cgroups_py: Using Linux Control Groups and Systemd to Manage CPU Time and Memory

Curtis Maves
Research Computing
Purdue University
West Lafayette, Indiana, USA
cmaves@purdue.edu

Jason St. John
Research Computing
Purdue University
West Lafayette, Indiana, USA
jstjohn@purdue.edu

***Abstract*—cgroups provide a mechanism to limit user and process resource consumption on Linux systems. This paper discusses *cgroups_py*, a Python script that runs as a systemd service that dynamically throttles users on shared resource systems, such as HPC cluster front-ends. This is done using the cgroups kernel API and systemd.**

***Index Terms*—throttling, cgroups, systemd**

I. Introduction

A frequent problem on any shared resource with many users is that a few users may over consume memory and CPU time. When these resources are exhausted, other users have degraded access to the system. A mechanism is needed to prevent this.

By placing throttles on physical memory consumption and CPU time for each individual user, *cgroups_py* prevents resource exhaustion on a shared system and ensures continued access for other users. If systemd is configured correctly, it will automatically generate a cgroup for each user, called *user-<uid>.slice*, containing all of their login sessions. This provides a convenient mechanism with which to throttle users' resource consumption.

While resource management systems like TORQUE and Slurm provide this functionality for cluster jobs, there is no mechanism for controlling computing resources on more traditional shared systems that are not based on the job submission model (i.e. cluster frontends available to users to submit jobs to a cluster.)

II. Background

A. What are cgroups?

The Linux kernel provides Control Groups, more commonly shortened to *cgroups*, as a feature that provides a way for processes to be grouped together and have their resources limited.

Cgroups are organized in a hierarchical structure, with each cgroup containing processes, and/or child cgroups. Limits for CPU time, memory, or I/O can then be applied to each cgroup. The resources allocated by this limit are then shared by each process in the cgroup and processes in all child cgroups.

In the classic cgroups, each different resource controller, such as CPU, memory, or blkio, get their own cgroupfs. These filesystems are mounted at `/sys/fs/cgroup/<controller>`, and they allow for the control of each individual resource. [1] Each directory in a cgroupsfs hierarchy represents a cgroup. Subdirectories of a directory represent child cgroups of a parent cgroup. Within each directory of the cgroups tree, various files are exposed that allow for the control of the cgroup from userspace via writes, and the obtainment of statistics about the cgroup via reads [1].

B. Systemd and cgroups

Systemd uses a named cgroup tree (called `systemd`)—that does not impose any resource limits—to manage processes because it provides a convenient way to organize processes in a hierarchical structure.

At the top of the hierarchy, systemd creates three cgroups [2]:

- **system.slice**: This contains every non-user systemd service. Each service runs as a child cgroup under `system.slice`.
- **user.slice**: This contains cgroups which contain the login sessions created by `systemd-logind` and systemd user services. These cgroups are called *user-<uid>.slice* and are where *cgroups_py* sets its limits.
- **machine.slice**: This contains virtual machines and containers registered via `systemd-machined`.

A diagram of the final tree representing how systemd units are organized is shown in Fig. 1.

Because processes inherit their cgroup from their parent process after a `fork` or `exec` call, services and user sessions can easily be monitored by systemd. Systemd has the ability to create trees identical to the `systemd` tree in the other cgroup controller hierarchies and can use these trees to limit resources. The *cgroups_py* program uses this systemd capability to impose resource limits on users across all their sessions and user services [3].

III. The *cgroups_py* Approach

A. Previous Approach

The problem of users using too many resource on HPC cluster front-ends is not new. Purdue RCAC's previous approach to

[1] There are two versions of the cgroups tree. The original (v1) is discussed here. The newer *unified cgroups* (v2), combines every individual controller into a unified hierarchy. *cgroups_py* is written to work with the original cgroups trees, rather then the unified hierarchy; although, switching would be a simple matter of changing some paths in the source code.

