Prof. Hany Ramadan KAUST - CS 240 - Computing Systems & Concurrency

Project : Kaust Shared Memory (KSM)

In this project, you will add shared memory functionality to the xv6 code base.
Processes today can only communicate with pipes or through the file system, but we would like to have a communication mechanism that doesn't involve the kernel each time we want to communicate. Shared memory is such a mechanism - and the design below is loosely modeled on a real shared memory interface (SysV IPC) - though with important simplifications and differences in semantics (stick to what is written below and what we reach in our class discussions). Your new system calls will be put to use by a "Math Server" application later in the semester that uses shared memory to communicate between a client and server application (you will be provided with a copy of this application source code). Your report is allowed to exceed the standard 1-2 pages. If you want, you can use up to 4-5 pages.

System Call Interface

```
handle = ksmget (identifier, size, {flags} )
address = ksmattach (handle, {flags} )
ksmdetach (handle)
ksmdelete (handle)
ksminfo (handle , &ksm_info)
int pgused ()
```

ksm_info structure

- size of segment
- PID of creator
- PID of last operation
- nattch (number of current attachments)
- time of last attach
- time of last detach
- time of last change
- total number of currently existing segments
- total number of shared memory pages
- total number of attachments (?)

General Semantics

Processes that want to communicate with shared memory will all agree on an identifier. Then they will each call ksmget() with the same identifier, to receive 'handles'. The handles are opaque to user processes, whether they are the same or not does not matter - what matters is that they ultimately refer to the same underlying memory

region. As soon as a region actually exists (i.e. after the first get), the processes can call ksmattach with that handle, to map the shared memory into their address spaces. Once the memory is mapped, the processes can access it concurrently. Processes can detach regions as well (say, to free up space in their virtual address space, if they need it for something else). Regions are deleted by detaching them, and through the use of the ksmdelete system call (see below for details).
At any point in time, the ksminfo system call can return information about a valid (existing) region, by passing in the handle. The information is returned by coping various statistics into a user-supplied memory region (which is of type struct ksm_info) Passing in a 0 handle returns only total information for the entire system.
The pgused() system call can be called at any time to see how much physical memory the kernel is using at any point in time.

Design Discussion

- In this project you should add a pgused() system call, that returns the number of physical pages allocated inside the kernel. Make sure you are using the regular kernel allocator (not the one you wrote in the previous coding project). This system call will be used to test that your KSM project is working correctly (e.g. not leaking any memory, etc.)

- Can a process attach a segment multiple times ? In which case, ksmdetach should not take a handle, but the actual address of one of the mapping (the one desired to be detached).

- Where in the Process Address Space will implementations map shared memory ? (Each student - if they want - can do their own design here.)

Do we pretend it is like an sbrk - and modify sz - (but what then happens if one calls sbrk to then decrease memory size?) Or do we work from the top of the user address space on down, and proc->sz doesn't change (this is what we decided on). Or do we use a special x86 segment for shared memory mappings, and assume that user-mode code will use the appropriate selectors ? Note that if we do allow multiple attachments, it will be pretty easy to test for exhaustion (even though we only support 256 MB of real memory), since 2GB can be easily exhausted with multiple attach calls. Your code needs to check that sbrk and the limits of attachments never overlap, and fail the appropriate calls in that case. You will also need to have a methodology for allocating memory (e.g. do you ignore holes?), esp. since there can be a mixture of attach and detach calls for regions of different sizes. In all cases, you must not break sbrk's functionality.

- Do you allocate the underlying physical memory when a region is created in the first get, or upon it's first attachment ? Minor point, it is up to you. But make sure the rest of your code can keep track of the state correctly.

- What happens on process exit / fork / exec ? We decided that on exit and exec, we will automatically detach. And on fork, we will inherit the attachments.

- Data type of the identifier (string, integer, etc.). We settled on integer.

- Data type of handle (integer).

- We discussed whether to add an optional "flags"parameter to ksmget and/or ksmattach. Some uses for that flag, which we discussed, were to indicate in ksmget if you expect the ksm region to already exist or not. The default behavior is "find or create". Another use for the flag to ksmattach is to support "read-only" and "read-write" selectable access when mapping a page. It seems like this would be a fairly easy and useful addition. We therefore decided to have the flags in ksmattach, but not in ksmget.

