KING ABDULLAH UNIVERSITY OF SCIENCE AND TECHNOLOGY
COMPUTER SCIENCE DEPARTMENT

CS240 - Computing Systems and Concurrency - Project

KAUST Shared Memory (KSM)

Maruan Al-Shedivat (ID: 129096)

November 18, 2013

1 INTRODUCTION

In this project, xv6 operating system was extended with a shared memory functionality. In the extended version of the system, processes can communicate with each other via shared memory regions they request the kernel to allocate for them. Below, the report will go as follows. First, in the DESIGN section, the main design points are highlighted and the interface is presented. Also, KSM process state diagram is presented and discussed. Second, in the IMPLEMENTATION section, the edge cases of the system behaviour are considered and some implementation details are described. Each of the design decisions, advices, and implementation hints that were given in the KSM handout are addressed at least in one of the following sections. Finally, some additional notes are mentioned.

2 SHARED MEMORY DESIGN

We used the provided interface and implemented the following system calls

```
int ksmget(int key, uint size);
int ksmattach(int handle, int flag);
int ksmdetach(int handle);
int ksmdelete(int handle);
int ksminfo(int handle, ksminfo_t* ksm_info);
int pgused();
```

We decided the key (the identifier) to be an integer parameter, not a string parameter for the simplicity of the parameters parsing in the actual system call[1]. We also decided to return an integer from `ksmattach` instead of the address (`void*`) to be able to return negative error codes instead of 0 for any error that might happen. In our implementation only `ksmdetach` takes a flag: if it is zero, the segment is attached with read-only permission, otherwise it is attached with read-write permission.

If a process in xv6 uses shared memory, we should guarantee that the provided functionality is safe and behave in a predictable way. In order to accomplish this, we design a state machine that completely specifies the behaviour of the process that works with shared memory region(s). The diagram of the state machine is presented in Figure 2.1. The design is the following:

- From the initial state, a process can successfully call only `ksmget`. All the other calls should fail, because we do not allow attaching, detaching, deleting, and getting info about shared segments by a processes that has not got the shared segment. So, it means that any of these calls should fail even if the process provided a valid handle but have not got this handle via `ksmget` system call. We show in the next section the actual implementation of how we force this rule. The kernel also allocates the underlying physical memory when a shared segment is created (on the first `ksmget`). Current implementation also guarantees that `ksmget` returns the smallest possible free handle in case of getting a new shared segment.

- A process cannot attach any segment multiple times. In fact, `ksmattach` will return the same address it returned for the first attach. If attach is successful, it only maps the shared segment physical pages into the user memory space of the current process, since the actual segment has already been created by the first `ksmget`. The interesting point here is that between `ksmget` and `ksmattach` it might happen that some other process has deleted this segment, and a different group of processes have got this segment (actually, this handle) for their own purposes. In this case, the `ksmattach` call of the current process should also fail – otherwise, if it is not, the current process will get access to the shared memory of another group of processes (probably, without even knowing this) which is a security violation.

- To be successful, `ksmdetach` requires an attach of the segment with the same handle being done previously by the current process. Otherwise, it should return an error code. If it is successful, it should clean the page table of the current process from the entries that point to the shared segment we are detaching. Also, if the current process was the last one who detaches this segment, `ksmdetach` should also take care of destroying the segment – freeing the underlying physical pages, cleaning all the statistics for the segment, and so forth. After the segment is destroyed, another `ksmget` can return this handle if it is the smallest one among all the free handles.

- `ksmdelete` can be called any time after the segment has been created and got by the

[1]In the project handout, it was decided to have an integer identifier. So, the last page of the handout with the proposed KSM header file is not consistent with the previous discussion, and also it has a number of mistakes.

process. If the segment has never been attached, ksmdelete should behave similar to ksmdetach and destroy the segment. If it is still attached by some processes, it should be only marked for deletion. The current implementation also allows any number subsequent attaches to a segment which was marked for deletion. A “deleted” segment will anyway stay in the system while it is attached at least by one of the processes.

- ksminfo returns the info on the shared segment with provided handle if the segment with such handle exists and if the segment has been got before by the current process.

Besides all the above mentioned points, all five system calls should record and the information such as attach time, detach time, number of attached processes, etc.

3 Implementation Discussion

3.1 KSM Get and Attach Cooperation

As it was mentioned above (and also mentioned in the project handout), shared memory physical pages allocation and virtual addresses mappings should be conceptually separated. ksmget does the first job, ksmattach does the second. So, internally, within the kernel we maintain an array ksm_sgtable of shared segment structures ksmseg_t:

```
struct ksmseg_t {
  int id;              // Segment identifier
  pde_t* pgdir;        // Page directory for the shared memory segment
  uint ksmsz;          // The size of the shared memory segment
  int cpid;            // PID of the creator
  int mpid;            // PID of the last modifier
  uint attached_num;   // Number of attached processes
  uint atime;          // Last attach time
  uint dettime;        // Last detach time
  uint deltime;        // Last delete time (setted after the segment destroyed)
};
```

The handle that ksmget returns to the process, is actually the $index + 1$ of the segment in the ksm_sgtable array of structures[2].