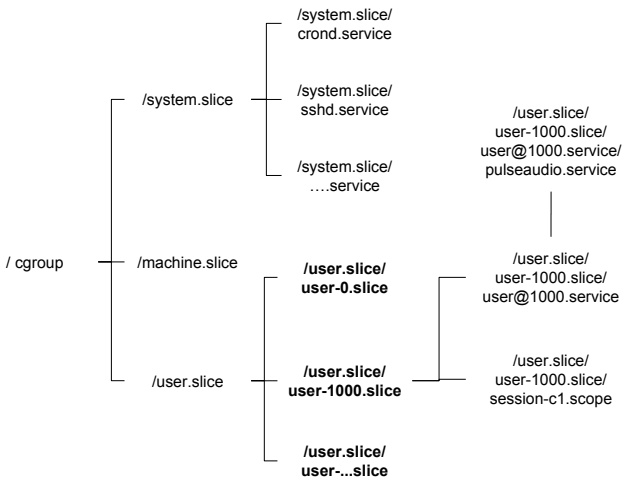

Fig. 1. A tree that represents the cgroups hierarchy that systemd creates for all of its units. The **bolded nodes** represent the cgroups on which *cgroups_py* sets resource limits.

limiting came in the form of *procman.py*, which is a script that was run at a regular interval. During each iteration, it would parse output from `ps`, look for users whose total memory consumption or CPU time was too high, and select a process to kill from these users if they exceeded defined thresholds. Users would be emailed and the event would be logged in the system logs.

This approach was undesirable because it ran on a two minute interval. This provided a window in which a user could potentially over-consume resources, temporarily denying service. This is further compounded because the script would run as an ordinary user-space process. Under memory pressure situations with heavy swapping, it was possible that it would hang for a significant amount of time until it could properly complete its execution.

Additionally, a script named "cgroup_py", which was created by Indiana University, was run to place processes into cgroups on RHEL 6 systems [4]. Unfortunately, this script is not compatible with CentOS 7 and other systemd-based systems. This incompatibility, along with the major changes to cgroups between versions 6 and 7, led to the creation of an entirely new script, *cgroups_py*.

B. The new cgroups_py Approach

The new *cgroups_py* script takes advantage of resources provided by systemd that were previously unavailable. The most important resource is systemd's ability to create a separate cgroup for each user. This provides a convenient mechanism for monitoring and throttling users with a kernel mechanism that directly hooks into the CPU scheduler and the page fault handler.

For CPU throttling, we monitor every user's CPU usage by reading the `cpuacct.usage` file present in each user's cgroup. This file is read every two seconds and the Δ is used to determine CPU usage during each two second interval. Users whose CPU usage is over 5% of the system's total CPU time are placed on a list of throttled users. They are throttled to Max CPU time as defined in Eq. 1:

$$\begin{aligned} n &= \text{Number of users over } 5\%, \\ \text{Max CPU time} &= \begin{cases} 5\% & n \geq 16 \\ \frac{80\%}{n} & n < 16 \end{cases}. \end{aligned} \tag{1}$$

Rather than kill processes, users have their CPU time limited. It is less intrusive to have a program slowed down. This simply results in the program taking longer to run, while maintaining enough free CPU time for other users to carry out activities. The throttled user will experience no data loss, unlike if processes were killed. When a user is throttled a message is printed to stdout. If run as systemd service, the messages will be placed in the system journal by default allowing for logging.

A dynamic, moving CPU time limit was selected over a static limit because it allowed better utilization of CPU resources on the system. There is little harm in allowing a user to use more than their share of CPU resources, as long as it doesn't interfere or slow down other users' experiences. An advantage a static limit would have had is that it would have result in simpler code, and a lighter weight code. However in practice, *cgroups_py* remains very unobtrusive when running, and the benefit of better CPU utilization on the machine made the additional development time worth it.

