**Exercise Goal:** You will learn how the existing System V shared memory facility is implemented in the kernel. Then, you will modify the standard kernel so that it uses a dynamic data structure for managing memory segments.

**Introduction**

Shared memory is a common block of memory that is mapped into the address spaces of two or more processes. Information that a process writes to a location in the shared memory can be read with a memory read operation in any other process that is using the shared memory. Within an individual computer—either uniprocessor or multiprocessor—shared memory is usually the fastest way for two processes to share information.

Linux's shared memory feature is derived from the form of shared memory introduced in System V UNIX. Even though the shared memory mechanism allows multiple processes to map a common memory segment into their own address spaces—the segment is logically a part of the memory manager—it was designed and implemented in System V UNIX as a part of the IPC (interprocess communication) mechanism.
The Shared Memory API

You can use shared memory to allow any process to dynamically define a new block of memory that is independent of the address space created for the process before it began to execute. As explained in Part 1 of this manual, every Linux process is created with a large virtual address space, only a portion of which is used to reference the compiled code, static data, stack, and heap. The remaining addresses in the virtual address space are initially unused. A new block of shared memory, once defined, is mapped into a block of the unused virtual addresses, after which the process can read and write the shared memory as if it were ordinary memory. Of course, more than one process can map the shared memory block into its own address space, so code segments that read or write shared memory are normally considered to be critical sections.

The following four system calls define the full kernel interface to support shared memory.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, int size, int shmflg);
void *shmat(int shmid, char *shmaddr, int shmflg);
void *shmdt(char *shaddr);
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- `shmget()` creates the shared memory block.
- `shmat()` maps an existing shared memory block into a process's address space.
- `shmdt()` removes (unmaps) a shared memory block from the process's address space.
- `shmctl()` is a general-purpose function (in the style of `ioctl()`) used to perform all other commands to the shared memory block.

To create a new block of shared memory, the process must call `shmget()`. If `shmget()` successfully creates a new block of memory, then it returns a shared memory identifier of type `int`. The shared memory identifier is a handle into a kernel data structure. The next section discusses how shared memory is implemented, illustrating that it is actually an index into a kernel array. If `shmget()` can create the new block of shared memory, then the kernel array whose index is returned references an instance of the `struct shmid_kernel` data structure, which includes a field, as follows.
struct shmid_kernel
{
    shmid_ds u;
    ...
};

The arguments to shmget() are key_t key, int size, and int shmflg. The size argument specifies the size of the new block of memory. All memory allocation operations are in terms of pages. That is, if a process requests 1 byte of memory, then the memory manager allocates a full page (PAGE_SIZE = 4,096 bytes on i386 machines). Thus the size of the new shared memory block will be the value of the size argument rounded up to the next multiple of the page size. A size of 1 to 4,096 results in a 4K (one-page) block, 4,097 to 8,192 results in an 8K (two-page) block, and so on.

The key argument may be either the key of an existing memory block, 0, or IPC_PRIVATE. When it is IPC_PRIVATE, the shmget() call creates a new block of shared memory. When it is 0, and the IPC_CREAT flag is set in the shmflg argument, then shmflg also can cause a new block to be created. A process that wants to reference a shared memory block that was created by another process (such as a parent or server) usually will obtain the struct shmid_kernel reference from the creator. However, it also can set the key argument to the key value of an existing memory block. If key’s value is so set, and shmflg is set with IPC_CREAT | IPC_EXCL, then shmget() will fail. shmflg also must define the access permissions for user, group, and world accesses to the memory block as its lower 9 bits (by using the same bit pattern as for file protection).

