COP 6611: Operating Systems

Project 3 - Copy-On-Write Fork

Due Date: 04/24/2019

Objectives

This assignment will require you to put together all the components of previous assignments to implement the fork system call.

All work should be done individually.

Assignment Setup

If needed, review the previous assignment readings and code.

Code Checkout

Please follow these instructions to fetch the code and merge your solution from project 2. If you

First, go into your working directory i.e. your project directory from project2.

```
$ cd [PATH TO YOUR PROJECT DIRECTORY]
```

From here on, all the instructions will be run in your project directory.

Fetch from remote upstream repository:

```
$ git fetch upstream
```

Checkout branch for project3:

```
$ git checkout -b project3 upstream/project3
$ git config --unset branch.project3.remote
```

This will checkout the starter code from the upstream repository and remove the upstream link so that you can work in your own repository after fetching the code.

Merge your solution:

```
$ git merge project2
```

Testing The Kernel

The only test provided is the `startfork` command on the monitor. You may choose to write your own test cases for your own implementations. You may add additional test cases in any of the existing layers.

If you are implementing any additional layers, you can include a `test.c` file with your test cases. Use the `test.c` files in the other layers as an example. You will also have to include these tests in the `kern/init/init.c` file. Then you can run the following instructions as before.

```
$ make clean
$ make TEST=1 [SOL=1]
```

While writing your code, use these commands to run unit tests to verify the working status of your code. We have left the test cases from project2. The changes you make for this assignment should not impact these test cases.

```
Testing the PKCtxNew layer...

test 1 passed.

All tests passed.


Testing the PTCBInit layer...

test 1 passed.

All tests passed.


Testing the PTQueueInit layer...

test 1 passed.

test 2 passed.

All tests passed.


Testing the PThread layer...

test 1 passed.

All tests passed.


Test complete. Please Use Ctrl-a x to exit qemu.
```

If you don't see these tests, make sure you have the correct bootloader code, virtual memory code and process management code from project 0, project 1 and project 2!

Normal Execution

Building the image:

```
$ make clean
$ make [SOL=1]
```

These two commands will do all the steps required to build an image file. The make clean step will delete all the compiled binaries from past runs. After the image has been created, use one of the following execution modes:

Executing WITH QEMU VGA Monitor

```
$ make qemu
```

You can use `Ctrl-a x` to exit from qemu.

Executing WITHOUT QEMU VGA Monitor

```
$ make qemu-nox
```

Executing with GDB and QEMU VGA Monitor

```
$ make qemu-gdb
```

IMPORTANT NOTE: The solutions we have provided is for the complete project2. But as you will read below, you may want to modify the existing code from project 2. If you choose to do so and still want to use the solutions, then the file you chose to edit has to be completed by you. i.e. if you have existing bugs in that file, you will have to fix it as well.

In order to use the solutions but exclude certain files, you will have to change the corresponding makefile. We have provided some sample makefiles which will not use the solutions in PThread layer. You can rename the file called `Makefile.inc.no-sol` to `Makefile.inc` to stop using the solution for that layer. For other layers, you can make similar changes in the corresponding makefile.

Running User Processes

We have implemented a new monitor task **startfork** in addition to **startuser** in the last assignment. Once run, it starts the forker process implemented in `user/fork/forker.c`, which in turn spawns a new user process defined in `user/fork/fork.c`. It will then fork a new process and the resulting child will fork another process. The **startfork** command will test both `sys_spawn` and `sys_fork` system calls.

