CSE506 PROJECT REPORT

Ankit Gupta
109721857
ankit.gupta.1@stonybrook.edu

Anshul Anshul
109721547
anshul.anshul@stonybrook.edu

PHASE 1

In first phase of the project, we have modified the file-system to be **write-enabled** also. This task was very necessary to be done. As, our next two task: process migration and encryption-decryption of file-system heavily relies on the quality implementation of this phase of the project.

PHASE 2

Process Migration is the activity to transport a process's state from one machine to another machine. Process migration has many advantages some of them are fault resilience and dynamic load distribution etc. There has been much research done in this field. Day after day people are coming up with very clever solutions to migrate process state from one machine to another. There are some pretty good implementation of this concept like OpenMosix and Sprite OS etc. But due to many restrictions and design-limitations in JOS, one cannot successfully build a resilient and scalable process migration. Process migration helped us to understand the virtualisation in particular.
We have implemented the process migration in microkernel fashion where saving, loading, sending and receiving process's state are implemented as user-level programs which are invoked using the shell(icode binary) inside JOS.

Implementation

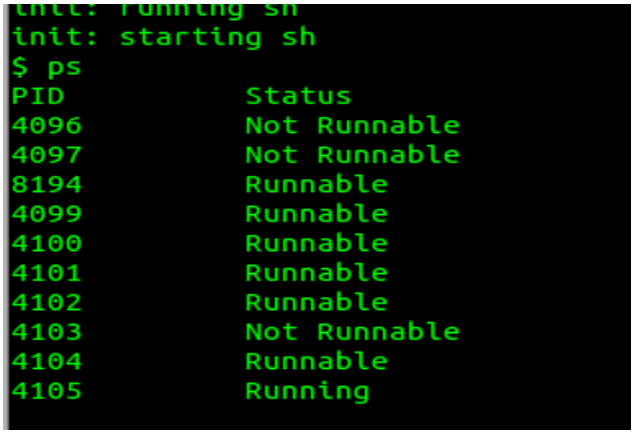
User level Programs Implemented:
ps - to list all the processes currently present on the machine.
pause - to pause the process whose id is passed as parameter
resume - to continue the process which was paused earlier.
save - save page tables, pages, stack and register state of the paused process into an image.
load - load the page tables, pages, stack and register state from image into a new process.
send - send saved state of a process over network to the receiver.
receive - receive state sent over network in small chunks and load it into a new process.

Methodology

We will first boot up the JOS using

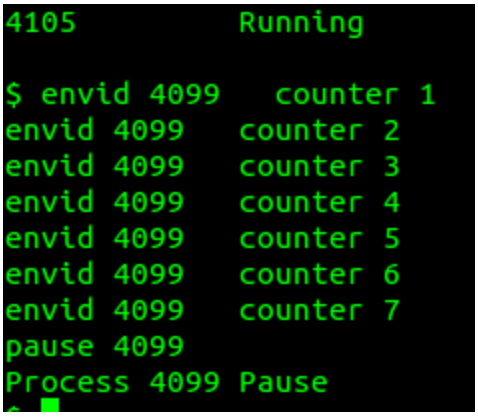
- `make qemu-nox`

After booting it for the first time, we will check for all the process that are currently present on the machine, this can be done using the command below:



Then, we have to wait for some 2-3 seconds to check if a counter process is successfully running or not. We can check this like below.

We simply created a user program which runs in background and increments counter on each timer interrupt. This helped us to keep track that a current process is really running on the machine.

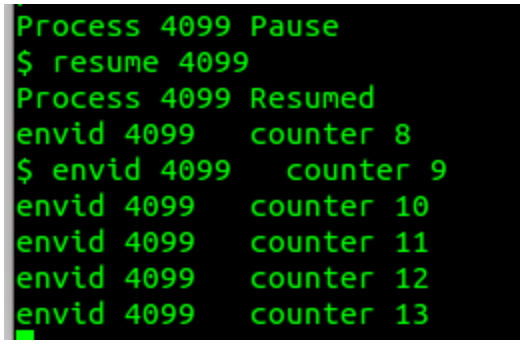


We now have to pause this process. As shown above, we can run a pause command with PID supplied as parameter, to pause a given process.

- `pause <PID>`

We can again resume the process using the resume command and supplying the PID into it. Please look at the below screenshot for clarity:

- `resume <PID>`



By the above two commands, we have tested that we were able to successfully pause and resume a state of process. As, while running resume the counter has started incrementing where it was stopped last time.