When a shared memory region is successfully created, shmget() returns an integer reference to its struct shmid_ds.

struct shmid_ds {
    struct ipc_perm shm_perm;  /* operation perms */
    int shm_segsz;  /* size of segment (bytes) */
    __kernel_time_t shm_atime;  /* last attach time */
    __kernel_time_t shm_dtime;  /* last detach time */
    __kernel_time_t shm_ctime;  /* last change time */
    __kernel_ipc_pid_t shm_cpid;  /* pid of creator */
    __kernel_ipc_pid_t shm_lpid;  /* pid of last operator */
    unsigned short shm_nattch;  /* no. of current attaches */
    unsigned short shm_unetch;  /* compatibility */
}

1 The shmflg argument is a set of single-bit flags. Setting multiple flags in shmflg involves combining them with a logical OR operator ("|") and then assigning them to shmflg.
You use the struct ipc_perm shm_perm field to define the owner of the shared memory block and the permissions for other processes to use the block. It contains fields in which to specify the owner’s user and group ids, the creator’s user and group ids, the access mode (read or write), and the key value for the memory block. (The man page for shmget() describes the shmid_ds and ipc_perm structure fields.)

The void *shmat(int shmid, char *shmaddr, int shmflg) system call maps the memory block into the calling process’s address space. The shmid argument is the result returned by the shmget() call that created the block. The shmaddr pointer is the virtual address to which the first location in the shared memory block should be mapped. If the calling process does not wish to choose the address to which the memory block will be mapped, it should pass a value of 0 for shmaddr. shmaddr’s value should be aligned with a page boundary. If shmaddr is specified, and SHM_RND is asserted in shmflg, then the address will be rounded down to a multiple of the SHMLBA constant. shmflg is used in the same way as the corresponding flag in shmget(), that is, to assert a number of different 1-bit flags to the shmat() system call. In addition to asserting the SHM_RND flag, the calling process can assert SHM_RDONLY to attach the memory block so that it can be only read, and not also written.

When a process finishes using a shared memory block, it calls void *shmdt(char *shaddr), where shaddr is the address used to attach the memory block. The kernel then updates the corresponding struct shmid_kernel to reflect that this process is no longer using the memory block.

The final shared memory call is int shmctl(int shmid, int cmd, struct shmid_ds *buf). This call performs control operations on the shared memory block descriptor. The shmid argument identifies the shared memory block, and the cmd argument specifies the command to be applied to the descriptor. If cmd is set to IPC_STAT, then the calling process must provide a buffer, buf, that is at least as large as a struct shmid_kernel. shmctl() fills in the current values of the shmid_ds and returns them in buf. If cmd is set to IPC_SET, then the data structure will be updated, provided that the calling process is an owner or creator (or has superuser permission). A call with cmd set to IPC_RMID causes the memory segment to be destroyed (it will not otherwise be destroyed, even when no process is attached to it). A superuser can lock or unlock the block.
Here is a simple example of a parent process that creates a shared memory block and then creates a child process that also can use the block.

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/wait.h>

#define SHM_SIZE ...

void run_child(int, int);

int main() {
    int pid, shm_handle, status;
    char *my_shm_ptr;

    /* Create the shared memory */
    shm_handle = shmget(IPC_PRIVATE, SHM_SIZE, IPC_CREAT | IPC_CREAT | 0x1C0);
    if(shm_handle == -1) {
        printf("Shared memory creation failed\n");
        exit(0);
    }

    /* Start the child */
    if((pid = fork()) == 0) {
        run_child(childNum, shm_handle);
        exit(0);
    }

    /* Do work, share results with child via shared memory */
    my_shm_ptr = (char *) shmat(shm_handle, 0, 0);
    if(my_shm_ptr == (char *) -1) {
        printf("Shared memory attach failed\n");
        exit(0);
    }

    /* Wait for the children to finish */
    wait(&status);
    shmctl(shm_handle, IPC_RMID, 0); /* Remove shared
```
```c
memory */
printf("Parent: Terminating\n");
}

void run_child(int me, int shm_handle) {
    char *my_shm_ptr;
    int i;
    unsigned int shm_flag  = 0;

