CS 367 Project 1 - Fall 2022: TRILBY User Manual

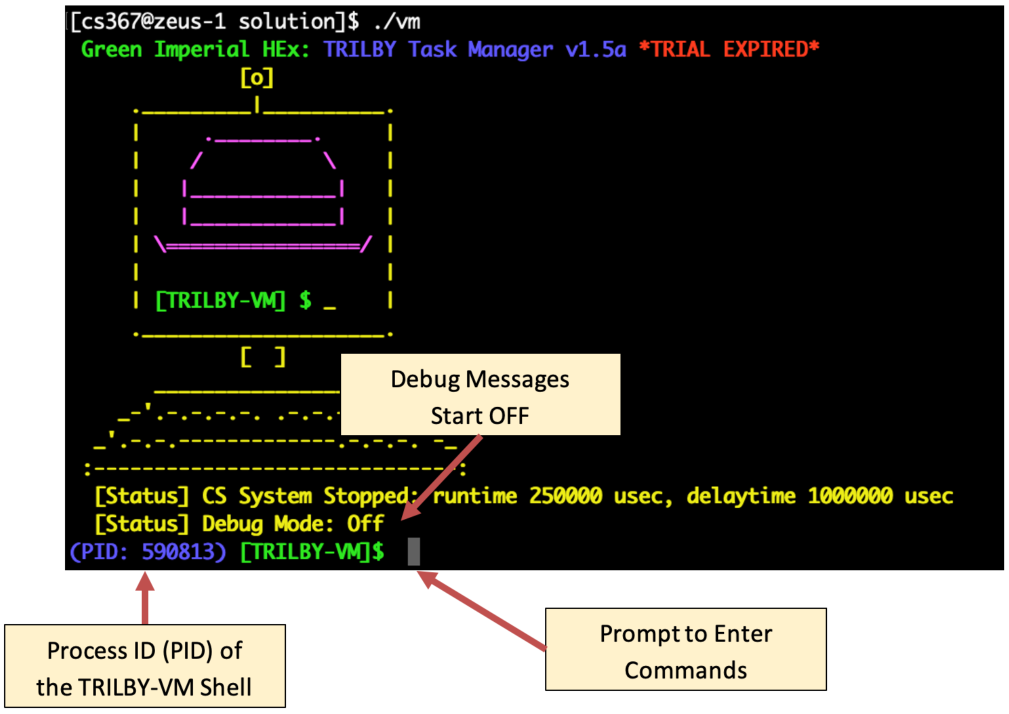
And Sample Executions

1 Introduction

Your code in **op_sched.c** forms one library in the TRILBY Virtual Machine. You can test your code entirely separately from TRILBY (see P1_Self_Testing.pdf), but if you want to test the code running in full with the main VM, this manual will help you understand the various commands to use when running the TRILBY VM (the vm executable that is generated when you run make).

1.1 TRILBY-VM Details

TRILBY-VM is not a simulator, it provides a shell that you can use to run normal non-interactive Linux commands like **ls**, **cal**, and **clear**. It can deal with arguments (like **ls -al** or **wc foo.txt**) but it cannot deal with any commands with an interactive interface – you cannot use it run **wc** without arguments, or to run **vim**, for example.



```
[cs367@zeus-1 solution]$ ./vm
 Green Imperial HEx: TRILBY Task Manager v1.5a *TRIAL EXPIRED*
              [o]
     .__________|__________.
     |                     |
     |     .__________.    |
     |    /            \   |
     |   |______________|  |
     |   |______________|  |
     | \================/  |
     |                     |
     | [TRILBY-VM] $ _     |
     .______________________.
              [ ]
  [Status] CS System Stopped: runtime 250000 usec, delaytime 1000000 usec
  [Status] Debug Mode: Off
(PID: 590813) [TRILBY-VM]$
```

When TRILBY-VM starts, you will get a prompt just like it was a Shell in Linux. You can enter two types of commands here: **Built-In TRILBY Commands** or **Programs**. Built-In commands are used to control TRILBY-VM (such as '**quit**'). Programs are normal Linux programs that you can run from either the current directory or from /usr/bin (like **'cal'** or **'ls'**)

When you start TRILBY-VM, the main execution loop, which is called the Context Switch (CS) system, is not running. This is a nice feature that we're going to abuse for our own debugging as well. To start the CS system that will be looping and controlling the programs to be run on the CPU, use the built-in command **start**. You can stop it at any time with **stop**.