Unlike CPU time, which can be spread out over time, memory is a finite resource that is simultaneously shared by all processes on a system. Users who use excessive amounts of physical memory force swapping and bring shared systems to a crawl for all users and services. The only option to maintain a usable and responsive system under memory pressure is to immediately reduce memory usage.

cgroups_py sets a hard memory limit on every `user-<uid>.slice` at 20% of physical system memory. If a user's processes use or request too many pages in resident memory, memory recovery is triggered by the kernel, that attempts to flush pages to swap, and empty file buffers from all processes within that user's cgroup. If the user's processes fail to reign in physical memory usage, then OOM-Killer is invoked on that user's set of processes, until memory usage is back below the limit.

Upon an OOM-Killer event, *cgroups_py* will email the offending user with a message detailing the processes killed, when they were killed and their memory usage at the time of the OOM-Killer event. A message will also be output to *cgroups_py*'s stdout. The email provides users with necessary feedback to prevent this in the future, and prevents confusion from their perspective of why a program would fail.

This method is superior to `procman.py`'s memory throttling because once the cgroup memory limit is in place (which occurs within a few seconds of login), the kernel is responsible for enforcing the cgroup's limit. It is impossible for a set of user-space processes to circumvent their collective limit because the kernel implements the memory limit in the page fault handler [5].

A moving hard limit versus a static limit was debated. The moving limit would set the memory limits significantly higher than 20% when a system is not experiencing memory pressure. As memory became more scarce the limit would be set progressively lower. This would allow for better utilization of the memory on system.

The static limit was ultimately chosen at the cost of poorer memory utilization because it provides consistency. The moving hard limit may cause a program to run normally one day because nobody else is using memory, and be killed the next day because the system is under memory pressure from other users. This inconsistency would lead to confusion by the end-user. A static limit creates a consistent point in which programs will fail due to overzealous memory utilization.

IV. *cgroups_py* IMPLEMENTATION

A. System Requirements

- **Linux:** The kernel must be compiled with support of CPU and Memory Cgroup controllers. The system must then use the original cgroups v1 with separate hierarchies.
- **systemd:** The systemd-logind login manager must be used. The systemd-journald logging service must also be used if used if OOM-Killer events are going to be logged.
- **mailx:** This is required if email functionality is needed.
- **Python 3:** The script was developed and tested using Python 3.4. Using an earlier version may result in the script breaking.

B. Installation and Configuration

The installation of *cgroups_py* is simple. If the above system requirements are met, then the script simple needs to be present on the system as an executable script. Typically the script is run as a systemd service, and runs in the background with no direct interaction. `systemd-journald` will log the output from *cgroups_py* to create a log of throttling and OOM-Killer events triggered by *cgroups_py*.

cgroups_py provides a few command line arguments that can be used to disable various functions. The most notable being that CPU or Memory throttling can be disabled with arguments shown in "Table I".

For control over some of the constants such as the physical memory limit, CPU throttling threshold, and total CPU Time divided between high CPU users, one can change the values to constants at the top of the script.

TABLE I
COMMAND-LINE ARGUMENTS

Short arg	Description
-m	Disables the memory limiting
-c	Disables the CPU throttling
-u	When logging throttling and OOM-Killer events, log the slice instead of the username
-q	Quiet Mode. Do not print anything to stdout
-e	Disable the email functionality of the OOM-Killer notifier

C. cgroups_py Code Details

The *cgroups_py* program is implemented as a class called **Throttler**. When the class constructor is called, after it initializes its variables, it creates a thread that parses JSON-formatted kernel messages from systemd-journald using `journalctl` command. This thread looks for OOM-Killer events, and emails the user responsible, with a message explaining the situation. The class constructor allows one to configure many parameters about the program. These are mostly the same ones that can be configured using the constant at the top of the program.

Once the Throttler object is instantiated, its **forever()** function is called. This calls the Throttler's **iterate()** function in an infinite loop. The iterate function does the bulk of the *cgroups_py*'s work. It fetches an up-to-date list of currently active users. **iterate()** starts two short lived threads, one for setting memory throttling, and another for setting CPU time throttling. This function is set up to make it easy to add additional possible threads that could throttle other resources, such as block IO or network IO. The new active users since the last iteration of **iterate()** are passed to these threads. The memory thread simply calls `systemctl` to set the physical memory limit on new active users, by setting the `MemoryLimit` property of each *user-<uid>.slice*. No logging is done by this thread. OOM-Killer events are only handled by the first `journalctl` parsing thread that was mentioned earlier.

