Assignment 3

Part 1: Memory leaks and tools to find them

Write a program that allocates memory using malloc() but forgets to free it before
exiting. What happens when this program runs? Can you use `gdb` to find any problems with it? How about `valgrind` (with the command: *valgrind --leak-check=yes null*)

The following program is a test case that allocates dynamic memory but forgets to free the memory. If it's run in shell, everything would look normal. However, regarding the memory leak, in general, modern general-purpose operating systems do clean up after terminated processes. But it also depends on the OS.

```
vagrant@vagrant:~/hw3$ cat memleak.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char** argv){
  char *str;
  str = (char *) malloc(20);
  //free(str);
  return 0;
}
```

Essentially, gcc can't be used to find memory leak. Even if gcc is run in instruction level, it's not easy to find the problem. For this toy example, we can count whether there's a complete pair of malloc and free, however, which is not practical for a real project.

```
main:
.LFB5:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $32, %rsp
        movl    %edi, -20(%rbp)
        movq    %rsi, -32(%rbp)
        movl    $20, %edi
        call    malloc@PLT
        movq    %rax, -8(%rbp)
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

On the other hand, valgrind is an excellent tool to find any memory leaks. From the following screenshot, we can see that there are 20 bytes of definitely lost, which is the exactly size that I malloc in the test case.

```
vagrant@vagrant:~/hw3$ valgrind --tool=memcheck --leak-check=full ./memleak
==2035== Memcheck, a memory error detector
==2035== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2035== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==2035== Command: ./memleak
==2035==
==2035==
==2035== HEAP SUMMARY:
==2035==     in use at exit: 20 bytes in 1 blocks
==2035==   total heap usage: 1 allocs, 0 frees, 20 bytes allocated
==2035==
==2035== 20 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2035==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==2035==    by 0x108662: main (memleak.c:7)
==2035==
==2035== LEAK SUMMARY:
==2035==    definitely lost: 20 bytes in 1 blocks
==2035==    indirectly lost: 0 bytes in 0 blocks
==2035==      possibly lost: 0 bytes in 0 blocks
==2035==    still reachable: 0 bytes in 0 blocks
==2035==         suppressed: 0 bytes in 0 blocks
==2035==
==2035== For counts of detected and suppressed errors, rerun with: -v
==2035== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Create other test cases for `valgrind`. Explain why you choose them and the expected results.

Test case – uninitialized memory

For this test case, the program accesses an uninitialized array and valgrind succeeds to detect such case. Furthermore, it is able to locate where the program uses that variable, which is expected for the purpose of this test case.

```
vagrant@vagrant:~/hw3$ cat uninit_mem.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv){
  int a[5];
  printf("%d\n", a[0]);

  return 0;
}
```

```
vagrant@vagrant:~/hw3$ valgrind --tool=memcheck --leak-check=no --track-origins=yes ./uninit_mem
==2048== Memcheck, a memory error detector
==2048== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2048== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==2048== Command: ./uninit_mem
==2048==
==2048== Conditional jump or move depends on uninitialised value(s)
==2048==    at 0x4E9A9DA: vfprintf (vfprintf.c:1642)
==2048==    by 0x4EA3015: printf (printf.c:33)
==2048==    by 0x1086DD: main (in /home/vagrant/hw3/uninit_mem)
==2048==  Uninitialised value was created by a stack allocation
==2048==    at 0x1086AA: main (in /home/vagrant/hw3/uninit_mem)
==2048==
==2048== Use of uninitialised value of size 8
==2048==    at 0x4E9696B: _itoa_word (_itoa.c:179)
==2048==    by 0x4E9A00D: vfprintf (vfprintf.c:1642)
==2048==    by 0x4EA3015: printf (printf.c:33)
==2048==    by 0x1086DD: main (in /home/vagrant/hw3/uninit_mem)
==2048==  Uninitialised value was created by a stack allocation
==2048==    at 0x1086AA: main (in /home/vagrant/hw3/uninit_mem)
```