When the CS system is running, any processes you have entered to run will be in the Ready Queue and will start being executed. When you finish writing your OP Scheduler code, then this CS system **calls your functions** in this sequence:

On Startup of TRILBY-VM

- **op_create** Create a new Scheduler and the three Queues

Command Entry (you can enter commands at any time)

- **op_new_process** Create a new Process Struct and Initialize its Values
- **op_add** Insert a Process Struct into a Ready Queue

CS Engine Main Loop

- **op_select_high** Chooses the Process to Run Next from the Ready Queue – High
- If op_select_high returned NULL...
    a. **op_select_low** Chooses a Process to Run from the Ready Queue – Low
- **op_promote_processes** Ages Low Priority Processes and Moves some to High
- < At this point, we run the selected process for 250ms >
- If the process did not exit during this run...
    a. **op_add** Insert a Process Struct into a Ready Queue
- else, if the process did exit during this run...
    a. **op_exited** Insert a Process Struct into the Defunct Queue

CS Engine Shutdown

- **op_deallocate** Deallocate all Allocated Memory.

TRILBY-VM Command-Based Calls

- **op_terminated** Move a Process Struct into the Defunct Queue
- **op_get_count** Returns the number of Processes in a given Queue

TRILBY will choose a process to run, then it will run it for a little while (default is 250ms), then it will return the process and repeat those steps until all processes have finished. The selection algorithm is detailed in the main Project Documentation.

Very generally speaking, when we run processes, they will cycle through in a **round-robin** manner. The exceptions are if the process is low-priority, in which case it will run far less frequently, or if a process is critical, in which case it will run immediately until it finishes.

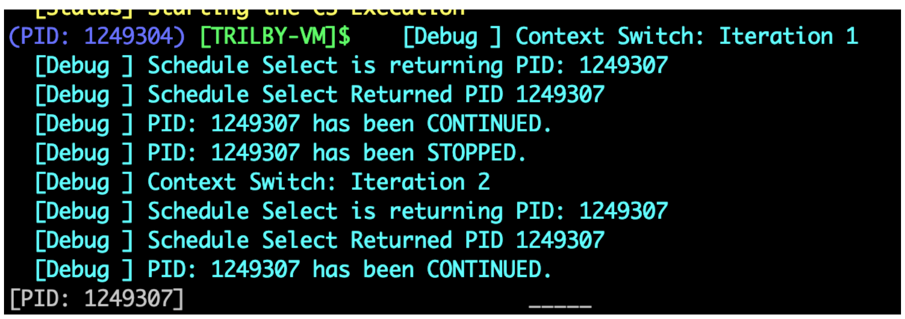
For very simple processes, like 'cal', those programs will finish in a single execution on the CPU. For something that runs for a long time, like 'slow_cooker', you will see it run for a long time on its own, so it will keep getting selected to run on the CPU over and over and it'll keep doing that until it's finished.

This, of course, also only works once your code is written.

2 Notes on Execution

When a process is scheduled to run, it will be executed on the CPU for 250ms. Some processes, however, take longer than this before they start printing out their first output! So, you may have to wait for it to be scheduled a few times before it prints out results.