User Processes

We have implemented some simple procedures in the fork process so that you can see that each forked process is isolated from one another. We also update a variable called `global_test`, which will test for Copy-On-Write functionality. After you can successfully run and understand the user processes we provided, you can replace it with more sophisticated user process implementations. A sample output for `startuser` is shown below.

```
$> startuser
[D] kern/lib/monitor.c:67: process idle 1 is created.
Start user-space ...
idle
ping in process 4.
pong in process 5.
ding in process 6.
ping started.
ping: the value at address e0000000: 0
ping: writing the value 100 to the address e0000000
pong started.
pong: the value at address e0000000: 0
pong: writing the value 200 to the address e0000000
ding started.
ding: the value at address e0000000: 0
ding: writing the value 300 to the address e0000000
ping: the new value at address e0000000: 100
pong: the new value at address e0000000: 200
ding: the new value at address e0000000: 300
```

A sample output for `startfork` is shown below.

```
$> help
help - Display this list of commands
kerninfo - Display information about the kernel
startuser - Start the user idle process
startfork - Start the user fork process
$> startfork
[D] kern/lib/monitor.c:50: process forker 1 is created.
Start user-space ...
forker
fork in process 4.
parent forks 13, global = 0x12345678
parent global_test1 = 0x5678
Child forks 40, global = 0x1234
This is grandchild, global = 0x1234
```

Assignment Description

For this assignment you will implement copy-on-write fork. We will provide a software specification, and you will need to determine the best way to implement the spec. To get the required functionality you will need to debug parts of past assignments.

POSIX systems' way to create new processes is using the fork system call. Fork will duplicate the currently executing process copying all its memory state, then return 0 in the child and return the child's PID in the parent process. Copying the whole memory state of a running process might be prohibitively expensive. So instead of doing this, modern POSIX systems implement a copy-on-write mechanism.

Copy On Write

The copy-on-write (COW) idea is to map the same pages in the two processes, but when a process tries to write to some memory shared with another process, the MMU is instructed to trap into the kernel, the kernel then allocates a fresh page, copies the contents that were about to be (partially) over-written and updates the memory mapping in the page directory to use the fresh page. The fresh page is set to be writeable and the write proceeds to execute normally. This whole mechanism makes the copy of pages a "lazy" procedure.

Technically, when a process is forked and it's address space is cloned, the page table entries of this process and the ones of the newly created child are all marked read-only and have the PTE_COW bit set. The trap handler then needs to differentiate invalid memory accesses from a copy-on-write access and take appropriate action.

Page Sharing and Ownership

Because the some pages may be shared between different processes, the notion of ownership is now more complicated. For example, assume two processes A and B share a page, if A tries to write to the page, the COW mechanism will kick in and give it a new page; the original shared page now only belongs to B. If B now tries to write to the page (still marked read-only COW in its page table), then the COW logic described above would work too. However, it is slightly inefficient because now, only B owns the page, ideally, we should take advantage of this by unsetting the COW bit and allow the write to proceed. Modern operating systems use a reference counting mechanism to implement this optimization. Each physical page has a number associated that stores how many processes have a mapping to that page.

Code Guidelines

This project’s specification will be more open-ended than previous assignments so that you can demonstrate that you have fully understood the code base we have developed.

Exercise 1: The MPTCow Layer

The first exercise is to extend the existing layer system with a new `vmm` layer. To do this you will need to replicate the structure you have seen for other layers. (Note that we have not provided you with any boilerplate and you will have to add this layer yourself.)

To add a new layer we need to include it in build system. The `make` build system starts by reading the Makefile in the src folder. We can see that this file includes `kern/Makefile.inc`

```
# Code segment from src/Makefile ...

# Sub-makefiles
include boot/Makefile.inc
include user/Makefile.inc
include kern/Makefile.inc
```

This file which in turn includes the `Makefile.inc` in `vmm`, `pmm`, and so on:

```
# Code segment from src/kern/Makefile.inc ...

# Sub-makefiles
include          $(KERN_DIR)/config.mk
include          $(KERN_DIR)/dev/Makefile.inc
include          $(KERN_DIR)/lib/Makefile.inc
include          $(KERN_DIR)/init/Makefile.inc
include          $(KERN_DIR)/pmm/Makefile.inc
include          $(KERN_DIR)/vmm/Makefile.inc
include          $(KERN_DIR)/thread/Makefile.inc
include          $(KERN_DIR)/proc/Makefile.inc
include          $(KERN_DIR)/trap/Makefile.inc
```

