CS 450 PA2:
Yuyang Luo A20372496
Jiaqian Yan A20325624

Part (1) trace system call read:

When user call *malloc()*:

- *void* malloc(uint nbytes)* defined at **umalloc.c:64** first be called, in this function.
- Calculate the size of memory for a block include header and saved in local variable "nunits".
- Check the free list (which is circular linked list), if list is null (freep=null) just initial it as an empty list.
- Search in the free list to find the first block that not smaller than the target size. Split the founded block to the size we want (if any needed) and return the pointer point to the top of this block (not include Header).
- If no suitable block founded, call *morecore()* to get more heap space and add the new space in free list, then search again; if system cannot get more space, return 0.

Furthermore----------

- *static Header* morecore(uint nu)* defined in **umalloc.c:47**
- Set the target size larger than 4096, then call *sbrk()* to increase the heap size of nu*8 bytes;
  - sbrk is a syscall, defined in **usys.S:29**, then jump to kernel mode and start at **syscall.c:132**. The function *void syscall()* will handle which syscall to use by get information from current proc.
  - For sbrk, *sys_sbrk(void)* defined at **sysproc.c:46** will be called. First use *argint()* to get parameter (same way as pa2), if the parameter's value is smaller than 0, return -1 for error.
  - Save the current size of proc heap in variable addr, and call *growproc()* to increase the heap size.
    - *int growproc(int n)* defined in **proc.c:159**,
    - First het process state by using *myproc()*;(same as pa2).
    - Then check the size change: if change > 0, call *allocuvm()* to increase the heap size; else if change <0, use *deallocuvm()* to decrease the heap size. If success, these two functions will return new size of process heap, or return -1 for failed.
      - In *int allocuvm(pde_t *pgdir, uint oldsz, uint newsz)* defined in **vm.c:221**
      - Check the newsize: if it is larger, then KERNBASE. return -1 for error; if newsize < oldsize, just return old size.
      - Use the oldsize and macro PGROUNDUP to get the last page address in current process.
      - In a for loop, just try to add new pages in current heap:
        - In loop, first call *char* kalloc(void)* defined at **kalloc.c:84** to get an address of 4096-byte page of physical memory. It just tries to pick up the first page in the freelist of physical memory and remove it.
        - If kalloc failed, print error message and return 0 for error.
        - Use memset to make the new page empty.
        - Use mapepages to create PTEs for new page
          - *static int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)* defined in **vm.c:61**
          - Use macro PGROUNDDOWN, the given virtual address to get the range of the new virtual address for new pages,
          - For each page call *walkpgdir()* defined in **vm.c:36** with last parameter as 1 to

create PTE. In *walkpgdir()*, first get the indexes for multiple layer page tables, as the past parameter alloc is 1, the secondary table will be generate if the secondary layer table does not exist. Then, this function returns the address of PTE if success or 0 for failed.

- After get PTE address, check its usability, if the pte is in use, print error message using *panic()*;
- Put the physical address and mark bits into pte.
- This function returns -1 for error or 0 for success.

◆ If *mappages()* failed, use *kfree()* defined in **kalloc.c:61** to re-add the page into free list of physical memory.

◆ If any one of kalloc of mappages return error, it needs deallocuvm to change the heap size back, and return 0 for error.

◆ Return new size of proc heap.

■ For *int deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)* defined at **vm.c 255**, it's just try to free PTEs between the virtual address of oldsize and newsize.

- If the "% allocuvm" success, use switchuvm defined in **vm.c:157** to finish TSS operation and load page table into cr3 register. Then return 0.
- If failed return -1 for error.

◆ If *growproc()* success (return value > 0), return the addr; else return -1 for error.

◆ Syscall finished, return to user mode.

■ Get the address from *sbrk()*, and build a header into this block,

■ Use *free()* to add this block into process free list.

■ If sbrk failed (return -1), return 0 for error; else return the freep.

Get back to malloc----------

- If *morecore()* success, search the free list again; else return 0 for error.
- Malloc finished.

When user call *free()*:

- First get the header from the given virtual address.
- Then search the free list in a for loop to find the place for new free block that keeps the address increase.
- After getting the right position for the new free block, check the block which is after it; if the two blocks are next to each other, merge them together.
- Check the block before it: if they are next to each other, merge them.
- Set the head of the free list to where this block is.