When ksmget is called, and the segment is not created (the element of the ksm_sgtable for the given handle has 0 id), it allocates a page for the internal page directory ksmseg_t->pgdir, then it calls allocuvm providing this page directory, finally, if the previous actions are successful, it fills up the rest of the information into the segment structure. We do not round up or down the size to page sizes in ksmget since allocuvm doesn’t need it and does this itself.

Internal page directory allows us dynamically allocate shared segments of any size, keeping the kernel relatively small. Also, it allows us to attach the segment by just copying the internal page directory entries into the process’ user page directory. That is what ksmattach does besides the number of checks.

[2]We decided to make handles strictly positive, since ksminfo(handle) should also have a specific behavior if 0 handle is provided: it should return only the global information.

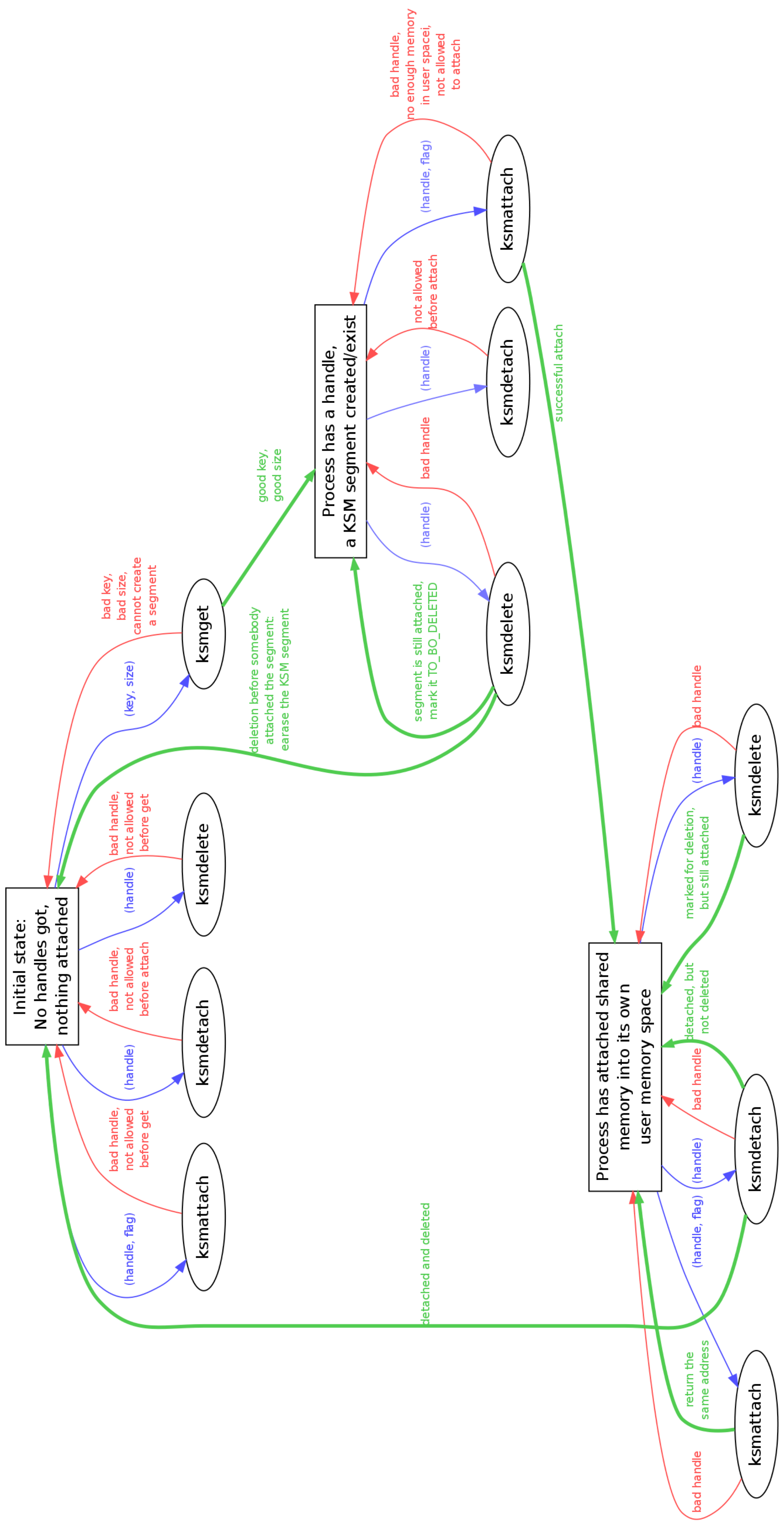

Figure 2.1: KSM process state machine diagram. Boxes denote process states in user mode, ellipses denote process states in kernel mode. Blue arrows denote user calls of `ksmget`, `ksmattach`, `ksmdetach`, and `ksmdelete`; red arrows stand for error returning; green arrows are successful state transitions.

