Project 3: s2dsm (Super Simple Distributed Shared Memory)

- **Handed out:** Thursday, September 29, 2022
- **Due dates**
  - Part 1: Wednesday, October 12, 2022
  - Part 2: Friday, October 21, 2022

Introduction

The goal of this project is to implement `s2dsm`, a distributed shared memory (user-level) program in which two `s2dsm` processes share a set of pages in a coherent manner. For coherence, `s2dsm` supports a page-granule MSI (coherence) protocol, using `userfaultfd`. Two `s2dsm` processes will communicate their page information using sockets. Though `s2dsm` may run on different machines to support a (truely) distributed shared memory, in this project we will run two `s2dsm` processes on a single machine for simplicity. This project consists of two parts.

Recommended Background Reading

- Socket Programming in C/C++
- POSIX thread (pthread) libraries
- userfaultfd
- MSI protocol

=== Part 1 ===

Part 1a: Understanding userfaultfd demo source code and output

[10 points] The attached source code `uffd.tar.gz` is a simple application that demonstrates how `userfaultfd` can be used to handle page faults at the user level. It contains two files shown below.

```
$  tar xzvf uffd.tar.gz uffd/
uffd/
uffd/Makefile
uffd/uffd.c
```

Carefully read, compile and execute uffd.c file to understand how userfaultfd can be used to manage paging. Once you completely understand uffd.c, add comments of each line (H1-H10, M1-M7, U1-U7) explaining what the line means. For U1-U7, explain the output of `./uffd 1`.

If you see the error message "userfaultfd Function not implemented", then you should recompile your kernel with `CONFIG_USERFAULTFD=y`.

Create a gzip-ed tarball named `uffd.tar.gz`.

```
$  tar czvf uffd.tar.gz uffd/
uffd/
uffd/Makefile
uffd/uffd.c
```

Part 1b: Pairing memorey regions between two processes

[20 points] Create a (user-land) application named `s2dsm` that takes as input a local port number and a remote port number. Two instances of `s2dsm` communicates each other over sockets. For example, when running `./s2dsm 5000 6000` and `./s2dsm 6000 5000`, two processes pair up each other. The first `s2dsm` process listens on the port 5000 and send messages to the port 6000. The second process listens on the port 6000 and send messages to the port 5000.

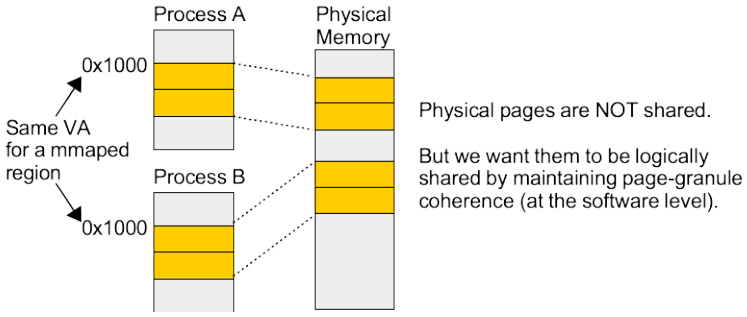
After pairing, the first `s2dsm` process:

- asks a user to specify the number of pages to allocate through stdin: e.g., " > How many pages would you like to allocate (greater than 0)?"
- mmaps a memory region of the size specified by `<the number of pages> * PAGESIZE`;
- printfs the address of the mmaped region (the return value of mmap) and the mmapped size; and
- sends a message including the mmaped address and size to the second process over a socket communication.

The second `s2dsm` process:

- receives the message including the mmapped address and size from the first process;
- mmaps a memory region of the same size to the same address (by specifying the first argument of mmap); and
- printfs the address of the mmaped region and the mmapped size.

The following figure illustrates that two `s2dsm` processes allocate a mmaped memory region at the same virtual adress, yet the physical pages are not shared. In Part 2, we will make them logically shared by maintaining data coherence at the page granularity.



Program your code in `s2dsm_P1b.c` and create `Makefile`.

Take a screenshot of the printfs for memory address and size for mmaped region of two paired `s2dsm` processes. To do this, you must use `tmux` to split the terminal vertically, left-side running the first process, followed by the right side running the second process. The first process will ask the user to input a number of pages it should mmap. It will then mmap first, and communicate the mmap information to the second process. You must show to printf of mmap information from both processes. Name your screenshot as `s2dsm_P1b.png`.

Part 1c: Implement `userfaultfd`

[20 points] Picking up from Part 1.2, once you have a mmapped memory region, register the memory region with `userfaultfd`. Once paired, `s2dsm` should repeatly ask a user for two inputs:

- " > Which command should I run? (r:read, w:write): "
- " > For which page? (0-%i, or -1 for all): "

You will have to replace "%i" with the the number of pages specified by the user from Part 1b.

The r (read) command reads the contents in the specified (or all) pages. When a page fault occurs (e.g., for the first access to a page), printf `" [x] PAGEFAULT\n"`. For each page, printf the contents as follows: `" [*] Page %i:\n%s\n"` where `%i` is the page number and `%s` is the page content.

The w (write) command asks a user for the content to write on the specified (or all) pages:

- " > Type your new message: "

Write the new message into the specified (or all) page. Similar to the r command, when a page fault occurs, printf: `" [x] PAGEFAULT\n"`. Printf the (updated) contents as follows: `" [*] Page %i:\n%s\n"` where `%i` is the page number and `%s` is the page content.