Part (2) implement of system call uv2p() on XV6

File modify:

```
--- defs.h_ 2017-09-26 22:52:08.000000000 -0500
+++ defs.h  2017-10-27 15:09:41.801600000 -0500
@@ -121,6 +121,7 @@
 void            wakeup(void*);
 void            yield(void);
 int             procState(void);
+uint            uv2p_(char*);

 // swtch.S
 void            swtch(struct context**, struct context*);
@@ -186,6 +187,7 @@
 void            switchkvm(void);
 int             copyout(pde_t*, uint, void*, uint);
 void            clearpteu(pde_t *pgdir, char *uva);
+uint            uva2pa(pde_t*,char*);

 // number of elements in fixed-size array
 #define NELEM(x) (sizeof(x)/sizeof((x)[0]))
```

```
--- proc.c_ 2017-10-03 08:09:56.000000000 -0500
+++ proc.c  2017-10-27 15:16:45.377090000 -0500
@@ -560,3 +560,8 @@
   release(&ptable.lock);
   return 0;
 }
+
+uint uv2p_(char *uva){
+  struct proc *curproc = myproc();
+  return uva2pa(curproc->pgdir, uva);
+}
\ No newline at end of file
```

```
--- syscall.h_  2017-09-26 22:34:34.000000000 -0500
+++ syscall.h   2017-10-27 15:21:11.547234000 -0500
@@ -21,3 +21,4 @@
 #define SYS_mkdir  20
 #define SYS_close  21
 #define SYS_ps     22
```

```
+#define SYS_uv2p   23
```

```
--- syscall.c_  2017-10-03 07:50:24.000000000 -0500
+++ syscall.c   2017-10-27 15:26:52.783602000 -0500
@@ -104,6 +104,7 @@
 extern int sys_write(void);
 extern int sys_uptime(void);
 extern int sys_ps(void);
+extern int sys_uv2p(void);

 static int (*syscalls[])(void) = {
 [SYS_fork]    sys_fork,
@@ -128,6 +129,7 @@
 [SYS_mkdir]   sys_mkdir,
 [SYS_close]   sys_close,
 [SYS_ps]      sys_ps,
+[SYS_uv2p]    sys_uv2p,
 };

 void
```

```
--- sysproc.c_  2017-09-26 22:37:34.000000000 -0500
+++ sysproc.c 2017-10-27 15:26:51.138000000 -0500
@@ -56,6 +56,15 @@
   return addr;
 }

+uint sys_uv2p(void)
+{
+  int addr;
+
+  if(argint(0, &addr) < 0)
+    return -1;
+  return uv2p_((char*)addr);
+}
+
 int
 sys_sleep(void)
 {
```

```
--- usys.S_ 2017-09-26 22:42:04.000000000 -0500
+++ usys.S  2017-10-27 15:22:51.947770000 -0500
@@ -30,3 +30,4 @@
 SYSCALL(sleep)
 SYSCALL(uptime)
 SYSCALL(ps)
\ No newline at end of file
+SYSCALL(uv2p)
\ No newline at end of file
```

```
--- vm.c_ 2017-08-23 00:40:36.000000000 -0500
+++ vm.c  2017-11-07 20:29:32.456468700 -0600
@@ -356,6 +356,25 @@
   return (char*)P2V(PTE_ADDR(*pte));
 }

+
+
+uint uva2pa(pde_t *pgdir,char *uva)
+{
+  if((uint)uva>0x3FFFFF)
+    return -3;
+  int offset = 0xfff&((uint)uva);
+  uva = (char*)PGROUNDDOWN((uint)uva);
+  pte_t *pte;
+  pte = walkpgdir(pgdir, uva, 0);
+  if(pte==0)
+    return -2;
+  if((*pte & PTE_P) == 0)
+    return -1;
+  if((*pte & PTE_U) == 0)
+    return -1;
+  return offset + PTE_ADDR(*pte);
+}
+
 // Copy len bytes from p to user address va in page table pgdir.
 // Most useful when pgdir is not the current page table.
 // uva2ka ensures this only works for PTE_U pages.
```

```
--- Makefile_   2017-09-26 22:46:24.000000000 -0500
+++ Makefile    2017-10-27 15:24:24.056366000 -0500
@@ -177,6 +177,7 @@
    _sh_xv\
    _test\
    _ps\
+   _testup\

 fs.img: mkfs README $(UPROGS)
    ./mkfs fs.img README $(UPROGS)
```

- All modify information saved in /diff;
- This PA is working on the XV6 system which includes ps function (PA2), so some column numbers may not match the original XV6.
- Old files are renamed with an additional charter '_' at the end.

