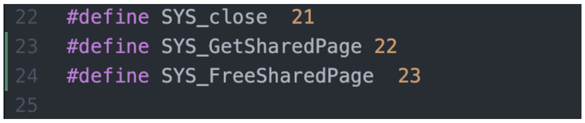
Timur Gaimakov: A20415319
Hassan Alamri: A204730047

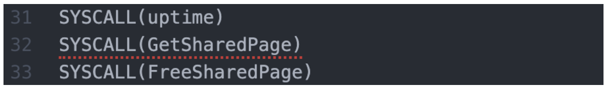
Files modified

syscall.h (lines 23-24)



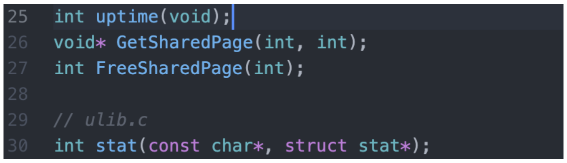
```
22  #define SYS_close  21
23  #define SYS_GetSharedPage 22
24  #define SYS_FreeSharedPage  23
25
```

usys.S (lines 32-33)



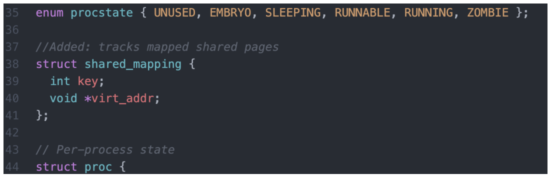
```
31  SYSCALL(uptime)
32  SYSCALL(GetSharedPage)
33  SYSCALL(FreeSharedPage)
```

user.h (lines 26-27)

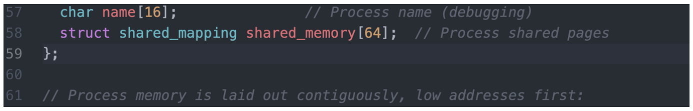


```
25  int uptime(void);
26  void* GetSharedPage(int, int);
27  int FreeSharedPage(int);
28
29  // ulib.c
30  int stat(const char*, struct stat*);
```

proc.h (lines 38-41 and line 58)

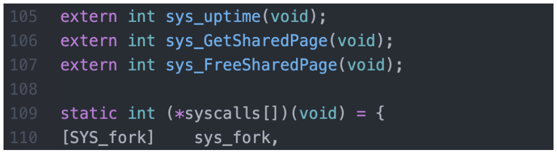


```
35  enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
36
37  //Added: tracks mapped shared pages
38  struct shared_mapping {
39    int key;
40    void *virt_addr;
41  };
42
43  // Per-process state
44  struct proc {
```

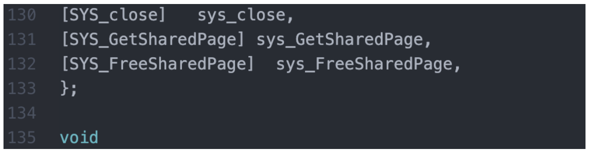


```
57    char name[16];               // Process name (debugging)
58    struct shared_mapping shared_memory[64];  // Process shared pages
59  };
60
61  // Process memory is laid out contiguously, low addresses first:
```

syscall.c (lines 106-107 and 131-132)

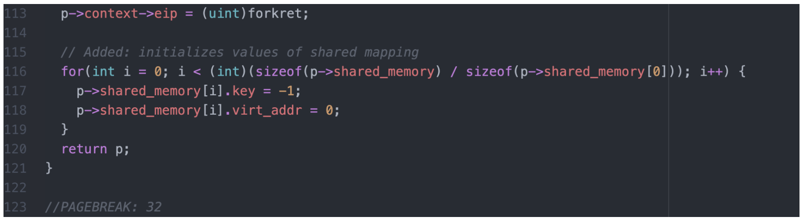


```
105  extern int sys_uptime(void);
106  extern int sys_GetSharedPage(void);
107  extern int sys_FreeSharedPage(void);
108
109  static int (*syscalls[])(void) = {
110  [SYS_fork]    sys_fork,
```

