Cool Container Crew

GROUP MEMBERS
Alexander Lent [lenta@uw.edu]
Garrett Marconet [gmarco@uw.edu]
Carmen Hanish [carmenyh@uw.edu]
Thomas Nguyen [tomn@uw.edu]
Vadim Eksarevskiy [veksarev@uw.edu]

ABSTRACT

This work describes adding containers to `xv6`, an open source operating system designed for teaching fundamentals and showing the inner workings of an OS. `xv6` is simple enough to be relatively straightforward to reason about, and allows the demonstration of what containers are and how they can be implemented. At the same time, it is robust enough to allow a full-fledged implementation and can allow others to reason about parts that can be further improved and added on in the future, making it the perfect platform to demonstrate a container implementation. In this paper, we address the benefits of containers, how we implemented them in `xv6`, what different variations of them exist, and how containers can be expanded upon.

1. INTRODUCTION

This paper discusses our container implementation for `xv6`, which aims to allow groups to be created with process isolation, memory limiting, and CPU priorities. This was implemented through adding a notion of pgroups. Pgroups are essentially a mix between namespaces and cgroups, providing both in one. Specifically, they add process isolation, by preventing processes in different pgroups from interacting with or seeing each other, except for the root pgroup. Additionally, they also provide I/O isolation for the console, preventing multiple shells running simultaneously in different pgroups from interfering from each other. Throughout this paper, the terms group, pgroup, and container are used interchangeably.

As far as the aspects of pgroups that were implemented, this project added memory limiting, making it possible to limit how much memory a single pgroup can use. Dynamic CPU time distribution based on pgroup settings was attempted initially, but we ran into issues implementing it and ended up giving each container an equal portion of CPU time.

Our implementation supports multi-core configurations, while preserving the integrities guaranteed by our definition of pgroups. Specifically, we support up to 8 cores.

The rest of the paper is organized as follows. §2 gives the background for this project, including linux namespaces and cgroups. §3 details the related work of other implementations of containers, both in `xv6` and in linux. §4 discusses the details of our project, including the initial design and final implementation. §5 presents experimental results, primarily performance effects of our implementation. §6 notes ways to improve this work, including bugs that could be fixed and possible additional features. Finally, §7 describes what lessons we learned during this project, and §8 provides our conclusion from it.

2. BACKGROUND

Containers are a notion similar to that of Virtual Machines. They are made primarily for the purposes of providing isolation and allowing multiple potentially adversarial processes to share a single physical machine. As cloud computing becomes more widespread, guaranteeing a given cloud instance is isolated from others running on the same machine is becoming more important.

Modern Linux containers work by creating namespaces and cgroups that provide different levels of isolation and resource limiting for each container. Namespaces allow each container to have multiple or all of: isolated processes, file system, memory, I/O, etc. They attach each process to a certain container, and only allow that process to access and see things in a way consistent with the view presented by the container.

Cgroups, on the other hand, provide a way to limit the resources used by each container. They do not provide isolation by themselves but set a limit in the amount of memory and/or CPU usage that a single container can consume. This is useful for systems where fair allocation is important, such as cloud systems with multiple users, who may be requesting different amounts of memory and/or CPU usage.

While Virtual Machines, when implemented properly, can provide near complete security and isolation, they incur a large overhead by having multiple guest operating systems running at the same time. When running multiple VM instances of the same guest OS, many of the resources stored by the hypervisor are duplicated and redundant. Additionally, extra overhead is added due to a guest OS having to go through the underlying hypervisor whenever they need to communicate with the hardware. All of this combined introduces a large amount of overhead to simple operations such as disk access, therefore inevitably slowing down the system.

Containers as implemented are inherently less secure than VMs but are good enough for most purposes. They are also more efficient than VMs. They are especially useful in cases such a single user running multiple programs that have different dependencies, relying on different versions of some code. In this case, the user is not concerned about the security, but needs isolation in order to allow both programs to run simultaneously. While VMs can provide this, they do so with a large amount of overhead. In the case of containers, however, this overhead is much lower, while still allowing the programs to be run in tandem due to both seeing their necessary dependency code. Therefore, it is easy to see the particular applications of containers, and why it is interesting to build them in a system like `xv6`, in order to make it easier for others to understand the basics behind containers and to see how they are implemented in a relatively straightforward system.