We can see that `kern/vmm/Makefile.inc` includes the `Makefile.inc` for each layer:

```
# Code segment from src/kern/vmm/Makefile.inc ...

# -*-Makefile-*-


include   $(KERN_DIR)/vmm/MPTIntro/Makefile.inc
include   $(KERN_DIR)/vmm/MPTOp/Makefile.inc
# ... You will need to add you own layer in this file.
```

Use the `Makefile.inc` in the other `vmm` layers as an example for what to use in your new layer. Additionally look at the other layers as an example of how to write the `import.h` and `export.h` files. Finally in the code for your new layer you need to export two new functions.

In the file `kern/vmm/MPTCow/MPTCow.c`, you must implement the function listed below:

- `map_cow`
- `map_decow`

The `map_cow` function should copy the page directory and page tables of one process to another. Be careful to adjust the permissions correctly (by setting the `PTE_COW` bit and marking it read only) and copy only what is necessary (user space mappings). You can use functions from `MPTIntro` or add new ones to suit your needs. Hints:

1. Only user space mapping has to be copied.
2. Page table entries in both processes should be marked as COW.

The map_decow function will allocate a new page, copy it, and remove the COW permissions for the corresponding vadr. This is the function that will be called once the kernel traps into due to a COW interrupt.

Use the following API:

```c
void map_cow(uint32_t from_pid, uint32_t to_pid)
void map_decow(uint32_t pid, uint32_t vadr);
```

Exercise 2: The PProc Layer

A useful function to create would be proc_fork, a function similar to proc_create that copies the current process. Allocate half of the quota size remaining to the child. Be really careful when you setup the user context of the child (think of the return value of fork in that context). Hint: that it might be useful to write a new function called thread_fork, which is similar to thread_spawn.

Exercise 3: The TTrapHandler Layer

Modify the page-fault handler function to handle page faults due to COW traps. If the error number given by the trap is "3" then the page fault was due to a COW trap.

Exercise 4: The TSyscall Layer

In kern/trap/TSyscall/TSyscall.c, fill in the sys_fork() function. This system call has no arguments, it forks the current process and returns the child process id in the parent. The newly created process will start in the exact same context as the parent except that 0 is returned by the system call. This function shouldn't do any work directly, instead calling proc_fork.

Exercise 5: The TDispatch Layer

Add your new system call to the syscall dispatcher under SYS_fork.

Exercise 6: User Library

Add your new system call to the user library..

Extra Credit (+50%):

Add the reference counting optimization described above.

Hand in Procedure

You will submit your assignment via git by creating a tag. The following instructions will explain the needed steps. To get started first change into the root directory of your repository.

Git: Adding Changes AND Git: Committing Changes

These steps are same are same as the previous project. Please `git add` and `git commit` all your changes.

Git: Push Branch: FIRST TIME ONLY

This is a checkpointing step and submits your local branch and commit history to the remote git server. Doing this means your changes will not be lost.
First time push for the new project 3 branch:

```
$ git push -u origin project3
```

The additional -u flag has to be used for the first time you push the branch. This sets the proper remote tracking branch.
Subsequent push:

```
$ git push origin project3
```

Submitting: Creating a tag

```
$ git tag -a v3 -m "Submission for project 3."
```

This will create a `tag` named `v3` from the latest commit on your local.

Submitting: Pushing the tag

```
$ git push origin v3
```

This will push the `tag` named `v3` to the remote repository. Note that this step is different from “Git: Push Branch” step above.

Re-Submission

If you make changes after creating the tag, you will have to forcefully update the tag and (if already pushed) forcefully push the tag.

Follow steps for Adding Changes, Committing Changes and Push Branch as above.

Re-Submitting: Updating the tag

```
$ git tag -a -f v3 -m "Re-Submission for project 3."
```

This will forcefully update the existing tag named `v3` from the latest commit on your local.

Re-Submitting: Forcefully Pushing the tag

```
$ git push -f origin v3
```

This will forcefully push the `tag` named `v3` to the remote repository.