OPERATING SYSTEMS: ASSIGNMENT 3

April 25, 2019

Jay Kumar Modi (2016CS10356)
Manish Tanwar (2016CS10363)

Virtualization in xv6

1. In this assignment we worked on implementing virtualization in our toy OS: xv6.
2. Container Manager and functionalities have been implemented in the kernal mode.

We implemented container related services to obtain virtualisation in our OS. Various global and local data structures, w.r.t container, were introduced to achieve this:

Container table:

```
struct{
  struct spinlock lock;
  struct container container[NPROC];
  int allocated[NPROC];
} container_table;
```

- It is a global struct, which maintains all the allocated and available containers
- "allocated" array keeps track of empty and allocated containers.

```
struct proc {
  ..
  uint container_id;
  ..
}
```

- Inside the `proc` struct a new variable `container_id` is added which stores in which container this process lies.(Contains **0** if process is not in any container)

1 Container Manager

```
uint create_container(void);
uint destroy_container(uint container_id);
uint join_container(uint container_id);
uint leave_container(void);
```

1. **Container creation**
   - Allocates the available unused data structure to be used as a container and returns the id. This id acts as a pointer to access the data structure

- To ensure the locality of data structure w.r.t container, a unique id is associated with each data structure which is referred as container id.
- Creation also initialises local fields marker for round robin scheduler within the container.
- Return type: if a data structure is succesffuly allocated then the container id, as defined above is returned else 0.

2. **Deleting a container**

- All the processes associated with the container are killed and the data structure corresponding to given container id is de-allocated.
- Return type: if the given id corresponds to some data structure that corresponds to a container, returns 1, else returns 0
-

3. **Joining a container**

- On joining the container id corresponding to that process is changed from 0 to given container id.
- The global table storing mapping of pid to container id is updated.
- Return type: if the given container id corresponds to a valid container then returns 1, else returns 0. Note that here valid is defined as that the container id be positive and some data structure is allocated to it, i.e. it is already created.

4. **Leaving a container**

- When a process requests to leave the container, the container id corresponding to that process is made equal to 0, which, in our setting, corresponds to xv6 kernel.

2 Virtual Scheduler

1. Kernel's scheduler is being modified to work for containers as well.
2. Implementation of virtual scheduler can be understood as consisting of two round-robin schedulers. The main scheduler acts on all the kernel processes and containers treating them all as a single unit.
3. Every container also has it's own scheduler which maintains the last process that got scheduled within the container and also implements a round-robin approach within the container.
4. This implementation ensures that all that all processes and containers get equal scheduling and processes within a container also get equal scheduling.
5. This scheduling strategy is fair as kernel cannot see the processes inside the containers.
6. Consider the following example with two containers and 8 processes:
   - $p0, p1$ in host
   - $p2, p3$ in container $c1$
   - $p4, p5, p6$ in container $c2$

   Scheduling : $p0, p1, p2, p4, p0, p1, p3, p5, p0, p1, p2, p6...$

Log Calls:

```
int scheduler_log_on(void);
int scheduler_log_off(void);
```

It is implemented by keeping a toggle variable:

```
int scheduler_log; // ON or OFF
```

3 Resource Isolation

3.1 Processes (ps):

1. A process inside container c can only see the processes in the container c and any process on host cannot see any process inside a container.

2. This isolation is implemented by `container_id` of each process.

3. **Function** `ps`: If processer p calls `ps()` then it only prints the processes which has same `container_id` as of p.

3.2 File System (`ls`):

1. Resource Isolation in files system to implement virtual file system is obtained by associating the files corresponding to a process with its container id.

2. Every file created by a process running in a particular container has attached with it the container id, which uniquely identifies its locality w.r.t the container. Filename is modified in the following manner for isolation:

   `filename` → `filename$cid`

   (`cid` = Container Id, cid is to be assumed with 2 digits)

3. **ls** :

   - On a process in a container : With the implementation of virtual file system as defined above, ls command only outputs the files that are local to the current container(which contains `$cid` in filename), which gives the appearance of virtualization required.
   - Careful implementation is done to prevent duplication printing in case of same file existing on the host(before joining the container) and within the container.
   - On host : print all the files which were created on the host.

4 Copy on write/open

Copy on Write:

- The implementation can be described as copy on write which considers following cases:
  - File is opened directly if the open file is being requested by non-container process
  - Else if a process in some container (say j) calls to open some file (say "name") then two cases are possible depending on availability of `"name_$_j"`:

* if available, then "`name_$_j`" is opened and it fd returned, else
* Further two cases are possible based on the availablity of "name":
  - If available, and open is requested in write mode then copy of "name", "`name_$_j`" is made and its fd returned
  - Else, a file named "`name_$_j`" is open and its fd returned.