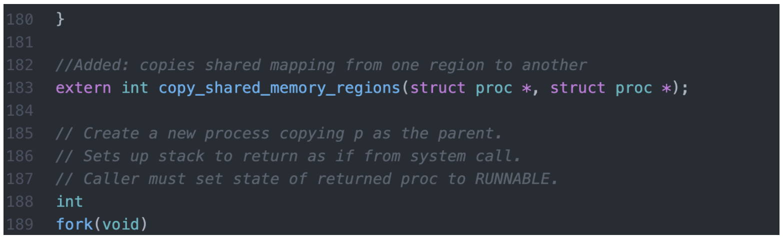


```
130  [SYS_close]   sys_close,
131  [SYS_GetSharedPage] sys_GetSharedPage,
132  [SYS_FreeSharedPage]  sys_FreeSharedPage,
133  };
134
135  void
```

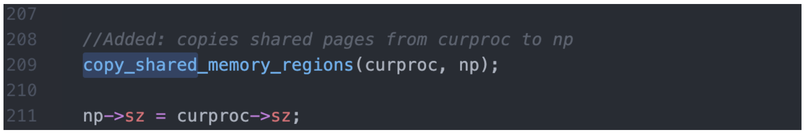
proc.c (lines 116-119; line 183 [between growproc() and fork()]; line 209 [in fork()])



```
113    p->context->eip = (uint)forkret;
114
115    // Added: initializes values of shared mapping
116    for(int i = 0; i < (int)(sizeof(p->shared_memory) / sizeof(p->shared_memory[0])); i++) {
117      p->shared_memory[i].key = -1;
118      p->shared_memory[i].virt_addr = 0;
119    }
120    return p;
121  }
122
123  //PAGEBREAK: 32
```

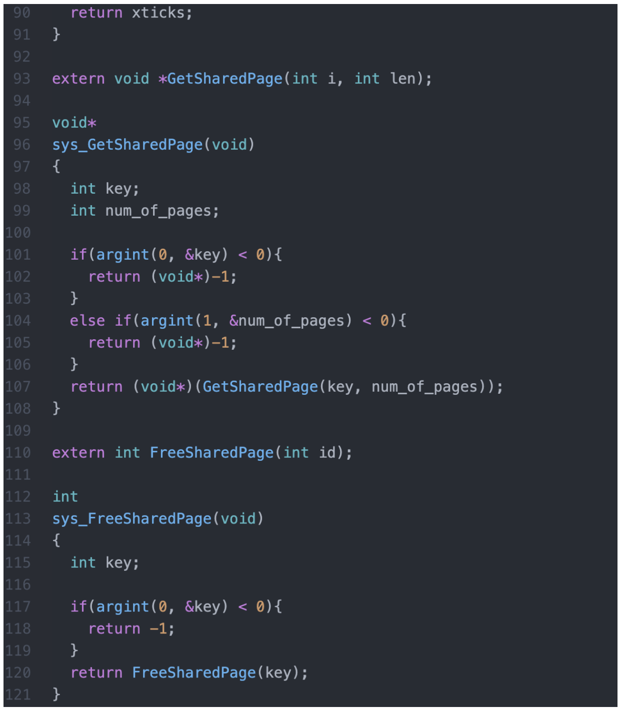


```
180  }
181
182  //Added: copies shared mapping from one region to another
183  extern int copy_shared_memory_regions(struct proc *, struct proc *);
184
185  // Create a new process copying p as the parent.
186  // Sets up stack to return as if from system call.
187  // Caller must set state of returned proc to RUNNABLE.
188  int
189  fork(void)
```



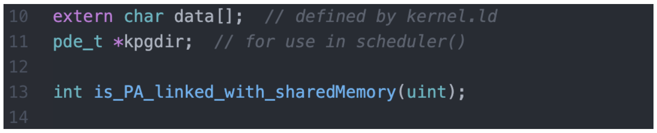
```
207
208    //Added: copies shared pages from curproc to np
209    copy_shared_memory_regions(curproc, np);
210
211    np->sz = curproc->sz;
```

sysproc.c (EOF; lines 93 - 120)