    /* Attach the shared memory */
    my_shm_ptr = (char *) shmat(shm_handle, 0, 0);
    if(my_shm_ptr == (char *) -1) {
        printf("Shared memory attach failed\n");
        exit(0);
    }
    *(my_shm_ptr+64+i) = ...; /* Write shmem location i */
    ... = my_shm_ptr+i; /* Read shmem location i */
}

The Implementation

Shared memory is implemented entirely within the file ipc/shm.c. The four system calls described in the previous section appear in this file.

asmlinkage int sys_shmget (key_t key, int size, int shmflg)
asmlinkage int sys_shmat (int shmid, char *shmaddr, int shmflg,
ulong *raddr)
asmlinkage int sys_shmdt (char *shmaddr)
asmlinkage int sys_shmctl (int shmid, int cmd, struct shmid_ds
    *buf)

The shm_init() function initializes the shared memory data structures. It is called by ipc_init, which is called by start_kernel() during the general startup procedure.

void __init shm_init (void)
{
    int id;

    for (id = 0; id < SHMMNI; id++)
        shm_segs[id] = (struct shmid_kernel *) IPC_UNUSED;
    shm_tot = shm_rss = shm_seq = max_shmid = used_segs = 0;
}
The shared segment table, `shm_segs[]`, is the kernel data structure that is filled in whenever a new shared memory block (or segment) is created by `shmget()`. The `shmid` value returned by `shmget()` is the index into `shm_segs[]`. This routine initializes the array by setting each entry in the static array to IPC_UNUSED. It then initializes various global variables that represent the number of segments, and so on.

The `sys_shmget()` system call is shown next in its entirety.

```c
asmlinkage int sys_shmget (key_t key, int size, int shmflg)
{
    struct shmid_kernel *shp;
    int err, id = 0;

    down(&current->mm->mmap_sem);
    lock_kernel(); // New in Version 2.2 for SMP support
    if (size < 0 || size > shmmax) {
        err = -EINVAL;
    } else if (key == IPC_PRIVATE) {
        err = newseg(key, shmflg, size);
    } else if ((id = findkey(key)) == -1) {
        if (!(shmflg & IPC_CREAT))
            err = -ENOENT;
        else
            err = -EACCES;
    } else
        err = (int) shp->u.shm_perm.seq * SHMNI + id;

    unlock_kernel(); // New in Version 2.2 for SMP support
    up(&current->mm->mmap_sem);
    return err;
    return newseg(key, shmflg, size);
}
```

Most of the work is done in the `newseg()` function. Notice that when the key is set to IPC_PRIVATE, the system call only checks to see that the size is positive and not too large and then it calls `newseg()`. If another option is chosen,
then the findseq() function is called to determine whether the segment currently exists.

An interesting race condition can happen between findseq() and newseg() that is resolved by their code using the value of the shm_segs[i] entry. See the code for these functions in ipc/shm.c and further discussion in the following Attacking the Problem.