If you want to make sure the scheduling is happening properly, you can use the **debug** option:



```
[Status] Starting the OS Execution
(PID: 1249304) [TRILBY-VM]$    [Debug ] Context Switch: Iteration 1
  [Debug ] Schedule Select is returning PID: 1249307
  [Debug ] Schedule Select Returned PID 1249307
  [Debug ] PID: 1249307 has been CONTINUED.
  [Debug ] PID: 1249307 has been STOPPED.
  [Debug ] Context Switch: Iteration 2
  [Debug ] Schedule Select is returning PID: 1249307
  [Debug ] Schedule Select Returned PID 1249307
  [Debug ] PID: 1249307 has been CONTINUED.
[PID: 1249307]                    _____
```

In this screenshot here, we started the engine and slow_hat was selected (returned from op_select_high) to run. We see this happening in Iteration 1. When debug says it was CONTINUED, it means the process is being run on the CPU. When debug says it is STOPPED, it means it is being taken off the CPU. The 250ms run didn't produce any output, but debug mode let us see that it was actually running properly. The next loop calls op_select_high again and now leads to the process printing.

One more note is that there are multiple processes running at the same time and they're all using the same screen, so sometimes text appears to be cut-off, or on multiple lines, or without a prompt. This is all normal! It's just because many programs are sharing the same screen.

Press ENTER at any time to get a clean prompt.

3 Building and Running TRILBY-VM (./vm)

You will receive the file **project1_handout.tar**, which will create a handout folder on Zeus.

```
kandrea@zeus-1:handout$ tar -xvf project1_handout.tar
```

In the handout folder, you will have three directories (**src, inc,** and **obj**). The source files are all in **src/**. The one you're working with is **op_sched.c**, which is the only file you will be modifying and submitting. You will also have very useful header files (**op_sched.h**) along with other files for the simulator itself in the **inc/** directory.

3.1 Building TRILBY-VM

To build TRILBY-VM, run the make command:

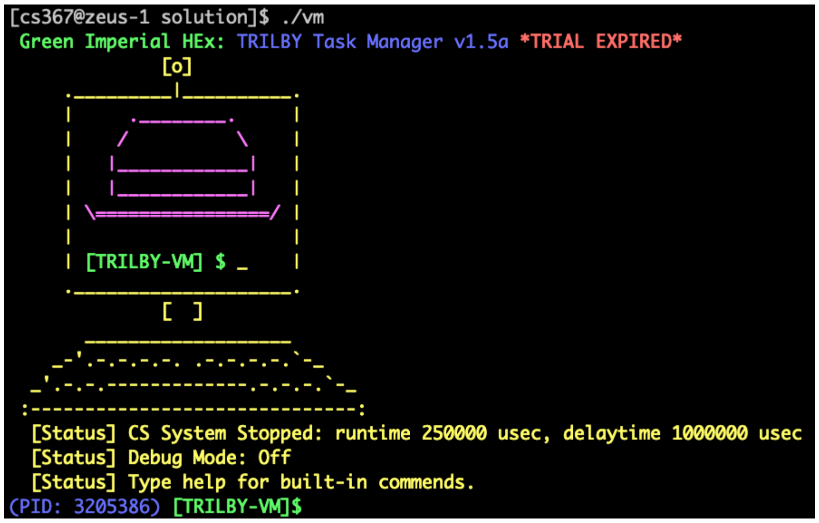
```
kandrea@zeus-1:handout$ make
gcc -std=gnu99  -Og -Wall -Werror -Wno-error=unused-variable -Wno-error=unused-function -
pthread -I./inc -L./obj -g -c -o obj/vm.o src/vm.c
gcc -std=gnu99  -Og -Wall -Werror -Wno-error=unused-variable -Wno-error=unused-function -
pthread -I./inc -L./obj -g -c -o obj/vm_cs.o src/vm_cs.c
gcc -std=gnu99  -Og -Wall -Werror -Wno-error=unused-variable -Wno-error=unused-function -
pthread -I./inc -L./obj -g -c -o obj/vm_shell.o src/vm_shell.c
gcc -std=gnu99  -Og -Wall -Werror -Wno-error=unused-variable -Wno-error=unused-function -
pthread -I./inc -L./obj -g -c -o obj/vm_support.o src/vm_support.c
gcc -std=gnu99  -Og -Wall -Werror -Wno-error=unused-variable -Wno-error=unused-function -
pthread -I./inc -L./obj -g -o vm ./obj/vm.o ./obj/vm_cs.o ./obj/vm_shell.o ./obj/op_sched.o
./obj/vm_support.o -lvm_sd
```

