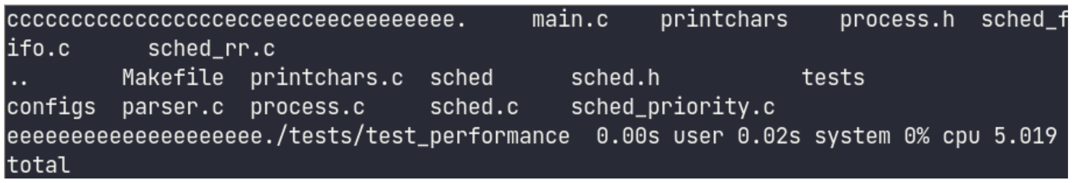
```
❯ time ./tests/test_performance
.        main.c    printchars    process.h  sched_fifo.c      sched_rr.c
..       Makefile  printchars.c  sched      sched.h           tests
configs  parser.c  process.c     sched.c    sched_priority.c
cccccccccccccccccccceeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee./tests/test_performance  0.
00s user 0.01s system 0% cpu 5.520 total
```

- RR Scheduler



```
❯ time ./tests/test_performance
.        main.c    printchars    process.h  sched_fifo.c      sched_rr.c
..       Makefile  printchars.c  sched      sched.h           tests
configs  parser.c  process.c     sched.c    sched_priority.c
cecceecceecceecceecceecceecceecceecceecceeceeeeeeeeeeee./tests/test_performance  0.
00s user 0.01s system 0% cpu 3.519 total
```

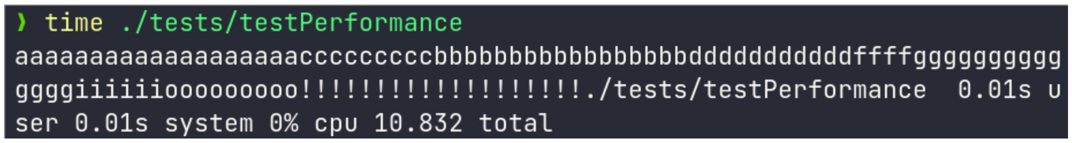
- Priority Scheduler



```
ccccccccccccccccccecceecceeceeeeeeeee.     main.c    printchars    process.h  sched_f
ifo.c      sched_rr.c
..       Makefile  printchars.c  sched      sched.h           tests
configs  parser.c  process.c     sched.c    sched_priority.c
eeeeeeeeeeeeeeeeeeee./tests/test_performance  0.00s user 0.02s system 0% cpu 5.019
total
```

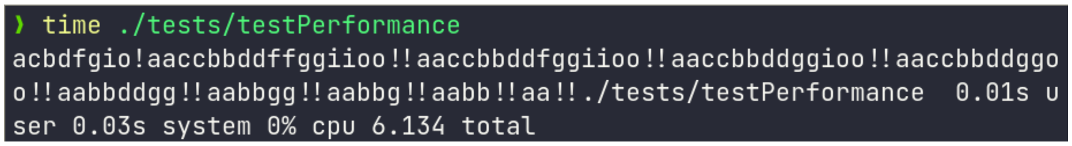
However, I this example uses arbitrary priorities for the items in the configuration, and the time is for the total average wait time. I created a separate test which took into consideration priority by creating another config file with lots of low-priority tasks, and then added a high-priority task. The scheduler then waits for the high-priority task to complete and exits immediately.

- FIFO Scheduler



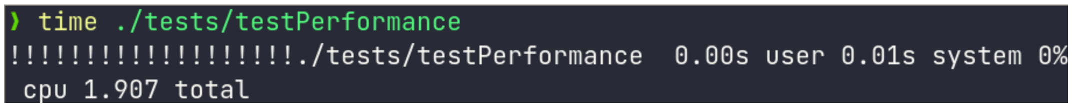
```
❯ time ./tests/testPerformance
aaaaaaaaaaaaaaaaaaacccccccccbbbbbbbbbbbbbbbbbdddddddddddffffgggggggggg
ggggiiiiiiooooooooo!!!!!!!!!!!!!!!!!!!./tests/testPerformance  0.01s u
ser 0.01s system 0% cpu 10.832 total
```

- RR Scheduler



```
❯ time ./tests/testPerformance
acbdfgio!aaccbbddffggiioo!!aaccbbddfggiioo!!aaccbbddggioo!!aaccbbddggo
o!!aabbddgg!!aabbgg!!aabbg!!aabb!!aa!!./tests/testPerformance  0.01s u
ser 0.03s system 0% cpu 6.134 total
```

- Priority Scheduler



```
❯ time ./tests/testPerformance
!!!!!!!!!!!!!!!!!!!!./tests/testPerformance  0.00s user 0.01s system 0%
 cpu 1.907 total
```

2.2 Potential Improvements

As explained in the design section, I would improve my solution by changing the implementation of process ageing to make old process increase in priority rather than frequently ran processes decreasing in priority.

This is because I would also improve the scheduler by allowing processes to be added whilst the scheduler is running, and not make the scheduler stop running when all processes have been completed. With a typical CPU scheduler, the scheduler is always waiting for new processes to be added, and it would be possible to send signals to the scheduler with new processes during exection to be admitted.