- Whether to allow the process to specify a desired 'target address' in ksmattach, or always have the system decide. (Agreed on the latter, to simplify).

- Discussed how segments get deleted. Discussed having a 'persistent' segment creation flag (where the segment can outlive any active processes), or have it reference counted. We decided to go with the latter. But even then, a created segment that no one has mapped, is not actually being used -- so we added a ksmdelete system call, to "mark" the segment for deletion the next time that there are no more attachments. An interesting question is what happens when the segment is *not* marked for deletion, and all processes that were using it exit, does the segment still stay around (i.e. is essentially persistent by default ?)

- What happens if someone attaches after a segment has been marked for deletion, do we reject that attempt, or let it through ? (We went with the latter)

- What are the cases where these functions fail, and what do they return ? (e.g. when trying to attach with an invalid handle). We need to define error codes (e.g. different negative handle values?)

- Is size (in ksmget) rounded up or down to page sizes, or does it have to be a multiple of page size already ?

- Your implementation should perform proper error checking on all user parameters coming in from user-mode, especially pointers and what they point to. Generally speaking, you must not introduce new vulnerabilities into the kernel with your implementation.

- If a handle of 0 is passed to ksminfo() then only the last few fields (the ones with 'totals') are populated in the structure.

Implementation hints

- Notice that existing kernel code can be re-used to allocate physical pages (e.g. the kernel allocator) that will provide the "contents" of the shared memory, as well as the virtual address spaces whenever there

is a mapping. The existing kernel code often allocates both of these together (e.g. allocuvm does both things) , but conceptually you will need to have these occur at separate times.
- Memory for a process gets freed when the process dies (e.g. freevm, after a fork/exec, or on exit when a ZOMBIE is found in wait()). The virtual addresses are known based on proc->sz, and physical pages are freed as well (with kfree). Again, shared memory will need to be done differently. First, the virtual pages (depending on where you put them in the address space), can't just be found based on "sz", since it doesn't even reflect them. Second, the physical pages of the shared memory aren't freed, until the entire shared region gets freed (i.e. no more attachments after a call to ksmdelete). Note that files are freed differently (e.g. exit() frees files, not waiting for cleanup in wait)

Other Simplifications
- This project ignores permissions (since xv6 doesn't have a concept of user identities), and will instead depend on identifier to provide some "security"/"isolation" between applications.
- SuperPages (a.k.a. HUGETLB / 4 MB pages). We discussed whether to do this automatically, or as an optional flag. We decided to simplify (and re-use existing xv6 code), we won't support huge pages in this project.

Historical Note
The interface for our KSM design is loosely modeled on the System V IPC shared memory interface. However, our design is simpler, and we deviate from the semantics significantly, so do not base your project on the System V, but rather on the specification as described here and elaborated on in our in-class discussions.
You can find out more information about the System V ipc mechanisms from the svipc man page. Shared memory is covered in more depth in the following three man pages: shmget, {shmop,shmat,shmdt}, shmctl.

Proposed Header File (supplied by student)
```
// KAUST SHARED MEMORY
#define KSM_CREATE         0x01
#define KSM_OPEN           0x02
#define KSM_READ           0x04
#define KSM_WRITE          0x08
#define KSM_RDWR           0x10
#define KSM_MUSTNOTEXIST   0x20
#define KSM_HUGETLB        0x40

// KSM global info structure
struct ksmglobalinfo_t {
```

```
  uint total_shrg_nr;    // Total number of existing shared
regions
  uint total_shpg_nr;    // Total number of existing shared
pages
};

// KSM info structure
struct ksminfo_t {
  uint ksmsz;          // The size of the shared memory
  int cpid;            // PID of the creator
  int mpid;            // PID of last modifier
  uint attached_nr;    // Number of attached processes
  uint atime;          // Last attach time
  uint dtime;          // Last detach time
  struct ksmglobalinfo_t* ksm_global_info; // Global ksm
information
};

// Put the following functions into defs.h
int             ksmget(char* name, uint size, int flag);
int             ksmattach(int hd, int flag);
void*           ksmdetach(int hd);
int             ksminfo(int hd, struct ksminfo_t* info);
int             ksmdelete(int hd);
```