This may be the first time you are building a large project. Your code is just one small piece.

3.2 Running TRILBY-VM

To start TRILBY-VM, run **./vm**

This will give you the shell interface for the VM, which you can use to enter your commands into.



```
[cs367@zeus-1 solution]$ ./vm
 Green Imperial HEx: TRILBY Task Manager v1.5a *TRIAL EXPIRED*
              [o]
     ._________|__________.
     |     .________.     |
     |    /          \    |
     |   |____________|   |
     |   |____________|   |
     | \================/ |
     |                    |
     | [TRILBY-VM] $ _    |
     .____________________.
              [  ]

     ______________________
    _-'.-.-.-.-. .-.-.-.-.`-_
  _'.-.-.-------------.-.-.-.`-_
 :------------------------------:
  [Status] CS System Stopped: runtime 250000 usec, delaytime 1000000 usec
  [Status] Debug Mode: Off
  [Status] Type help for built-in commends.
(PID: 3205386) [TRILBY-VM]$
```

At the prompt, you can either type in the name of a program you want to run, or you can type in one of the TRILBY-VM commands to control the VM. All of the TRILBY-VM Commands are detailed after the function details of what you will be writing.

3.3 Compiling Options

There is one very important note about our compiling options that may differ from what you used in CS262/CS222 (or other C classes). We're using **-Wall -Werror**. This means that it'll warn you about a lot of bad practices and makes every warning into an error.

To compile your program, you will need to address all warnings!

4 TRILBY-VM Manual

TRILBY-VM is a full-featured Virtual Environment that lets you run normal Linux commands and programs, but with a custom scheduler.

The basic idea is that when the Context Switch (CS) engine is running, it will call op_select_high (or op_select_low) to get a process to run from your code, then it will run that process for a **runtime** number of microseconds (usec). The CS engine will then stop it from running on the CPU and return it (op_add or op_exit). Finally, it waits for **delaytime** microseconds before getting the next process.

In a real live environment, **runtime** would be about 3000usec (3ms) and **delaytime** would be 0usec. But, we want to be able to follow the action and make it easier to debug, so the default values for this system in our class is **runtime** is set to 250000usec (0.25 sec) and **delaytime** is 1000000usec (1 sec).

TRILBY-VM starts up with all of its internal debug messages turned off and with the CS engine off. Because of this, when you start it up, you can type in commands to run and nothing will happen! To make them actually run, you need to start the CS engine.

Built-In Commands to Control the CS Engine

- **help** Prints out this reference.
- **start** This will start the CS Engine running.
    - Pressing ctrl-C will toggle the CS Engine as well.
- **stop** This will stop the CS Engine running. (You can start and stop it as much as you want!)
    - Pressing ctrl-C will toggle the CS Engine as well.
- **status** This will print out a status message about the CS Engine settings and if it's running.
- **schedule** This will print out the status of all three Queues.
- **terminate X** This will terminate a process from your Queues with PID X.
- **debug** Toggles extra debug messages.
- **runtime X** This will change the **runtime** to a new usec value. (Default 250000 usec)
- **delaytime X** This will change the **delaytime** to a new usec value. (Default 1000000 usec)

You can also press Enter at any time to see the prompt if it is overwritten by program output.