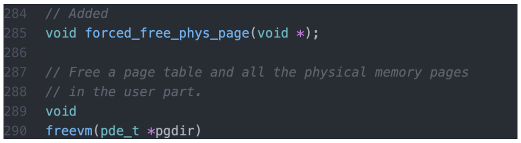


```
 90    return xticks;
 91  }
 92
 93  extern void *GetSharedPage(int i, int len);
 94
 95  void*
 96  sys_GetSharedPage(void)
 97  {
 98    int key;
 99    int num_of_pages;
100
101    if(argint(0, &key) < 0){
102      return (void*)-1;
103    }
104    else if(argint(1, &num_of_pages) < 0){
105      return (void*)-1;
106    }
107    return (void*)(GetSharedPage(key, num_of_pages));
108  }
109
110  extern int FreeSharedPage(int id);
111
112  int
113  sys_FreeSharedPage(void)
114  {
115    int key;
116
117    if(argint(0, &key) < 0){
118      return -1;
119    }
120    return FreeSharedPage(key);
121  }
```

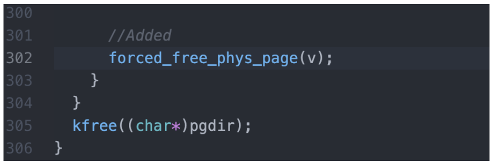
vm.c (lines 13, 285, 302, 403 - EOF)



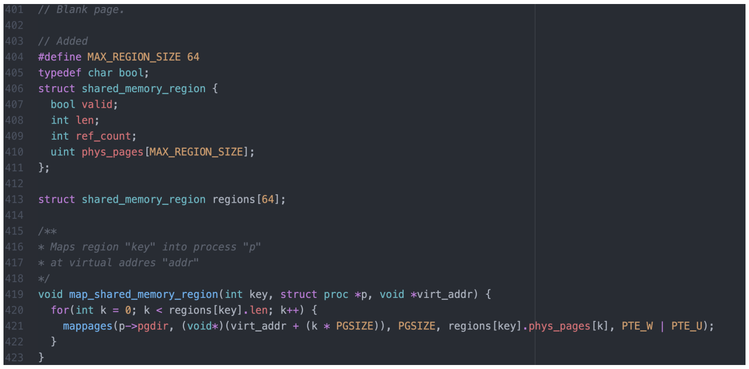
```
10  extern char data[];  // defined by kernel.ld
11  pde_t *kpgdir;  // for use in scheduler()
12
13  int is_PA_linked_with_sharedMemory(uint);
14
```