3. RELATED WORK

On GitHub, there is a repository that adds container support to `xv6` [1]. This work is based on adding a container struct in container.h. The container struct is similar to our pgroup one, but also has a disk use limit, a maximum number of processes for each container, and a map to every process. This repository does memory limiting through modifying kalloc and kfree. This allows them to have control over memory allocation for pages outside of the user virtual memory for each process.

In order to better gain an understanding of container isolation, our group did not look at the existing `xv6c` implementation before our implementation of pgroups was completed.

4. IMPLEMENTATION

4.1 Initial Design

4.1.1 Building `ps`

The first portion to implement is a simplified version of the unix command `ps` which lists information about all processes running on a machine at a given time. This is necessary to show process isolation and resource limiting later on.

To implement `ps`, a `getprocs()` system call is implemented that returns a list of all running processes in the environment and information about them. This system call is then used to implement the rudimentary version of `ps` that prints information about all running processes.

4.1.2 Namespace Data Structure

The next part of implementing containerization is adding support for a basic form of namespaces. This allows processes to be grouped together and for resources to be managed independently between groups.

The main portion of adding namespaces is adding a field to the `proc` struct that is used as an index into an array of namespaces. The existing system calls and kernel exit/entries will be verified to ensure that the kernel operates with namespaces in mind. Furthermore, the rest of the kernel will need to be verified to ensure that other operations (such as process initialization and forking) are done with the knowledge of namespaces in mind.

To create a new namespace, a user will use a `createns()` system call, which will need to be implemented. This system call would create a new namespace object.

4.1.3 Process Isolation

Processes will need to be isolated between containers so that different containers do not see each other's' processes unless the process was from the root where the container was spawned. This isolation should be able to be verified quickly by getprocs`()` and is the basic root of the containerization to be implemented.

4.1.4 CPU Resource Limiting

The goal with CPU Resource limiting is to allow for lower and higher priority containers based on the logical importance of their tasks. The CPU should be optimally utilized and kept as busy as possible if work is needed to be done.

The preliminary idea on how to implement this is to use something akin to a standard OS scheduler. It could use something as simple as timed interrupts to ensure every container gets to run for an equal amount of time, cycling between containers. In an attempt to design a scheduler more from the ground up, the scheduler algorithm for these pgroups was to be designed without looking at scheduler algorithms online.

The initial idea for implementation was to give all processes a ticker to see how long the process has been waiting and bump the ticker by the process' group priority each time the process is skipped by the scheduler, so tasks waiting longer with a higher priority are more likely to be ran by the scheduler next time. This also in theory helps ensure that so long as ridiculous priority ratios are not given to different pgroups, different processes will not starve.

4.1.5 Memory Resource Limiting

To limit access to memory resources, `sbrk()` will be modified to limit the memory allocated to each namespace and processes within it. The amount of memory used by each namespace will be the sum of the number of physical pages used by each of of its processes. The limits will be set on creation of the namespace.

4.2 Final Implementation

4.2.1 pgroup Creation

A `pgroup` struct is created through the user program `pgroupcreate`. `pgroupcreate` takes two optional parameters: the maximum number of pages that can be allocated for this group and the CPU priority given to the group for scheduling purposes. These variables are then stored with the `pgroup` struct and the defaults for them are the maximum number of pages for the memory limit, and the highest CPU priority.

By default, `xv6` launches a shell on startup. Therefore, it also makes sense for each new container to immediately start a new shell for interacting with that container. This shell is a fully-fledged shell that is identical to that of the original one, with the caveat of only being able to see console I/O when the console has the shell's group_id as the current active container. This will be described more in detail later.

This changed from the initial design in the name change from "namespace" to "pgroup," which was to indicate that this implementation falls somewhere between the cgroups and namespaces of the Linux kernel implementation.

4.2.2 Console Isolation

Console isolation is a concept we added to `xv6` that was not in our initial design document. Once we began experimenting with the other features, it became apparent that an intuitive way to direct input and output to a particular process or container would be invaluable.

Our implementation of console isolation is essentially a special case of I/O isolation. Specifically, we modified two methods in `console.c`: `consoleread` and `consolewrite`. For reading input from the console, we check if the running process is within the active pgroup and if it is not, we yield the CPU and continue execution at the start of the input loop.