- This is a very interesting concept! We have multiple programs running and writing to the screen at the same time, so you may see output from programs interrupting the command you're typing.
    - This doesn't affect your command at all, you can keep typing and hit enter.

```
[Status] CS System Stopped: runtime 250000 usec, delaytime 1000000 usec
  [Status] Debug Mode: Off
(PID: 178230) [TRILBY-VM]$  runtime 500000
  [Status] Setting CS System: runtime 500000 usec, delaytime 1000000 usec
(PID: 178230) [TRILBY-VM]$  delaytime 2000000
  [Status] Setting CS System: runtime 500000 usec, delaytime 2000000 usec
(PID: 178230) [TRILBY-VM]$  status
  [Status] CS System Stopped: runtime 500000 usec, delaytime 2000000 usec
(PID: 178230) [TRILBY-VM]$  start
  [Status] Starting the CS Execution
(PID: 178230) [TRILBY-VM]$
```

Before you have written anything for your op_sched.c functions, this is all it will do. For the CS Engine to do anything, it needs to get Processes from your code. You will need to implement op_create, op_add, op_new_process, op_select_high, and op_get_count, at a minimum, for the system to begin processing.

The **Shell** component of TRILBY-VM lets you run programs just like Linux. When you enter the program names and arguments, the shell will call your functions to create and add those programs to your Linked Lists. When the CS Engine is running, you'll see the output of those programs, and when the CS Engine is stopped, you'll see nothing because nothing is running.

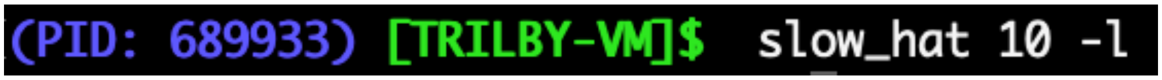
Built-In Commands for the TRILBY-VM Shell

- **schedule** This will print out all of the processes in all three of your Linked Lists!
- [Linux Command] You can enter any common Linux Command with arguments to run.
- [Local Command] Note, there are four special programs you can run that have very long outputs.
    - These are a lot easier to debug because you can see them being run over a long time.
    - **slow_cooker [X]** Prints out one message every second for 10 seconds (or X sec)
        - **Exits with exit code 0**
    - **slow_printer [X]** Prints out one message every half-second for 10 (or X) iterations
        - **Exits with exit code X**
    - **slow_bug** Prints out an ASCII art of a Bug Hunting Knight, one line per 0.5 sec
        - **Exits with exit code 0**
    - **slow_hat** Prints out an ASCII art of the Red Hat Logo, one line per 0.5 sec
        - **Exits with exit code 0**
        - **Courtesy of https://www.asciiart.eu/computers/bug (Author Unknown)**
- **debug** Toggles the TRILBY-VM Debug Messages On/Off (can be spammy)
- **quit** Shuts EVERYTHING down responsibly and quits. (calls your op_deallocate)

Special Options for Running Processes in the TRILBY-VM Shell (Use these AFTER the Name and Arguments)

- **-l** Run the process at Low Priority Level. (Without this option, processes default to High)
- **-c** Run the process with Critical Priority. (Always runs first to completion)

Here is an example of the Special Options to run slow_hat at Low Priority, with command line argument 10.



```
(PID: 689933) [TRILBY-VM]$  slow_hat 10 -l
```

You may also edit the top portion of **inc/vm_settings.h** header. This has all of the initial default settings for running TRILBY-VM in it. Most of this is changeable with the above commands, however, this lets you disable the colors (change to **#define USE_COLORS 0**) if you wish. Do not modify any code below the line that says do not modify anything below this line. Once changed simply run '**make**' again as normal.

4.1 Notes on Concurrency and Visual Interruptions

You can type in new commands to run while the CS Engine is running, but you may notice that your typing gets interrupted anytime a process outputs its text to the screen! This is because this is a multi-tasking virtual machine. Don't worry, the text you were typing is still there so you can keep typing and hit enter and it'll still work.

If you ever want to see the prompt again, you can always just hit Enter without typing anything.