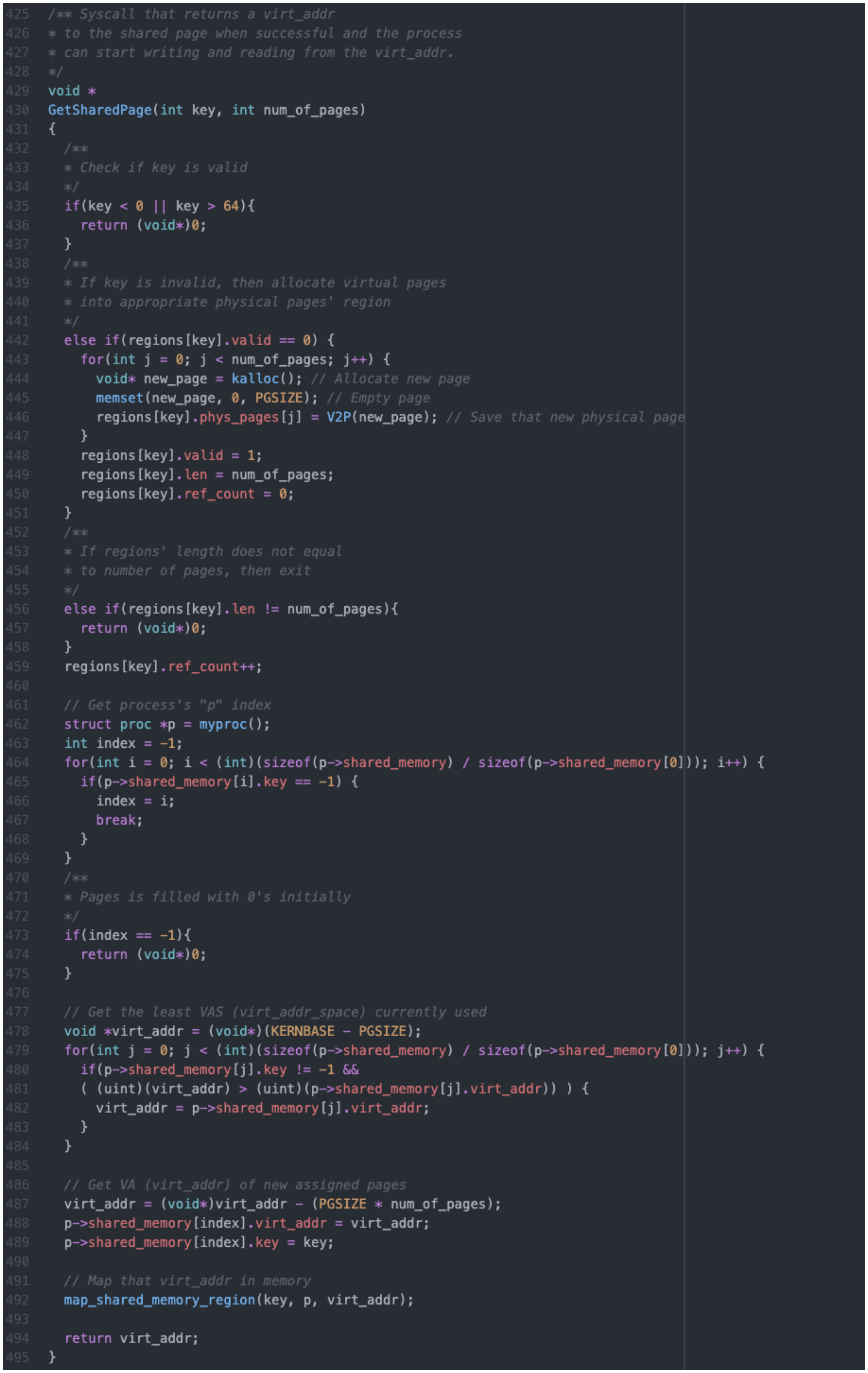
```
284  // Added
285  void forced_free_phys_page(void *);
286
287  // Free a page table and all the physical memory pages
288  // in the user part.
289  void
290  freevm(pde_t *pgdir)
```



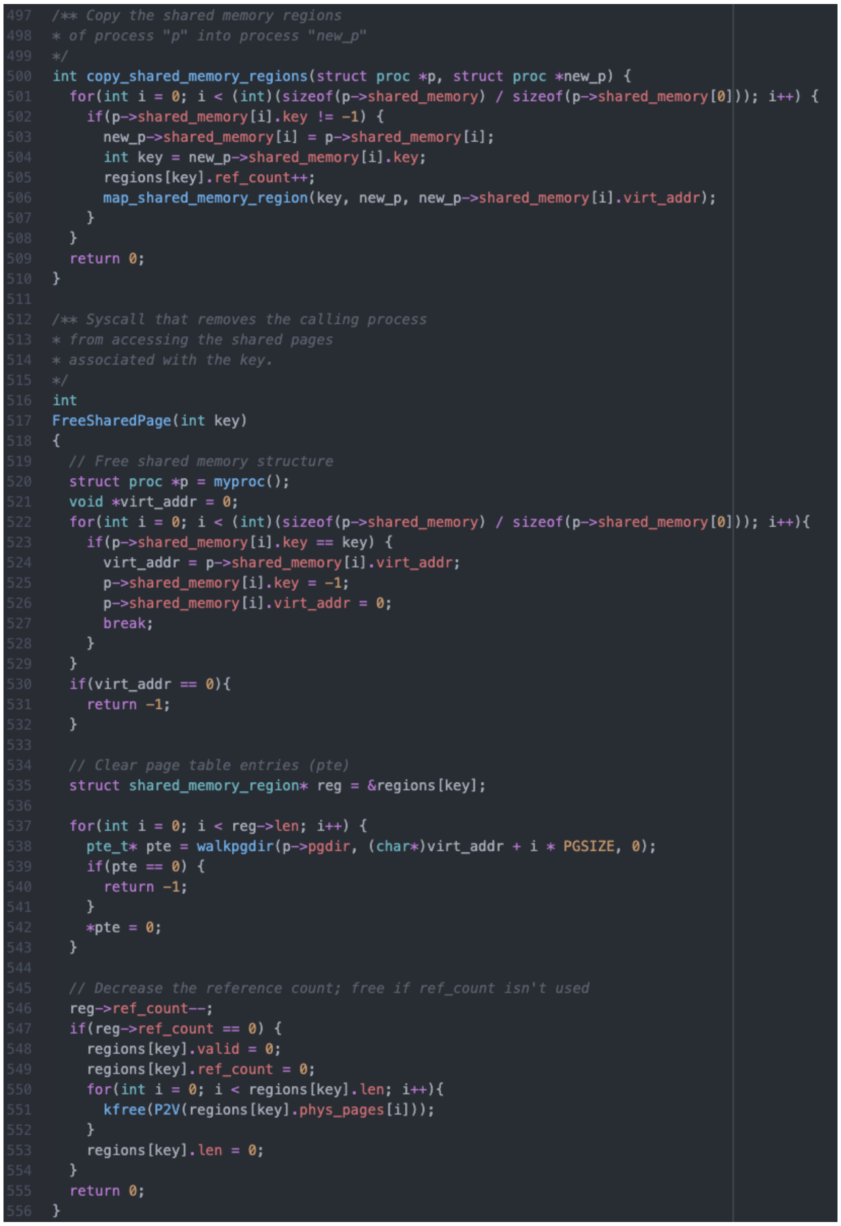
```
300
301        //Added
302        forced_free_phys_page(v);
303      }
304    }
305    kfree((char*)pgdir);
306  }
```