The CPU throttling measures the CPU usage of users by reading the `cpuacct.usage` in their cgroup, over a short interval defined by the **interval** kwarg (defaults to 2 seconds) of the Throttler() constructor. High usage users are logged, and a CPU Time throttle is put in place by setting `CPUQuota` property via `systemctl`.

These steps loop continuously until the script receives a SIGTERM, or SIGINT. Upon receipt of either of these signals, the loop is halted, and the throttles are removed with `systemctl` calls.

V. FUTURE WORK

cgroups_py still has room for improvement:

- Support for cgroups v2's unified hierarchy. The unified hierarchy also introduces better Memory control options, that allow for finer grained control of the Linux Cgroups behavior.
- Adding the capability to throttle block IO and network IO, with their respective cgroups controllers. The script is setup to easily allow the addition of new controllers, and would make *cgroups_py* a more complete solution to controlling user resource consumption.
- Improving the argument parsing to include configuration parameters that can currently only be alter by modifying the script itself.

VI. CONCLUSION

cgroups_py provides a effective tool limiting for managing system resource on a shared system. With the proliferation of Systemd as a fundamental part of most Linux distributions,

as well as the prevalence of Python, *cgroups_py* will run on most systems with no additional dependencies or modification to the system. By taking full advantage of existing features on systems, *cgroups_py* ensures many users can smoothly use a shared system simultaneous, while maintaining a low barrier to installation, and a light footprint.

REFERENCES

[1] P. Menage, "Cgroups," https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[2] *systemd.resource-control Resource control unit settings*, freedesktop.org.

[3] L. Poettering, "systemd and control groups," https://www.youtube.com/watch?v=7CWmuhkgZWs, November 2015.

[4] R. Perigo, "cgroup_py," https://github.com/rperigo/cgroup_py, Indiana University.

[5] "Memory resource controller," https://www.kernel.org/doc/Documentation/cgroup-v1/memory.txt.

APPENDIX

A. Abstract

cgroups provide a mechanism to limit user and process resource consumption on Linux systems. This paper discusses *cgroups_py*, a Python script that runs as a systemd service that dynamically throttles users on shared resource systems, such as HPC cluster front-ends. This is done using the cgroups kernel API and systemd.

B. Description

1) Artifact Meta Information:

- **Program:** *cgroups_py*
- **Run-time environment:** Requires systemd, Python 3 (¿3.4), mailx, and Linux kernel compiled with cgroups support (v1).
- **Execution:** Typically run as a systemd service
- **Output:** Outputs information about throttled users and OOM-Killer events. It also notifies the user who triggered the OOM-Killer event via email.
- **Publicly available?:** Yes

2) How software can be obtained: https://github.com/HPCSYSPROS/Workshop18/tree/master/cgroups_py_Using_Linux_Control_Groups_and_Systemd_to_Manage_CPU_Time_and_Memory

3) Hardware dependencies: None

4) Software dependencies: systemd, Python 3 ($\geq$3.4), mailx, and Linux kernel compiled with cgroups support (v1).

5) Datasets: None

C. Installation

Obligatory if the paper is paired with an artifact.

1) Configure a system to use systemd. (Default on most major Linux distributions)
2) Place the script on an executable and readable location on the system.
3) Modified the *cgroups_py.service* to point to the installed script
4) Install the *cgroups_py.service* file in `/etc/systemd/system/`.
5) Enable the *cgroups_py.service*.

D. Experiment workflow

Start the *cgroups_py.service*. (When the service is first started existing user sessions may not be throttled.)

E. Evaluation and expected result

Users should no longer be able to individually use more 20% of system memory, and users' CPU usage should be throttled among the high CPU users.