Test case – indirectly lost

By assignment nbrs = NULL, we have just lost the last pointer to our memory data blocks. We should have freed the memory first, but we can't do it now because we don't have a pointer to the start of the data structure anymore. So this should be reported as an indirectly lost.

```
vagrant@vagrant:~/hw3$ cat indirectly_lost.c
#include <stdlib.h>

struct nbrs{
  int *nums;
};

struct nbrs *nbrs;

void allocate(struct nbrs **arr){
  *arr = malloc((sizeof(struct nbrs) * 3));
  arr[0]->nums = malloc(sizeof(int)*10);
}

int main(int argc, char** argv){
  allocate(&nbrs);
  nbrs = NULL;
  return 0;
}
```

```
vagrant@vagrant:~/hw3$ valgrind --tool=memcheck --leak-check=yes ./indirectly_lost
==2270== Memcheck, a memory error detector
==2270== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2270== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==2270== Command: ./indirectly_lost
==2270==
==2270==
==2270== HEAP SUMMARY:
==2270==     in use at exit: 64 bytes in 2 blocks
==2270==   total heap usage: 2 allocs, 0 frees, 64 bytes allocated
==2270==
==2270== 64 (24 direct, 40 indirect) bytes in 1 blocks are definitely lost in loss record 2 of 2
==2270==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==2270==    by 0x108660: allocate (in /home/vagrant/hw3/indirectly_lost)
==2270==    by 0x1086A1: main (in /home/vagrant/hw3/indirectly_lost)
==2270==
==2270== LEAK SUMMARY:
==2270==    definitely lost: 24 bytes in 1 blocks
==2270==    indirectly lost: 40 bytes in 1 blocks
==2270==      possibly lost: 0 bytes in 0 blocks
==2270==    still reachable: 0 bytes in 0 blocks
==2270==         suppressed: 0 bytes in 0 blocks
==2270==
==2270== For counts of detected and suppressed errors, rerun with: -v
==2270== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Test case – still reachable

The "still reachable" category within Valgrind's leak report refers to allocations that fit only the first definition of "memory leak". These blocks were not freed, but they could have been freed (if the programmer had wanted to) because the program still was keeping track of pointers to those memory blocks. This test case inserts 'abort()' between the 'malloc' and 'free', which trigger the 'still reachable' scenario.

```
vagrant@vagrant:~/hw3$ cat still_reachable.c
#include <stdio.h>
#include <stdlib.h>


int main(int argc, char** argv){
  char *str;
  str = malloc(sizeof(char)*10);
  abort();
  free(str);
  return 0;
}
```

```
vagrant@vagrant:~/hw3$ valgrind --tool=memcheck --leak-check=yes ./still_reachable
==2350== Memcheck, a memory error detector
==2350== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2350== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==2350== Command: ./still_reachable
==2350==
==2350==
==2350== Process terminating with default action of signal 6 (SIGABRT)
==2350==    at 0x4E7CFB7: raise (raise.c:51)
==2350==    by 0x4E7E920: abort (abort.c:79)
==2350==    by 0x1086AB: main (in /home/vagrant/hw3/still_reachable)
==2350==
==2350== HEAP SUMMARY:
==2350==     in use at exit: 10 bytes in 1 blocks
==2350==   total heap usage: 1 allocs, 0 frees, 10 bytes allocated
==2350==
==2350== LEAK SUMMARY:
==2350==    definitely lost: 0 bytes in 0 blocks
==2350==    indirectly lost: 0 bytes in 0 blocks
==2350==      possibly lost: 0 bytes in 0 blocks
==2350==    still reachable: 10 bytes in 1 blocks
==2350==         suppressed: 0 bytes in 0 blocks
==2350== Reachable blocks (those to which a pointer was found) are not shown.
==2350== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==2350==
==2350== For counts of detected and suppressed errors, rerun with: -v
==2350== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
Aborted (core dumped)
```

Part 2: System calls to share a memory page (40%)

The design of the system calls

1) Declare share page data structure

Record reference counts and the page number, as well as the physical address pointer.

```
+struct sharepages
+{
+    int refcount;
+    int page_num;
+    void *physaddr[MAX_SHP_PAGE_NUM];
+};
```

2) Setup OS for share page

- Add shp_init() in the main.c, which allow the OS to initialize the share pages.

```
diff --git a/main.c b/main.c
index 9924e64..c8aefba 100644
--- a/main.c
+++ b/main.c
@@ -35,6 +35,7 @@ main(void)
   kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
   userinit();      // first user process
   mpmain();        // finish this processor's setup
+  shp_init(); //init share pages
 }
```

- Initialize shpkemark and shpbounds in proc.c when new processes are found.

```
+  //init share mem
+  p->shpkeymark = 0;
+  p->shpbounds = KERNBASE;
```

- Clean up in exec.c when a process exits by releasing the share pages and resetting shpbounds and shpkeymark.

```
+  shp_release(curproc->pgdir, curproc->shpbounds, curproc->shpkeymark);
   freevm(oldpgdir);
+  curproc->shpbounds = KERNBASE;
+  curproc->shpkeymark = 0;
```

3) Share page management (vm.c)

- shp_init(): Initialize share pages by creating memory lock and setting reference count to 0.
- shp_alloc():
    1. Create the entire share pages by calling memset().
    2. The new share page space starts from newshp to old shp, and it is set to zero by default.
    3. Call mappages() to map the page directory to the physical addresses.
- shp_deallocvm(): Walk through the page directory and release the page tables.
- shp_map(): Loop over the share page space and create translations from virtual address to physical address in existing page table.

```
int shp_map(pde_t *pgdir, uint oldshp, uint newshp, uint sz, void **physaddr)
{
    if (oldshp & 0xFFF || newshp & 0xFFF || oldshp > KERNBASE || newshp < sz)
    {
        return 0;
    }
    uint a;
    a = newshp;
    for (int i = 0; a < oldshp; a += PGSIZE, i++)
    {
        mappages(pgdir, (char *)a, PGSIZE, (uint)physaddr[i], PTE_W | PTE_U);
    }

    return newshp;
}
```

- shp_add():
    1. Check the validity of key.
    2. Add a number of new physical page addresses to the share page table at the index of key.