Next task was to save the process's state into an image and to successfully load into another newly created process. For this, we have implemented two user-level programs save.c and load.c.

save.c checks if the process provided as the parameter is currently paused or not. As, we have implemented preemptive process migration, it cannot save the state of a currently

running process. If the process is paused, then we will save the state into an image file supplied by user as argument.

- `save <PID> <image_file>`

```
Process 4099 Pause
$ save 4099 image
Saving Page : 0  at 0x00800000
Saving Page : 1  at 0x00801000
Saving Page : 2  at 0x00802000
Saving Page : 3  at 0x00803000
Saving Page : 4  at 0x00804000
Saving Page : 5  at 0x00805000
Saving Page : 6  at 0x00806000
Saving Page : 7  at 0x00807000
Saving Page : 8  at 0x00808000
Saving Page : 9  at 0x00809000
Saving Page : 10  at 0x0080a000
Saving Page : 11  at 0xef7fd000
Successfully saved 12 pages of env 4099 on disk to file image
$
```

As we can see all the page tables and pages inside an environment is successfully saved into the image file.

Next we will load the saved image file into the memory after creating a new process by using load command onto shell.

- `load <image_file>`

```
$ load image
Loading Page : 0  at 0x00800000
Loading Page : 1  at 0x00801000
Loading Page : 2  at 0x00802000
Loading Page : 3  at 0x00803000
Loading Page : 4  at 0x00804000
Loading Page : 5  at 0x00805000
Loading Page : 6  at 0x00806000
Loading Page : 7  at 0x00807000
Loading Page : 8  at 0x00808000
Loading Page : 9  at 0x00809000
Loading Page : 10  at 0x0080a000
Loading Page : 11  at 0xef7fd000
Load succeeded. New environment is : 4106
$
```

These two commands have helped us to test process migration on the same machine. We will next extend this functionality to successfully transport the state/image from one

machine to another machine in network using send and receive user-level programs. These two programs are responsible for establishing the connection between the two machines.

For the purpose of process migration, we have specified two ports. On these ports, receiver will listen for any incoming packets and try to receive all the packets that the sender will send. These packets are containing just the pages and page tables present in the image file. This will help us to rebuild the process state at the receiver end.

We first need to check that on which ports, the receiver will listen for any incoming requests. This we can do by running following command:

- `make which-ports`

```
root@vl166:/home/cse506/gupta1/phase2/vm2# make which-ports
Local port 26001 forwards to JOS port 7 (echo server)
Local port 26002 forwards to JOS port 80 (web server)
Local port 26003 forwards to JOS port 50001
Local port 26004 forwards to JOS port 50002
root@vl166:/home/cse506/gupta1/phase2/vm2#
```

We will use 50001 port for receiving the state of the process on vm2.

Now we will open two qemu terminals, one for running the receiver machine (mac2/) and another one is for sender machine (mac1/). We will run receiver machine first and type ‘receive’ command just to make it wait on 50001 port for listening all the requests.

- `receive`

```
$ receive
Waiting for connections on port 50001
Receiving Page : 0  at 0x00800000
Receiving Page : 1  at 0x00801000
Receiving Page : 2  at 0x00802000
Receiving Page : 3  at 0x00803000
Receiving Page : 4  at 0x00804000
Receiving Page : 5  at 0x00805000
Receiving Page : 6  at 0x00806000
Receiving Page : 7  at 0x00807000
Receiving Page : 8  at 0x00808000
Receiving Page : 9  at 0x00809000
Receiving Page : 10  at 0x0080a000
Receiving Page : 11  at 0xef7fd000
Process migrated. Resume : 4105
$ QEMU: Terminated
root@vl166:/home/cse506/gupta1/phase2/vm2#
```

Then we will run the sender machine (mac1/) on another qemu-terminal. We first need to pause the process and then run the following send command.

- `send <PID> <Receiver's IP> <Receiver's Port>`

```
$ send 4099 130.245.30.166 26003
Sending Page : 0  at 0x00800000
Sending Page : 1  at 0x00801000
Sending Page : 2  at 0x00802000
Sending Page : 3  at 0x00803000
Sending Page : 4  at 0x00804000
Sending Page : 5  at 0x00805000
Sending Page : 6  at 0x00806000
Sending Page : 7  at 0x00807000
Sending Page : 8  at 0x00808000
Sending Page : 9  at 0x00809000
Sending Page : 10  at 0x0080a000
Sending Page : 11  at 0xef7fd000
Sending successful
```

This will send all the pages and page tables from mac1/ to mac2/ over the network.