Description (How is system call uv2p work?):

When *uv2p* is called in a user program, something like save the parameter in program stack, jump to kernel mode and kernel read parameter from stack happened, which are just like other system calls that we traced. Kernel will run *sys_uv2p*. which reads the parameter and runs *uv2p_(char*)* defined in *proc.c*.

uv2p_(char)* defined in *proc.c* just get the process states by call *myproc()* and get the *pgdir* of the running process. Then past the user virtual address and the *pgdir* to *uva2pa (pde_t *,char*)* defined in *vm.c*

In *uva2pa*, first use *0xfff&((uint)uva)* to get the right most 12 bits of the virtual address, which is the offset of a physical address in the page. Then use the macro PGROUNDDOWN to get the address of the first byte in the page that associate with the given virtual address.
Then use the XV6 system function *walkpgdir* to get the address of target PTE, which includes the physical address of target page, and that address could be calculated by using the macro *PTE_ADDR*. And finally check the PTE address protected bits; if it passes the check, returns the *offset+PTE_ADDR* which is the physical address of the given virtual address, or return -1 if the PTE is not available.

Major translation happened in *walkpgdir()*:

```
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
```

```
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)];
}
```

As we can see, first we use macro PDX to get the page directory index from virtual address, which is the left most 10 bits of virtual address. Then we use the page directory index and the page directory table of current table to the PDE address, and check if this PDE is available. If PDE is unavailable, returns 0 for error, because we past 0 for the parameter alloc that mean do not change the page directory elements. Finally, we use the address of 2nd layer page table that get from PDE and the page table index that calculated with the macro PTX and virtual address to get the target page table entry address and return it.

Exception handling

There will be three types of error for the address translation,

- Virtual address illegal (more than 22 bits, the max length of a virtual address in XV6)
- PDE does not present (walkpgdir() return 0)
- PTE does not present or cannot access (PTE_P or PTE_U check failed).

For each kind of error, uva2pa(the final implement of uv2p) will return:

- -3 (0xFFFFFFFD) for virtual address illegal.
- -2 (0xFFFFFFFE) for PDE does not present.
- -1 (0xFFFFFFFF) for PTE does not present or cannot access.

Test Result:

Implementation of test program: “testup.c”:

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

int bss;
int data_ =11;
const int txt=12;
```

```c
void testS (int p){
    printf(2,"------------------va from stack\n");
    printf(2,"0x%x\t0x%x\n",&p,uv2p((char*)(&p)));
}
int main()
{
    int * test = (int*)malloc(sizeof(int));
    printf(2,"VA\tPA\n");
    printf(2,"------\t----------\n");
    printf(2,"------------------va from heap\n");
    printf(2,"0x%x\t0x%x\n",test,uv2p((char*)test));
    printf(2,"------------------va doesn't exist\n");
    printf(2,"0x%x\t0x%x\n",test+10,uv2p((char*)(test+10)));
    int * test1 = (int *)malloc(sizeof(int)*1000);
    printf(2,"------------------contiune va in heap\n");
   printf(2,"0x%x\t0x%x\n",test1,uv2p((char*)test1));
    printf(2,"0x%x\t0x%x\n",test1+1,uv2p((char*)(test1+1)));
    printf(2,"0x%x\t0x%x\n",test1+10,uv2p((char*)(test1+10)));
    printf(2,"0x%x\t0x%x\n",test1+100,uv2p((char*)(test1+100)));
    printf(2,"0x%x\t0x%x\n",test1+1000,uv2p((char*)(test1+1000)));
    printf(2,"0x%x\t0x%x\n",test1+1001,uv2p((char*)(test1+1001)));