The current shared memory implementation also adds ksmmemseg_t structure and extends proc structure with the following fields:

```
struct proc {
  // All the proc fields remain the same. The new field are presented below.
  //...
  // KSM information
  char* ksm_bottom;                // The current lowest address of the ksm
  char* ksm_freebitmap;            // Bit map for the shared memory region
  struct ksmmemseg_t* ksm_mstable; // Shared memory segments location info
};

// KSM proc shared memory segment info structure
struct ksmmemseg_t {
  char* bottom;          // A pointer to the bottom of a segment
  uint pgnum;            // The size of the segment in pages
  uint gettime;          // The time when the proc called get for this seg
};
```

While attaching, the kernel should keep track of the virtual memory state of the process. We decided to use *first fit* policy, and use ksm_freebitmap to implement this: Whenever a process attaches a new segment, the kernel searches from the beginning of the bit map for a free virtual memory chunk that fits the request. ksm_bottom points to the current bottom of the shared memory in the user virtual memory to make the kernel able to prevent user memory and shared memory overlaps, e.g. when sbrk is called. ksmmemseg_t->bottom points to the start address of a shared segment attached to the process.

One of the important fields in the ksmmemseg_t structure is gettime. This counter is updated every time a successful ksmget done by the process. Whenever we call ksmattach, ksmdetach, or ksminfo, the kernel checks whether the delete time of the segment with the requested handle is less than the get time for the same segment (the same handle) for the current process. If the check is not passed, the current process is not allowed to attach, detach, or get info on the current segment, since it has not got it before, or it has, but the segment was deleted and then a new one with the same handle was created by some other process. This exactly solves all the security issues mentioned in the previous section.

3.2 KSM Delete, Detach and the Fork/Exec/Exit Behavior

The behaviour of ksmdetach and ksmdelete were described in the design section of this report. The only thing that we should mention is that ksmdetach should properly update the ksm_mstable[handle].ksm_freebitmap to prevent any memory leakages or unpredictable behaviors of the system. This is controlled by update_ksmbitmap internal static function.

On fork, as it was mentioned in the handout, we make the child process inherit all the parent's shared segments, and hence we should update all the kernel shared memory structures, e.g. attached_num. One could think, that this could be implemented via attaching the same segments for the child using ksmattach. This doesn't work in the current implementation, since while attaching we use *first fit* algorithm to find the virtual address to attach the segment to. So, we do not know what is the current order of the attached segments in the parent's virtual memory, and hence we cannot call `ksmattach` in the child in the same order. So, `ksm_copy_proc` function was implemented to copy everything needed from a parent process to the child process and to update all the necessary KSM structures.

On exec and on exit, as it was also suggested in the project handout, we detach all the segments attached to the current process. We call `ksmdetach` in a loop for all the possible handles, and it works flawlessly.

4 Additional Notes

The proposed `ksminfo_t` structure was actually changed:

```
// KSM info structure (to be returned on sys call ksminfo)
struct ksminfo_t {
  uint ksmsz;            // The size of the shared memory segment
  int cpid;              // PID of the creator
  int mpid;              // PID of the last modifier
  uint attached_num;     // Number of attached processes
  uint gtime;            // Last get time by the current process
  uint atime;            // Last attach time
  uint dettime;          // Last detach time
  uint deltime;          // Last delete time
  uint total_shsg_num;   // Total number of existing shared segments
  uint total_shpg_num;   // Total number of existing shared pages
};
```

Besides some additional fields with debugging purpose, the crucial difference is that the global parameters `total_shsg_num` and `total_shpg_num` are plugged into the structure instead of returning a pointer to a `struct ksmglobalinfo_t`. If we returned such a pointer, it will contain a virtual address which points to an address above the KERNBASE. So, it will be useless in a user program: whenever one tries to access this memory, the trap 14 is invoked which stands for page fault.

The current KSM implementation has some limitations. First, it limits the number of possible shared segments in the whole system up to 64. Second, shared memory segment size is limited up to 2 MB. Shared memory functionality for xv6 has been implemented as a separate module that uses the existing functionality and API of the current system and follows DRY[3] principle. The public interface is described in the header file *ksm.h*, the intrinsic implementation structures and function are hidden in *ksm.c*. Also, process structure `proc` was extended, and some of the virtual memory functions were patched to make KSM possible to integrate into the system (e.g. `growproc` function got an additional check, and `freevm`'s interface was changed with an additional parameter).

The whole implementation was tested. All the unit tests were wrapped into `ksmtest` user program. Alternative tests, provided by the TA, were implemented in `ammar_ksmtest` user program. For any additional details, please refer to the code which is well commented.

[3]DRY stands for *Don't Repeat Yourself.*