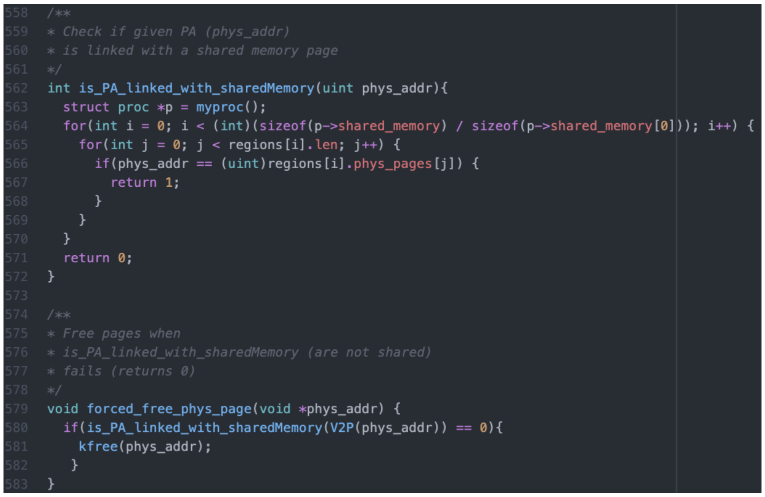
```
401  // Blank page.
402
403  // Added
404  #define MAX_REGION_SIZE 64
405  typedef char bool;
406  struct shared_memory_region {
407    bool valid;
408    int len;
409    int ref_count;
410    uint phys_pages[MAX_REGION_SIZE];
411  };
412
413  struct shared_memory_region regions[64];
414
415  /**
416  * Maps region "key" into process "p"
417  * at virtual addres "addr"
418  */
419  void map_shared_memory_region(int key, struct proc *p, void *virt_addr) {
420    for(int k = 0; k < regions[key].len; k++) {
421      mappages(p->pgdir, (void*)(virt_addr + (k * PGSIZE)), PGSIZE, regions[key].phys_pages[k], PTE_W | PTE_U);
422    }
423  }
```



```c
/** Syscall that returns a virt_addr
* to the shared page when successful and the process
* can start writing and reading from the virt_addr.
*/
void *
GetSharedPage(int key, int num_of_pages)
{
  /**
  * Check if key is valid
  */
  if(key < 0 || key > 64){
    return (void*)0;
  }
  /**
  * If key is invalid, then allocate virtual pages
  * into appropriate physical pages' region
  */
  else if(regions[key].valid == 0) {
    for(int j = 0; j < num_of_pages; j++) {
      void* new_page = kalloc(); // Allocate new page
      memset(new_page, 0, PGSIZE); // Empty page
      regions[key].phys_pages[j] = V2P(new_page); // Save that new physical page
    }
    regions[key].valid = 1;
    regions[key].len = num_of_pages;
    regions[key].ref_count = 0;
  }
  /**
  * If regions' length does not equal
  * to number of pages, then exit
  */
  else if(regions[key].len != num_of_pages){
    return (void*)0;
  }
  regions[key].ref_count++;

  // Get process's "p" index
  struct proc *p = myproc();
  int index = -1;
  for(int i = 0; i < (int)(sizeof(p->shared_memory) / sizeof(p->shared_memory[0])); i++) {
    if(p->shared_memory[i].key == -1) {
      index = i;
      break;
    }
  }
  /**
  * Pages is filled with 0's initially
  */
  if(index == -1){
    return (void*)0;
  }

  // Get the least VAS (virt_addr_space) currently used
  void *virt_addr = (void*)(KERNBASE - PGSIZE);
  for(int j = 0; j < (int)(sizeof(p->shared_memory) / sizeof(p->shared_memory[0])); j++) {
    if(p->shared_memory[j].key != -1 &&
    ( (uint)(virt_addr) > (uint)(p->shared_memory[j].virt_addr)) ) {
      virt_addr = p->shared_memory[j].virt_addr;
    }
  }

  // Get VA (virt_addr) of new assigned pages
  virt_addr = (void*)virt_addr - (PGSIZE * num_of_pages);
  p->shared_memory[index].virt_addr = virt_addr;
  p->shared_memory[index].key = key;

  // Map that virt_addr in memory
  map_shared_memory_region(key, p, virt_addr);

  return virt_addr;
}
```