For isolating console output, we had more options. The first option was to maintain a buffer of output for each pgroup. However, we felt that the storage overhead required was not worth it and we were weary of output possibly being truncated if a process printed too much. Another option for output isolation was to print all lines from any group to the console but prepend the pgroup ID to any line printed from outside of the active group. We decided against this as well, since it would be messy and yield output that is hard to follow and breaks up the running program's output. The solution we ultimately decided to go with was similar to our solution for handling input. If a process in a non-active group attempts to print something, it will return to the start of the output loop and yield the CPU. While this solution limits the amount of background computation a printing process can do, we figured such a solution would be best for our implementation of `xv6` as a research operating system since it resulted in the simplest and cleanest codebase.

With that in mind, if `xv6` were focused more on an efficient user experience, a variation on the first option would be ideal. In this variation, we would maintain a linked-list of strings that each represent a call to `consolewrite` while the associated pgroup is in the background. As soon as the pgroup became active, we would iterate through this linked list, printing each item. This solution would still allow for applications that print to the console to run in the background without dirtying the console while another pgroup is running.

The next steps for I/O isolation is to implement a general solution on `xv6`. Such an implementation would be remarkably similar to our solution for console isolation, in that it would modify the read and write calls from all inodes on `xv6` to be aware of pgroups and operate in a way that does not break isolation.

4.2.3 Process Isolation

In order to support process isolation, each `proc` struct, which stores information about a given process, was modified to add a group_id to it. This is used throughout the whole implementation to support pgroups. It is worth noting that currently it is impossible to change a process's container after it has started running, although this should not be needed. This also has a workaround in the form of forking the current process into a different container, and killing the parent process, which effectively achieves the goal of change a process's container. In a future iteration of our containers in `xv6`, we intend to fix this bug.

Process isolation takes multiple forms, with the most fundamental one being the inability of processes in different containers to interact with each other. Specifically, all the system calls are modified to check that the operation is valid, in the sense that the process doing the operation is not trying to perform it on a process in a container which it should not be able to interact with. This includes the newly added `ps` command, which if it is run from a process within a group, is only able to see and list processes that are in that group.

In addition to this, process creation was modified to ensure that processes in different containers had separate pid counters. This was implemented through modifying the function that creates a new process to instead give each process a pid based on the counter for that specific pgroup, rather than a global counter. Any functions that created new processes before the OS finished booting and set up containers now used a default group_id of 0 for the purpose of pid creation.

4.2.4 CPU Resource Limiting

The hope for allowing priorities to different pgroups was to let certain groups use more processor time in relation to other groups and finish tasks quicker. The caveat to allowing more processor time is that a good scheduler should ensure scheduling is still relatively fair, and that processes do not starve or have an extremely long response time.

After implementing the scheduler algorithm designed out in the initial implementation section that was supposed to prioritize processes based on their pgroup priority, multiple testing programs were written. One approach was to time how long it took to compute a large amount of primes numbers starting from one, and the other was to time and count how many times the scheduler picked a certain process that consistently yielded in relation to other similar processes that were given different priorities.

The testing programs revealed that the scheduler implementation and design did not work well. When there weren't many processes, the scheduler would schedule everything fairly and give equal CPU time. Occasionally with the right number of processes and setup the scheduler would work correctly, giving different amounts of CPU time to different processes based on their group priority, but overall the slight slowdown was not worthwhile and the scheduling was simply not correct. The scheduler may have given equal time to each process because the initial design did not account for such a small number of processes being switched between so quickly, so the counting of wait times did not work correctly. The second attempt at designing the scheduler over produced dramatically different results and did give different CPU times to different processes, but did not prioritize based on the group priority.

The original `xv6` scheduler cycled through all the processes and picked the first runnable one in the table. In the end, the original `xv6` scheduler was reverted to and slightly modified to pick the first runnable process after the last one ran so that each process would be given a similar priority.

4.2.5 Memory Limiting

Memory limiting is implemented through changes to `allocuvm`, `deallocuvm`, and pipe. As such, it does not take into account memory used to set up the kernel portion of each process, but does not any changes to a process' heap. The memory limit for a given group is set at the time of its creation, through the second parameter to the `createpgroup` call. This sets the limit of the number of pages. When any of the affected methods then attempt to allocate pages for a process's use, they call `change_mem_used` to request the amount of pages desired. This method then checks how many pages the current group has used, compared with the maximum amount it may use, and from there returns the number of pages that the given method is allowed to use.

The changes to the initial design were namely that, as part of testing this feature, a user program memleak was added. This attempts to allocate as much memory as possible in the current group, going into an infinite loop when the maximum is allocated. Another change was to have memory requests go through `allocuvm`, `deallocuvm`, and `pipe` rather than `sbrk`, because these methods had the code to allocate or deallocate memory at the page level.