If you are having a lot of trouble with the CS Engine interrupting you, you can use Control-C to toggle the CS Engine on and off as well. (This is more of a fall-back option and doesn't play well with GDB).

5 TRILBY-VM Sample Runs

Here is an example run. We added line numbers on the left to help with the explanation, which is after the output sample. Here's a quick description of some of the programs being run:

- slow_cooker: This program counts from X down to 0, printing once every second.
- slow_printer: This program counts up from 0 to X, printing once every half-second.
- slow_hat: This program prints out an ASCII Art image, one line every half-second.
- slow_bug: This program prints out an ASCII Art image, one line every half-second.

```
1   (PID: 615424) [TRILBY-VM]$  slow_cooker
2   (PID: 615424) [TRILBY-VM]$  slow_bug
3   (PID: 615424) [TRILBY-VM]$  cal
4   (PID: 615424) [TRILBY-VM]$  slow_printer
5   (PID: 615424) [TRILBY-VM]$  start
6     [Status] Starting the CS Execution
7   (PID: 615424) [TRILBY-VM]$  [PID: 615428] slow_cooker count down: 10 ...
8   [PID: 616432] Fight Bugs                      |     |
9       January 2023
10  Su Mo Tu We Th Fr Sa
11   1  2  3  4  5  6  7
12   8  9 10 11 12 13 14
13  15 16 17 18 19 20 21
14  22 23 24 25 26 27 28
15  29 30 31
16
17  [PID: 616434] slow_printer 0...
18  [PID: 615428] slow_cooker count down: 9 ...
19  [PID: 616432]                                 \\_V_//
20  [PID: 616434] slow_printer 1...
21  [PID: 615428] slow_cooker count down: 8 ...
22  [PID: 616432]                                 \/=|=\/
23  [PID: 616434] slow_printer 2...
24  [PID: 615428] slow_cooker count down: 7 ...
25  [PID: 616432]                                  [=v=]
26  [PID: 616434] slow_printer 3...
```

There are a few important things to note when looking at this portion of the sample output:

1. These are real processes running on a real computer.
   a. They don't all start and run at the same speed.
   b. Each run may be slightly different due to these factors.
   c. It may take being selected twice before any program produces output!
      i. Sometimes the first run is all setup before it gets to the first printf!
2. This is a **concurrent** environment.
   a. The prompt on TRILBY is running on a different processor than the programs.
   b. You may see a missing prompt from time to time.
      i. This happens because it may have printed earlier!

ii. Simply press the Enter key to get another prompt. It's ok.

c. You may see an extra prompt from time to time.

i. I/O on a real system is tricky in a concurrent environment. It's OK!

d. You may see program output at the end of other lines.

Let's look at this output line by line:

Lines 1-4 show us entering programs to run in that order.
Line 5 shows us entering a built-in command 'start' to start the engine to run the programs.
Line 6 is a status output to let us know we're now running processes.

At this point, TRILBY will call your code to pick the first process. OP told it to run slow_cooker.

Line 7 shows the prompt after out start command AND the output of the first process!
This is showing that the TRILBY-VM and the process we chose to run are running concurrently.
Line 8 shows the next process being selected and printing one line (this is slow_bug).
Lines 9-16 are actually the full output from the cal program. It was able to do all of its execution in just scheduling!
Lines 17-26 show how the remaining processes (slow_printer, slow_cooker, and slow_bug) get scheduled, run for a little while, then go back to the end of the Ready Queue – High, while the next one gets picked. So, high priority processes will alternate until they all have finished running.

The sample run continues after line 26 with that alternating pattern until all three processes have completed.