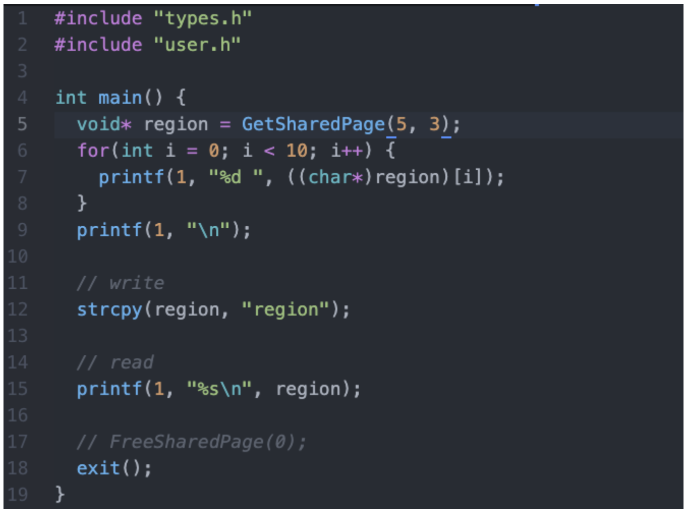


```c
/** Copy the shared memory regions
* of process "p" into process "new_p"
*/
int copy_shared_memory_regions(struct proc *p, struct proc *new_p) {
  for(int i = 0; i < (int)(sizeof(p->shared_memory) / sizeof(p->shared_memory[0])); i++) {
    if(p->shared_memory[i].key != -1) {
      new_p->shared_memory[i] = p->shared_memory[i];
      int key = new_p->shared_memory[i].key;
      regions[key].ref_count++;
      map_shared_memory_region(key, new_p, new_p->shared_memory[i].virt_addr);
    }
  }
  return 0;
}

/** Syscall that removes the calling process
* from accessing the shared pages
* associated with the key.
*/
int
FreeSharedPage(int key)
{
  // Free shared memory structure
  struct proc *p = myproc();
  void *virt_addr = 0;
  for(int i = 0; i < (int)(sizeof(p->shared_memory) / sizeof(p->shared_memory[0])); i++){
    if(p->shared_memory[i].key == key) {
      virt_addr = p->shared_memory[i].virt_addr;
      p->shared_memory[i].key = -1;
      p->shared_memory[i].virt_addr = 0;
      break;
    }
  }
  if(virt_addr == 0){
    return -1;
  }

  // Clear page table entries (pte)
  struct shared_memory_region* reg = &regions[key];

  for(int i = 0; i < reg->len; i++) {
    pte_t* pte = walkpgdir(p->pgdir, (char*)virt_addr + i * PGSIZE, 0);
    if(pte == 0) {
      return -1;
    }
    *pte = 0;
  }

  // Decrease the reference count; free if ref_count isn't used
  reg->ref_count--;
  if(reg->ref_count == 0) {
    regions[key].valid = 0;
    regions[key].ref_count = 0;
    for(int i = 0; i < regions[key].len; i++){
      kfree(P2V(regions[key].phys_pages[i]));
    }
    regions[key].len = 0;
  }
  return 0;
}
```