The shmat() system call maps a shared memory segment into the calling process’s address space. Following is a skeleton of the code. (Most of the error checking has been removed so that you can focus on the function’s essential parts.)

```c
asmlinkage int sys_shmat (int shmid, char *shmaddr, int shmflg, ulong *raddr)
{
    struct shmid_kernel *shp;
    struct vm_area_struct *shmd;
    int err = -EINVAL;
    unsigned int id;
    unsigned long addr;
    unsigned long len;

    shp = shm_segs[id = (unsigned int) shmid % SHMMNI];

    if (!addr = (ulong) shmaddr) {
        if (shmflg & SHM_REMAP)
            goto out;
        err = -ENOMEM;
        addr = 0;
        again:
        if (!addr = get_unmapped_area(addr, shp->u.shm_segsz))
            goto out;
        if (addr & (SHMLBA-1)) {
            addr = (addr + (SHMLBA-1)) & ~(SHMLBA-1);
            goto again;
        }
    } else if (addr & (SHMLBA-1)) {
        if (shmflg & SHM_RND)
            addr &=(~(SHLBA-1)); /* round down */
        else
            goto out;
```
/* Check if addr exceeds TAX_SIZE (from do_mmap) */
len = PAGE_SIZE*shp->shm_npages;
err = -EINVAL;
if (addr >= TASK_SIZE || len > TASK_SIZE
    || addr > TASK_SIZE - len)
    goto out;
/*
* If shm segment goes below stack, make sure there is some
* space left for the stack to grow (presently 4 pages).
*/
if (addr < current->mm->start_stack &&
    addr > current->mm->start_stack -
    PAGE_SIZE*(shp->shm_npages + 4))
{
    goto out;
}
err = -EACCES;
if (ipcperms(&shp->shm_perm, shmflg &
    SHM_RDONLY ? S_IRUGO : S_IRUGO|S_IWUGO))
    goto out;
    err = -EIDRM;
if (shp->u.shm_perm.seq != (unsigned int) shmid / SHMMNI)
    goto OUT;
shmd = kmem_cache_alloc(vm_area_cachep, shmd);
shmd->vm_pte = SWP_ENTRY(SHM_SWP_TYPE, id);
shmd->vm_start = addr;
shmd->vm_end = addr + shp->shm_npages * PAGE_SIZE;
shmd->vm_mm = current->mm;
shmd->vm_page_prot =
    (shmflg & SHM_RDONLY) ? PAGE_READONLY : PAGE_SHARED;
shmd->vm_flags = VM_SHM | VM_MAYSHARE | VM_SHARED
    | VM_MAYREAD | VM_MAYEXEC | VM_READ | VM_EXEC
    | ((shmflg & SHM_RDONLY) ? 0 : VM_MAYWRITE | VM_WRITE);
shmd->vm_file = NULL;
shmd->vm_offset = 0;
shmd->vm_ops = &shm_vm_ops;
The routine first looks up the struct shmid_ds for the given id, shmid, saving the pointer in shp. It then checks shmaddr, adjusting it as required (page or SHMLBA boundary). Next, the segment is checked to determine whether it will fit into the virtual address space, starting at shmaddr. In the next part of the code, the process descriptor (current) is checked to determine whether the block interferes with the stack. If permissions are acceptable, then the function allocates a new instance of a virtual memory structure, struct vm_area_struct *shmd, and fills it in so that it indicates that a shared memory block has been added to the virtual address space.

A currently attached shared memory segment is released by the shmdt() system call, shown as follows.

```c
asmlinkage int sys_shmdt (char *shmaddr)
{
    struct vm_area_struct *shmd, *shmdnext;

    down(&current->mm->mmap_sem);
    lock_kernel();
    for (shmd = current->mm->mmap; shmd; shmd = shmdnext) {
        shmdnext = shmd->vm_next;
        if (shmd->vm_ops == &shm_vm_ops && shmd->vm_start - shmd->vm_offset == (ulong) shmaddr)
            do_munmap(shmd->vm_start, shmd->vm_end - shmd->vm_start);
    }
}
```
unlock_kernel();
up(&current->mm->mmap_sem);
return 0;
}

This code steps through the struct vm_area_struct instances, which are added to the current->mm-mmap list by the insert_attach() call in shmat(). It then releases the block that contains shmaddr. The fields in the for statement qualify as “tricky code.”

The final system call is the shmct1() function. The following code skeleton shows only the code that implements the IPC_STAT command (some of the error checking code was removed).