5. EXPERIMENTAL RESULTS

Our implementation of containers in `xv6` has relatively few places that could serve as a bottleneck for performance. All added features or either O(1) or O(n) for n < 64 worst case runtime and memory complexity. That said, we decided to include brief benchmarks and summaries of all user-facing programs and operations.

	Runtime	Memory Usage
`pgroupcreate`	~0ms	sh overhead
`pgrouplist`	~2ms	164 bytes + normal user program overhead
Group switching	~0ms	4 bytes
`ps`	~1ms	94 bytes + normal user program overhead

As one can see from the experimental runtimes, there is near-negligible time added by our container implementation.

5.1 Cost from Modification and Addition of Global Variables

We added a global data structure and expanded one other. The data structure we added was the `grouptable`, which holds a spinlock and 8 pgroups. The `pgroup` struct is currently 20 bytes, and the `grouptable` is 212 bytes.

The structure we expanded was the `ptable`, which we did by adding 2 additional fields, or 8 bytes, to the `proc` struct. This then causes an additional usage of 512 bytes through the `ptable`.

6. NEXT STEPS

6.1 Known Bugs

As with any operating system, our implementation of containers in `xv6` has its fair share of bugs. For one, `pgroupcreate` will fail if run repeatedly without switching to the new groups. We have two ideas for why this bug may be happening. Our first intuition is that it has something to do with too much data being pushed to the stack at once, so new data is going over the edge and causing an issue. Another possible explanation is that our scheduler is faulty and for some reason is not working properly once we get to a certain number of runnable processes. This is supported by the fact that the number of times we can run `pgroupcreate` without it failing (and without switching to the new groups) fluctuates between 3 and 5. Further, we have issues with a shell seemingly randomly crashing when running `pgroupcreate` multiple times in quick succession. We believe this is a symptom of the same underlying bug.

6.2 Desired Features

As was discussed in the introduction to this paper, there are many different types of isolation that can be offered by containers, some of which we chose not to implement for the project. The most useful one that we could later add would be file system isolation, as this would bring containers closer to what VMs offer in terms of functionality. This could be implemented through a file system similar to copy-on-write fork. Such a file system would copy any directory or file that has a change from the root copy into a container. This would essentially achieve a state where all of the containers have their own file system state.

In addition to this, I/O isolation could be expanded to encompass other forms of I/O besides the console (such as network and other devices). However, it is worth noting that currently `xv6` lacks network support, so this would first need to be implemented for network isolation to become meaningful.

Finally, dynamic CPU limiting based on variable parameters would also be a useful feature of our groups. This was originally part of the design, but was cut due to an impractical scheduler design. Implementing this would require advanced knowledge of CPU schedulers.

Traditional linux containers often allow the choice of what to isolate, rather than doing it all at once. For this paper, it was a conscious choice to do it all at once, as this makes reasoning about the model easier and provides a more VM-like level of isolation. However, for purposes where only a single type of isolation might be required, it would be beneficial to split pgroups into different namespaces and cgroups so that a user has more control over the system.

7. LESSONS LEARNED

In working on allowing different priorities to different groups, the scheduler was designed and written without looking at other scheduler implementations. The implementation of the new scheduler algorithm did not work, but a massive amount was learned about the scheduler along the way. In summary, going in blind and trying to design and implement an algorithm using only previous knowledge is not the best way to approach a new topic always. A lot of time was spent between the design and implementation, and a quick check may have revealed how the scheduler did not work the way we had originally thought. That's not to say designing on your own code is bad, but sinking a lot of time into something to not produce results visible by other people is occasionally frowned upon if they did not see the work put in.

The implementation of the scheduler feels similar to hashcodes, where you should not go and write your hashcode algorithm blind but rather read a book first on how to do so, then go and implement said algorithm.

8. CONCLUSION

The intent of this project was to design and implement containers within `xv6`. Process ID isolation was achieved, and multiple pgroups could be launched to run simultaneously on `xv6`. In our implementation, processes cannot see other processes running in different pgroups. In addition, each pgroup starts initially with a dedicated shell. Furthermore, memory limiting was implemented so that pgroups can be constrained to not take up all of the available memory from the underlying shared resources. Designing this project without looking at previous implementations allowed for a deeper understanding about how isolation of resources works.

9. REFERENCES

[1] https://github.com/kierangilliam/xv6c