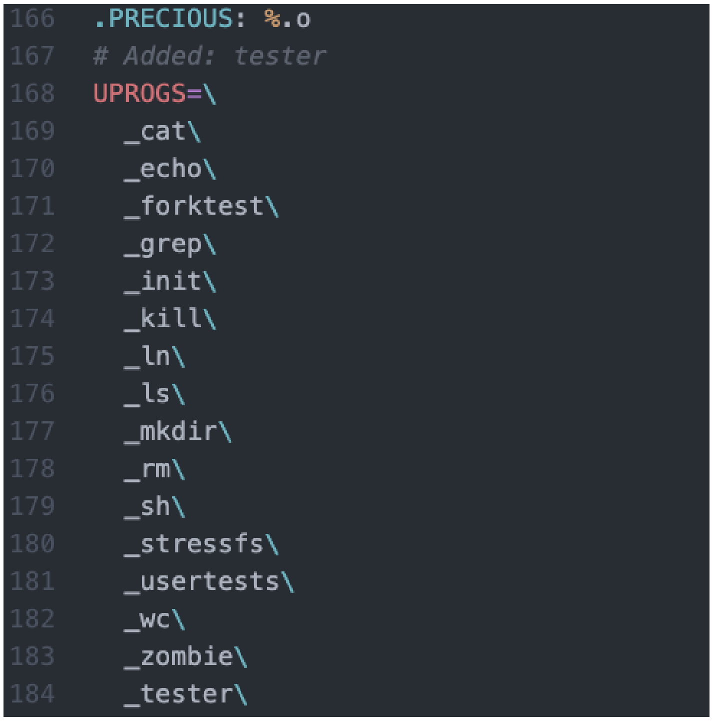
```c
/**
* Check if given PA (phys_addr)
* is linked with a shared memory page
*/
int is_PA_linked_with_sharedMemory(uint phys_addr){
  struct proc *p = myproc();
  for(int i = 0; i < (int)(sizeof(p->shared_memory) / sizeof(p->shared_memory[0])); i++) {
    for(int j = 0; j < regions[i].len; j++) {
      if(phys_addr == (uint)regions[i].phys_pages[j]) {
        return 1;
      }
    }
  }
  return 0;
}

/**
* Free pages when
* is_PA_linked_with_sharedMemory (are not shared)
* fails (returns 0)
*/
void forced_free_phys_page(void *phys_addr) {
  if(is_PA_linked_with_sharedMemory(V2P(phys_addr)) == 0){
    kfree(phys_addr);
   }
}
```

tester.c (create file)



```c
#include "types.h"
#include "user.h"

int main() {
  void* region = GetSharedPage(5, 3);
  for(int i = 0; i < 10; i++) {
    printf(1, "%d ", ((char*)region)[i]);
  }
  printf(1, "\n");

  // write
  strcpy(region, "region");

  // read
  printf(1, "%s\n", region);

  // FreeSharedPage(0);
  exit();
}
```

Makefile



```makefile
.PRECIOUS: %.o
# Added: tester
UPROGS=\
  _cat\
  _echo\
  _forktest\
  _grep\
  _init\
  _kill\
  _ln\
  _ls\
  _mkdir\
  _rm\
  _sh\
  _stressfs\
  _usertests\
  _wc\
  _zombie\
  _tester\
```