5.1 TRILBY-VM Sample Run with Critical and Low Priority Processes

Here is an example run with critical and low-priority processes.

```
(PID: 661049) [TRILBY-VM]$  slow_cooker 20
(PID: 661049) [TRILBY-VM]$  slow_printer -c
(PID: 661049) [TRILBY-VM]$  slow_bug -l
(PID: 661049) [TRILBY-VM]$  schedule
  [Status] Printing the current Schedule Status...
  [Status] ...[Ready - High Priority Queue - 2 Processes]
  [Status]      [PID :661051] slow_cooker
  [Status]      [PID :661053] [C] slow_printer
  [Status] ...[Ready - Low Priority Queue - 1 Processes]
  [Status]      [PID :661054] [L] slow_bug
  [Status] ...[Defunct Queue - 0 Processes]
(PID: 661049) [TRILBY-VM]$  start
  [Status] Starting the CS Execution
(PID: 661049) [TRILBY-VM]$  [PID: 661053] slow_printer 0...
[PID: 661053] slow_printer 1...
[PID: 661053] slow_printer 2...
[PID: 661053] slow_printer 3...
[PID: 661051] slow_cooker count down: 20 ...
[PID: 661051] slow_cooker count down: 19 ...
[PID: 661051] slow_cooker count down: 18 ...
[PID: 661051] slow_cooker count down: 17 ...
[PID: 661051] slow_cooker count down: 16 ...
[PID: 661054] Fight Bugs                      |     |
[PID: 661051] slow_cooker count down: 15 ...
[PID: 661051] slow_cooker count down: 14 ...
[PID: 661051] slow_cooker count down: 13 ...
[PID: 661051] slow_cooker count down: 12 ...
[PID: 661051] slow_cooker count down: 11 ...
[PID: 661054]                                 \\_V_//
[PID: 661051] slow_cooker count down: 10 ...
[PID: 661051] slow_cooker count down: 9 ...
[PID: 661051] slow_cooker count down: 8 ...
[PID: 661051] slow_cooker count down: 7 ...
[PID: 661051] slow_cooker count down: 6 ...
[PID: 661054]                                 \/=|=\/
[PID: 661051] slow_cooker count down: 5 ...
[PID: 661051] slow_cooker count down: 4 ...
[PID: 661051] slow_cooker count down: 3 ...
[PID: 661051] slow_cooker count down: 2 ...
[PID: 661051] slow_cooker count down: 1 ...
[PID: 661054]                                  [=v=]
[PID: 661051] slow_cooker count down: 0 ...
[PID: 661054]                                __\___/_____
[PID: 661054]                               /..[  _____  ]
[PID: 661054]                              /_  [ [  M /] ]
```

Let's look at this output line by line:

Lines 1-4 show us entering programs to run in that order.
Line 2 shows using the **-c** flag at the end of the process to run as Critical
Line 3 shows using the **-l** flag at the end of the process to run as Low Priority

Because slow_printer is run as Critical, it will always be selected first and will be picked over and over again until it completely finishes running, which is what we see on lines 14-17.

Line 18 shows the slow_cooker starting to run. This is the only high priority process, so it has no competition and will continue to run. However, every time a process is selected, TRILBY-VM calls your **op_promote_processes** function to increment the age of the processes in the Ready Queue – Low linked list.

After 5 (MAX_AGE) times being skipped over, the process in the Ready Queue – Low linked list will also be promoted to be put in the end of the Ready Queue – High, which gives it a chance to run! We don't see this happening in the output, but it happens first after Line 17. Since it has no output on its first run, we don't see slow_bug starting immediately, however on Line 23, after the next 5 times it's skipped, it will be promoted again and now we see it running its first output.

From this point on, every 5 times the processes from the Ready Queue – High linked list are picked, we get slow_bug from the Ready Queue – Low that gets moved up to the Ready Queue – High and gets a chance to run.

After line 42, all Ready Queue – High processes have completed, so now that **op_select_high** is returning NULL, **op_select_low** finally gets to be called and we see slow_bug running every iteration now.

Of course, in the middle of slow_bug's run, if we started a new high priority process, then slow_bug would go back to having to wait again.