After receiving the process's state onto the machine mac2/, we need to check if it is resuming correctly or not.

```
Receiving Page : 10  at 0x0080a000
Receiving Page : 11  at 0xef7fd000
Process migrated. Resume : 4106
$ resume 4106
Process 4106 Resumed
envid 4106   counter 8
$
```

As we can see above, the timer starts ticking from the place where it was paused on mac1/. This told us that the saved process's state on mac1/ is successfully transported over the network onto mac2/.

Limitations

Our task is also limited to a single process (single thread) migration as there is no threading concept in JOS which would have increased the scope of project. Also, we have not saved the state of a process during a page fault.

References

http://en.wikipedia.org/wiki/OpenMosix

Process Migration - DEJAN S. MILO, JI CI C (HP Labs), FRED DOUGLIS (AT&T Labs–Research), YVES PAINDAVEINE (TOG Research Institute), RICHARD WHEELER (EMC) AND SONGNIAN ZHOU (University of Toronto and Platform Computing)

http://pdosnew.csail.mit.edu/6.828/2014/labs/lab7/ - Project Idea

PHASE 3

Disk encryption ensures that files are always stored on disk in an encrypted form. The files only become available to the operating system and applications in readable form while the system is running and unlocked by a trusted user. An unauthorized person looking at the disk contents directly, will only find garbled random-looking data instead of the actual files. It can also be used to add some security against unauthorized attempts to tamper with your operating system.

All disk encryption methods operate in such a way that even though the disk actually holds encrypted data, the operating system and applications "see" it as the corresponding normal readable data as long as the cryptographic container (i.e. the logical part of the disk that holds the encrypted data) has been "unlocked" and mounted.

Basic principle: - Each block device is divided into **blocks/sectors** of equal length. The encryption/decryption then happens on a per-sector/block basis. For JOS, we encrypt the each allocated block. Block device encryption methods operate *below* the file system layer and make sure that everything written to a certain block device is encrypted. This means that while the block device is offline, its whole content looks like a large blob of random data, with no way of determining what kind of file system and data it contains. For the purpose of encryption/decryption, we have used Advanced Encryption Standard (AES). This is the one of the most common algorithm for a secure and fast solution to disk encryption.

Methodology: - When the JOS file system is booted up for the first time, it checks whether the disk is encrypted or not. If the disk is not encrypted, then it asks for the password to encrypt it. Using this password, we use AES to encrypt a random block that stores the key for the AES algorithm. On the basis of the key stored in this random block, all other allocated blocks are encrypted.

The disk encryption process takes some time when the disk is encrypted for the first time. We have encrypted only allocated blocks to speed up this process. However, once this initial process is done, any next encryption/decryption of block takes negligible time and happens on the go.

The contents in each allocated block are stored in the form of cipher text. Whenever the system requires to read some part of the disk, then each requested block is decrypted on the go. If some free blocks are allocated, then the data is first encrypted and stored in the form of cipher text in these blocks. At any time the data stored in the file system is in the encrypted form.

When the JOS is booted for the second time, it asks for the password for decrypt the disk. Without the correct password, the JOS will fail to boot up.

Implementation: - The code implementation is mainly in the files `fs.c` and `bc.c`.

Test case: - The code for the encryption part is in the encryption folder. When the user boots JOS for the first time, he is asked to supply a password to encrypt the disk. On any

other run, he is required to provide this password to successfully boot up the JOS. We ran the `testfile` test case and it was successfully passed.

- `make run-testfile-nox`

However as we cannot see the contents of the JOS file system anywhere on our disk, we cannot directly see whether the disk is encrypted or not. So we were not able to create a test case. But we can check the contents on the disk by printing out them just before the decryption steps occurs.

For that you need to uncomment the

```
#define SET (line 3)
```

in the fs/fs.c file. Then run testfilewrite in shell to create a new file named “newfile”. Now shut down the JOS. Now again boot up the JOS. It will first ask for the password. Then do the cat newfile in shell.

- `make run-icode-nox`
- `run the shell`
- `testfilewrite`

Shut down the JOS and again boot up and perform the below instruction in the shell

```
cat newfile
```

Here we are printing the 32 characters of the plain text and the cipher text. We can observe that after the block read and before the decryption, the content is in cipher text and after the decryption, it is in human readable form.

This shows that the data stored in the file system is in the form of cipher text.

Limitations: - Out of the all 10240 blocks in the JOS file system, we were unable to encrypt the zeroth and first (superblock) block. We tried to encrypt the superblock but were getting some random page faults. Due to the shortage of time we were unable to look deep into this task.

References: -

http://www.dfcd.net/projects/encryption/aes.h - AES code

https://wiki.archlinux.org/index.php/disk_encryption