asmlinkage int sys_shmctl (int shmid, int cmd, struct shmid_ds *buf)
{
    struct shmid_ds tbuf;
    struct shmid_kernel *shp;
    struct ipc_perm *ipcp;
    int id, err;

    lock_kernel();

    switch (cmd) { /* replace with proc interface ? */
        case IPC_INFO:
            ...
        case SHM_INFO:
            ...
        case SHM_STAT:
            ...
    }

    shp = shm_segs[id = (unsigned int) shmid % SHMMNI];
    ipcp = &shp->u.shm_perm;

    switch (cmd) {
        case SHM_UNLOCK:
            ...
        case SHM_LOCK:
            ...
        case IPC_STAT:
            ...
    }
}
After checking the input parameters, `shmctl()` uses a switch statement to check for commands that the `ipcs(8)` shell command uses (IPC_INFO, SHM_STAT, and SHM_INFO). Next, it retrieves the struct `shmid_ds` for the specified `shm_id` and enters a second switch statement that contains a case for IPC_STAT. It then determines whether the calling process has permission to read the status and permission to write the result back into the user space (in the area referenced by the `buf` parameter). Finally, it copies the data into a temporary buffer; the data is then copied into user space.

This introduction to memory management did not discuss how the memory manager implements virtual memory. That is the main topic of the next exercise.

### Problem Statement

#### Part A
Write a program that creates and uses — reads and writes — multiple blocks of shared memory.

#### Part B
The file `ipc/shm.c` implements the normal version of System V shared memory. It uses a static array, `shm_segs[]`, to keep track of each shared memory
segment that is created. UNIX kernels traditionally have used static arrays for multirecord data structures (for example, the process descriptors are usually implemented as a static array). Your task is to modify the existing version of ipc/shm.c so that it uses a list, rather than a static array, to keep track of shared memory segments. This is a pedagogic exercise rather than an actual improvement to the kernel. That is, although it allows an arbitrary number of memory segments to be created, list manipulation will be much slower than array dereferencing would be.

**Attacking the Problem**

First, you should write a simple driver program that uses the standard form of shared memory. Doing this will help you to understand how the facility works, as well as provide you with a program for testing your new version of ipc/shm.c. Be sure that the test program uses multiple shared memory segments.

You can edit and compile your new version of ipc/shm.c in user space (in your own development directory). Make two copies of ipc/shm.c in your development directory, one to be the original version that is stored in the Linux source tree and the other to be your new version. Study ipc/shm.c carefully to understand how it interacts with the shm_segs[] array.

You will discover an interesting race condition that is resolved by using special pointers in the shm_segs[] array (the IPC_NOID and IPC UNUSED pointers). The race condition can occur if findkey() is called when newseg() is in progress. That is, the data structure that newseg() creates may be only partially filled when findkey() runs; it might then return an id for an inconsistent data structure. The code uses IPC_NOID as a flag to block findkey() if it begins to run when newseg() is in progress. newseg() wakes the blocked findkey() process, if it became blocked on the IPC_NOID value for the segment pointer. Now read the code to see exactly how this works. When you modify the code, you will probably remove this race condition. However, you will add a new one, linking/unlinking a node in a list. Of course, you will need to protect this new race condition.

Design your list, and modify the code in your experimental version of ipc/shm.c in your development directory. Compile the code by using the compile flags –D__kernel__ and –Wall.

In designing your solution, you need to find all parts of the existing code that reference the struct shmid_kernel shm_segs[] data structure and then change those references so that they access the individual data structures from a list rather than from an array. First, observe that the existing file allocates shm_segs[] as a static, global array:
static struct shmid_kernel *shm_segs[SHMMNI];

You need to replace this declaration with your new list descriptor—a static variable that points to the list of struct shmid_kernel entries. Next, you need to find each place at which the code references the array and ensure that it references a list entry. You will probably have to change the shm.h file as well (it is located in include/linux/shm.h).

You should not change any parts of ipc/shm.c that do not manipulate shm_segs[]. You should change only the way that the struct shmid_kernel data structure is retrieved from the collection of such data structures.

Once your code compiles successfully, and you are sure that your solution is correct, place your new version of shm.c into the Linux source directory, replacing the original ipc/shm.c. You will have copies of your experimental version and the original version in your development directory. Recompile the kernel, and make a new boot floppy disk. Test your code by rebooting from the floppy disk and running your test code that worked with the original version of ipc/shm.c.