CS240 ASSIGNMENT 3:

KAUST Shared Memory (KSM)

NAME: XIAOPENG XU KAUST ID: 129052

Abstract:

In this assignment I added shared memory functionality to the original xv6 code. 5 system calls for KAUST shared memory (KSM) operations were implemented to provide an interface for processes to use KSM. Original process related functions such as `fork`, `exit`, `wait` and `exec` were modified for KSM.

Design & Implementation:

Below is the design decisions I made:

1. KSM structure contains information as follows:
```
volatile int khd; // KSM handle, initiated with 0
char* kname;      // KSM name
int flag;         // Flag bits: KSM_CREATE, KSM_READ, KSM_WRITE, KSM_RDWR, and KSM_DELETE.
uint memlist[KSM_PG_MAX];   // Physical pages allocated for KSM
uint ksmsz;       // The size of the shared memory
int cpid;         // PID of the creator
int mpid;         // PID of last modifier
int attached_nr;  // Number of attached processes
uint atime;       // Last attach time
uint dtime;       // Last detach time
```
   This KSM structure, along with several macros was defined in *ksm.h* file.
2. KSMs are managed by OS through a `ksmtable`. `Ksmtable` contains a `lock`, an array of `KSMs`, and `ksmglobalinfo`. This `ksmtable` struct was implemented in *vm.c* file. The `lock` is to make sure that each time only one process gains access to `KSMs` array of the systems. The KSMs arrary contains all KSMs in OS. The maximum number of KSMs is defined by `KSM_SEG_MAX`. `Ksmglobalinfo` is update through all KSM operations. The values in `ksmglobalinfo` are used in `ksminfo` to return global KSM information. (One thing needs to be cautious is that the size of `ksmtable` should not be too big, otherwise, it will overwhelm the 4M memory in first memory mapping.)
3. Each process maintains the information (the handle and virtual address) of KSMs attached through two arrays (`proc->ksmhd[]` and `proc->ksmaddr[]`). In my design, a KSM can be attached in a process multiple times, but there's a limit for how many KSMs a process can have, i.e. `PROC_KSM_MAX`. These modification are in file *proc.h*.
4. KSMs are mapped to from the top of user virtual address space. It starts mapping from `KERNBASE`, and ends until meet with `proc->sz`. A simple first fit algorithm was implemented to find a virtual address range for KSM. This might cause the fragmentation of virtual addresses if KSMs are not got and deleted as a

stack. This functionality is achieved through `ksmattach()`, `ksmdetach()` and `detachksm()` in file *vm.c*.

5. Several system calls are implemented for KSM operations. These system calls includes `ksmget()`, `ksmdelete()`, `ksmattach()`, `ksmdetach()` and `ksminfo()`.
    a. `Ksmget()` returns the handle of KSM if KSM is existed or created. Frist it searches through all the KSMs with the same name. If the KSM name does not match any KSM, it creates a new KSM. If all KSMs in OS is occupied and no free KSM found, it returns -1.
    b. `Ksmattach()` attaches a KSM to current process. The flag was considered while mapping process virtual memory with physical KSM memory. After mapping, the `proc->ksmbd` is updated if the mapped virtual address is lower than `proc->ksmbd`. It returns the mapped virtual address of attached KSM if succeeded. Otherwise, if maximum number of KSM in process is reached or there's no KSM with the given `ksmhd`, it returns -1.
    c. `Ksmdetach()` removes all attachments of the KSM from current process. If the `attached_nr` of KSM is 0 and KSM is marked as `KSM_DELETE`, the KSM will be deleted. Also `proc->ksmbd` is changed if free virtual spaces above `proc->ksmbd` are released.
    d. `Ksminfo`, get local and global KSM information of `ksmhd` or just global KSM information if `ksmhd` equals to 0. It reads the KSM local information through KSM struct and global information through `ksmglobalinfo` in `ksmtable`.
    e. `Ksmdelete()` marks a KSM to delete it. If a KSM is not marked as `KSM_DELETE`, it can still be there even if no process is using it.

    The files modified for these system calls include *defs.h*, *syscall.c*, *syscall.h*, *sysproc.c*, *user.h*, *vm.c*, and *usys.S*.
6. `Pgused` system call was implemented for testing purpose. It reports the number of physical pages currently used. The files modified include *syscall.c*, *syscall.h*, *sysproc.c*, *user.h*, and *usys.S*.
7. In file *proc.c and exec.c*, `fork`, `exit`, `wait`, and `exec` are modified in order to handle KSM.
    a. While `fork`, a child process copies the KSMs from parent. The data copied includes `proc->ksmbd`, `proc->ksmhd[]`, and `proc->ksmaddr[]`.
    b. While `exit` and `exec`, all KSM in process are detached through calling `detachksm`.
    c. `Wait` remains the same. But the parent process clears KSM virtual pages of child process during `wait`.
8. Also several functions are implemented in file *vm.c* to support the implementation of above functionalities. These functions include: `findksm`, `deleteksm`, `attachksm`, and `detachksm`.
    a. `Findksm` function searches for a KSM in `ksmtable` with KSM handle `ksmhd`.

   b. `Deleteksm` function delete KSM if `KSM_DELETE` is set and `attached_nr` equal to 0.
   c. `Attachksm` attach `KSM` with KSM handle `ksmhd`, it only change the KSM part. `Attachksm` was called in `fork` and `ksmattach`.
   d. `Detachksm` detach all KSMs within proc.
9. Also I modified `copyuvm` in file *vm.c* to copy not only process data and heap, which is from virtual address `0` to `proc->sz`, but also KSM virtual spaces, which is from virtual address `proc->ksmbd` to `KERNBASE`.
10. `Sbrk` was modified in *sysproc.c* to avoid `sbrk` a new memory, which overlaps KSM region.
11. Write a *ksmtest.c* file to test above KSM operations. The operations I tested include:
   a. Basic KSM operations such as `ksmget`, `ksmattach`, `ksmdetach` and `ksminfo` are first tested in a single process.
   b. `Ksmdelete`, `fork`, `exit`, and `wait` are tested with two processes using one KSM. Writing into and reading from KSMs are tested by two processes communicating through the KSM.
   c. Finally, `sbrk` and `ksmattach` was tested to make sure that KSMs and process data do not overlap, that is `ksmattach` does not attach below `proc->sz` and `sbrk` does not get virtual memory beyond the `proc->ksmbd`.

Results:

In my experiment, `ksmtest` and `usertests` are successfully passed.