Program your code in `s2dsm_P1c.c` and create `Makefile`.

Take a screenshot of the outputs (printfs) when following the operation sequence below:

- Pair the two processes with 3 pages [0-2]
- Process 1 reads page 0
- Process 1 reads all pages
- Process 1 writes "Hello Userfaultfd!" to page 2
- Process 1 reads all pages
- Process 1 writes "Writing this to all pages!" to all pages
- Process 1 reads all pages

You only need to show the outputs of one process. Name your screenshot as `s2dsm_P1c.png`.

Submission

Make a folder named with your SBU ID (e.g., 112233445).

Put `uffd.tar.gz`, `Makefile`, `s2dsm_P1b.c`, `s2dsm_P1c.c`, `s2dsm_P1b.png` and `s2dsm_P1c.png` in the folder.

Create a single gzip-ed tarball named `[SBU ID].tar.gz`, and turn the gzip-ed tarball to Brightspace.

```
$  tar czvf 112233445.tar.gz 112233445/
112233445/
112233445/Makefile
112233445/s2dsm_P1b.c
112233445/s2dsm_P1c.c
112233445/s2dsm_P1b.png
112233445/s2dsm_P1c.png
112233445/uffd.tar.gz
```

=== Part 2 ===

Part 2: Implement MSI page coherence protocol

[50 points] Picking up from Part 1c, you will implement the MSI protocol to ensure the coherence of a shared memory region between two processes. To do this, each process maintains an array, called `msi_array` that keeps track of MSI information (enum values: M, S, and I) for each page.

Once paired, `s2dsm` should repeatly ask a user for two inputs:

- " > Which command should I run? (r:read, w:write, v:view msi array): "
- " > For which page? (0-%i, or -1 for all): "

You will have to replace "%i" with the the number of pages specified by the user from Part 1. The new "v" command will allow the user to view the contents of the MSI array.

For the "r"ead and "w"rite commands, implement the MSI protocol as specified in MSI protocol Wikipedia. Note that the original MSI protocol is introduced to ensure cache coherence. In this project, we adopt its invalidation-based coherence mechanism to enable distributed shared memory at the page granularity.

Initially, all pages are in the invalid state (I). We will further assume the following initial read, invalidation and concurrency protocols.

- **Initial read protocol:**
    - Given two processes A and B and page number 1:
    - Initially, A1 (A's page 1) and B1 (B's page 1) are INVALID.
    - Read (A1) will trigger a page fault (as this is the first access).
    - A detects that A1 is in the INVALID state.
    - A attempts to fetch B1 from B.
    - B informs A that B1 is also in the INVALID state (if B1 is not INVALID, then B simply sends back the page and updates the state accordingly).
    - When A finds that B1 is also INVALID, then it printf null string: e.g., printf("[*] Page %d:n%sn",1,"");
    - A and B both move A1 and B1 to the SHARED state. (The shared state means that a "clean" copy is shared by all processes. If one process's page is in the Shared state, then the other processes' corresponding page should have the same Shared state.)
    - if B reads B1, it will trigger a page fault. (But it does not need to communicate with A because B knows A/B both have "clean" Shared (empty) copies.)
- **Invaidation protocol:**
    - Given two processes A and B and page number 1:
    - When B receives the INVALIDATE request for page 1 from A (as A writes on A1), B should make B1 INVALID and call `madvise(B1, PAGE_SIZE, MADV_DONTNEED)`.
    - This will make any following access to B1 a page fault.
    - If B1 is in the INVALID state, B should start handling Read (B1) or Write (B1) from the page fault: e.g., Read(B1) attempts to fetch A1.
    - This will allow us to have an unified read fetch logic for an initial read and a read from a invalidated page.
- **Concurrency protocol:**

- Assume there are no concurrent operations: B takes some actions after A completes its operation (and MSI protocol handling).

Run two processes as follows and take a screenshot (`s2dsm_P2.png`) of the outputs (printfs) from both processes. You may take multiple screenshots.

- Pair the two processes with 3 pages [0-2]
- Process 1 views all MSI contents
- Process 2 views all MSI contents
- Process 1 reads page 0
- Process 1 views all MSI contents
- Process 2 views all MSI contents
- Process 1 reads all pages
- Process 1 views all MSI contents
- Process 2 views all MSI contents
- Process 2 writes "Hello Userfaultfd!" to page 2
- Process 1 views all MSI contents
- Process 2 views all MSI contents
- Process 1 wites "Writing this to all pages!" to all pages
- Process 1 views all MSI contents
- Process 2 views all MSI contents
- Process 1 reads all pages
- Process 1 views all MSI contents
- Process 2 views all MSI contents

Submission

Make a folder named with your SBU ID (e.g., 112233445), put `Makefile`, `s2dsm_P2.c`, and screenshot `s2dsm_P2.png` files in the folder, create a single gzip-ed tarball named `[SBU ID].tar.gz`, and turn the gzip-ed tarball to Brightspace.

```
$  tar czvf 112233445.tar.gz 112233445/
112233445/
112233445/Makefile
112233445/s2dsm_P2.c
112233445/s2dsm_P2.png
```