```c
int shp_add(uint key, uint pagenum, void *physaddr[MAX_SHP_PAGE_NUM])
{
    if (key < 0 || MAX_SHP_TAB_NUM <= key || pagenum < 0 || MAX_SHP_PAGE_NUM < pagenum)
        return -1;

    shptab[key].refcount = 1;
    shptab[key].page_num = pagenum;
    for (int i = 0; i < pagenum; i++)
    {
        shptab[key].physaddr[i] = physaddr[i];
    }

    return 0;
}
```

- shp_release():
    1. Acquire the share page lock.
    2. Call shp_deallocvm() to deallocate page tables.
    3. Loop the whole share tables, if a share page's refcount is bigger than zero, decrease the reference count; if the reference count is zero, just free the share page.
    4. Release the share page lock.
- shp_key_used(): Detect if this page(referenced by key) is in use.

```c
int shp_key_used(uint key, uint mark)
{
    if (key < 0 || MAX_SHP_TAB_NUM <= key)
        return 0;

    return (mark >> key) & 0x1;
}
```

- free_shared_page():
    1. Get the share page by key.
    2. Then walk through its physical addresses and free them.
    3. Reset the reference count.

```c
int free_shared_page(int key)
{
    if (key < 0 || MAX_SHP_TAB_NUM <= key)
    {
        return -1;
    }

    struct sharepages *shp = &shptab[key];
    for (int i = 0; i < shp->page_num; i++)
    {
        kfree((char *)P2V(shp->physaddr[i]));
    }

    shp->refcount = 0;
    return 0;
}
```

- get_shared_page():
    1. Acquire share lock.
    2. Check if the share page of the current process is already in memory. If it is the case, just return the virtual address.
    3. If it hasn't been initialized regarding the key, call shp_alloc() to allocate the share page. Then save it into the process structure and add those physical addresses of the share page to the share table given a key.
    4. Otherwise, just get the page number and physical addresses by key, then save the page table to the process structure and increase the reference count.
    5. Update the share space bound and key mask.
    6. Release the share page lock.

3. Register functions in defs.h

```
diff --git a/defs.h b/defs.h
index 82fb982..d2ee100 100644
--- a/defs.h
+++ b/defs.h
@@ -184,7 +184,16 @@ pde_t*          copyuvm(pde_t*, uint);
 void            switchuvm(struct proc*);
 void            switchkvm(void);
 int             copyout(pde_t*, uint, void*, uint);
-void            clearpteu(pde_t *pgdir, char *uva);
+void            clearpteu(pde_t *, char *);
+int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);
+pte_t* walkpgdir(pde_t *pgdir, const void *va, int alloc);
+struct sharepages;
+void* get_shared_page(int, int);
+int free_shared_page(int);
+void shp_init();
+void shp_add_count(uint);
+int shp_key_used(uint, uint);
+int shp_release(pde_t *, uint, uint);
```

4. Add additional info in the proc structure.

```
+
+  uint shpbounds;               // Share memory boundary
+  uint shpkeymark;              // Share memory key mark
+  void* shpva[MAX_SHP_TAB_NUM]; // Share memory virtual address
```

5. Create system calls

Sys_get_shared_page:

- Get the first argument and assign to key
- Get the second argument and assign to num_pages
- If they are valid, call get_shared_page() to get share page pointer.

Sys_free_shared_page:

- Just get the first argument and assign to key
- Then call free_shared_page the free the shared page by key.

Test cases

Two processes, one for writing to share pages, another for reading from share pages. We can see the process with pid 0 succeeds to read the correct content which is written by the process with pid 4. Then Pid 4 can only read empty content after pid 0 frees the share pages.

```
int
main(int argc, char *argv[])
{
    int pid = fork();
    if (pid == 0)
    {
        sleep(10);
        char *shp = (char *)get_shared_page(1, 3);
        printf(1, "Child process pid:%d read share pages address:%x content:%s\n", getpid(), V2P(shp), shp);
        free_shared_page(1);
        printf(1, "Child process pid:%d free share pages\n", getpid());
    } else {
        char *shp = (char *)get_shared_page(1, 3);
        char *content = "hello";
        strcpy(shp, content);
        printf(1, "Parent process pid:%d write share pages address:%x content:%s\n", getpid(), V2P(shp), content);
        wait();
        printf(1, "Parent process pid:%d read share pages address:%x content:%s\n", getpid(), V2P(shp), shp);
    }
    exit();
}
```

```
$ test
Parent process pid:14 write share pages address:FFFFD000 content:hello
Child process pid:15 read share pages address:FFFFD000 content:hello
Child process pid:15 free share pages
Parent process pid:14 read share pages address:FFFFD000 content:
```