    printf(2,"------------------contiune va in stack\n");
    int xx=13,x[]={1,2,3};
    double y[]={1.1,2.2,3.3};
    printf(2,"0x%x\t0x%x\n",&x,uv2p((char*)&x));
    printf(2,"------------------type int\n");
    printf(2,"0x%x\t0x%x\n",&xx,uv2p((char*)&xx));
    printf(2,"0x%x\t0x%x\n",&x[0],uv2p((char*)&x[0]));
    printf(2,"0x%x\t0x%x\n",&x[1],uv2p((char*)&x[1]));
    printf(2,"0x%x\t0x%x\n",&x[2],uv2p((char*)&x[2]));
    printf(2,"------------------type double\n");
    printf(2,"0x%x\t0x%x\n",&y[0],uv2p((char*)&y[0]));
    printf(2,"0x%x\t0x%x\n",&y[1],uv2p((char*)&y[1]));
    printf(2,"0x%x\t0x%x\n",&y[2],uv2p((char*)&y[2]));
    printf(2,"------------------type pointer\n");
    printf(2,"0x%x\t0x%x\n",&test,uv2p((char*)&test));
    printf(2,"------------------type main function\n");
    printf(2,"0x%x\t0x%x\n",main,uv2p((char*)&main));
    printf(2,"------------------type user function\n");
    printf(2,"0x%x\t0x%x\n",testS,uv2p((char*)&testS));
    printf(2,"------------------type system function\n");
    printf(2,"0x%x\t0x%x\n",free,uv2p((char*)&free));
    printf(2,"0x%x\t0x%x\n",malloc,uv2p((char*)&malloc));
    printf(2,"------------------type syscall\n");
    printf(2,"0x%x\t0x%x\n",ps,uv2p((char*)&ps));
    printf(2,"0x%x\t0x%x\n",uv2p,uv2p((char*)&uv2p));
    printf(2,"------------------va from .bss\n");
    printf(2,"0x%x\t0x%x\n",&bss,uv2p((char*)&bss));
```

```
    printf(2,"-------------------va from .data\n");
    printf(2,"0x%x\t0x%x\n",&data_,uv2p((char*)&data_));
    printf(2,"-------------------va from .txt\n");
    printf(2,"0x%x\t0x%x\n",&txt,uv2p((char*)&txt));
    printf(2,"-------------------va 0x3FFFFF\n");
    printf(2,"0x%x\t0x%x\n",0x3FFFFF,uv2p((char*)0x3FFFFF));
    printf(2,"-------------------va 0x400000\n");
    printf(2,"0x%x\t0x%x\n",0x400000,uv2p((char*)0x400000));
    printf(2,"-------------------va 0x4FFFFF\n");
    printf(2,"0x%x\t0x%x\n",0x4FFFFF,uv2p((char*)0x4FFFFF));
    testS(10);
    free(test);
    free(test1);
    exit();
}
```

Run “testup“ in XV6

```
$ testup
VA      PA
------  ----------
-------------------va from heap
0xBFF8  0xDFC4FF8
-------------------va doesnt exist
0xC020  0xFFFFFFFF
-------------------contiune va in heap
0xB050  0xDFC4050
0xB054  0xDFC4054
0xB078  0xDFC4078
0xB1E0  0xDFC41E0
0xBFF0  0xDFC4FF0
0xBFF4  0xDFC4FF4
-------------------contiune va in stack
0x3F9C  0xDEDEF9C
-------------------type int
0x3F98  0xDEDEF98
0x3F9C  0xDEDEF9C
0x3FA0  0xDEDEFA0
0x3FA4  0xDEDEFA4
-------------------type double
0x3FA8  0xDEDEFA8
0x3FB0  0xDEDEFB0
0x3FB8  0xDEDEFB8
-------------------type pointer
0x3F94  0xDEDEF94
-------------------type main function
0x0     0xDEE2000
-------------------type user function
0x4D0   0xDEE24D0
```

```
------------------type system function
0xA50   0xDEE2A50
0xAE0   0xDEE2AE0
------------------type syscall
0x7F2   0xDEE27F2
0x7FA   0xDEE27FA
------------------va 0x3FFFFF
0x3FFFFF        0xFFFFFFFF
------------------va 0x400000
0x400000        0xFFFFFFFD
------------------va 0x4FFFFF
0x4FFFFF        0xFFFFFFFD
------------------va from .bss
0x1184  0xDEE0184
------------------va from .data
0x1174  0xDEE0174
------------------va from .txt
0xE88   0xDEE2E88
------------------va from stack
0x3F80  0xDEDEF80
```

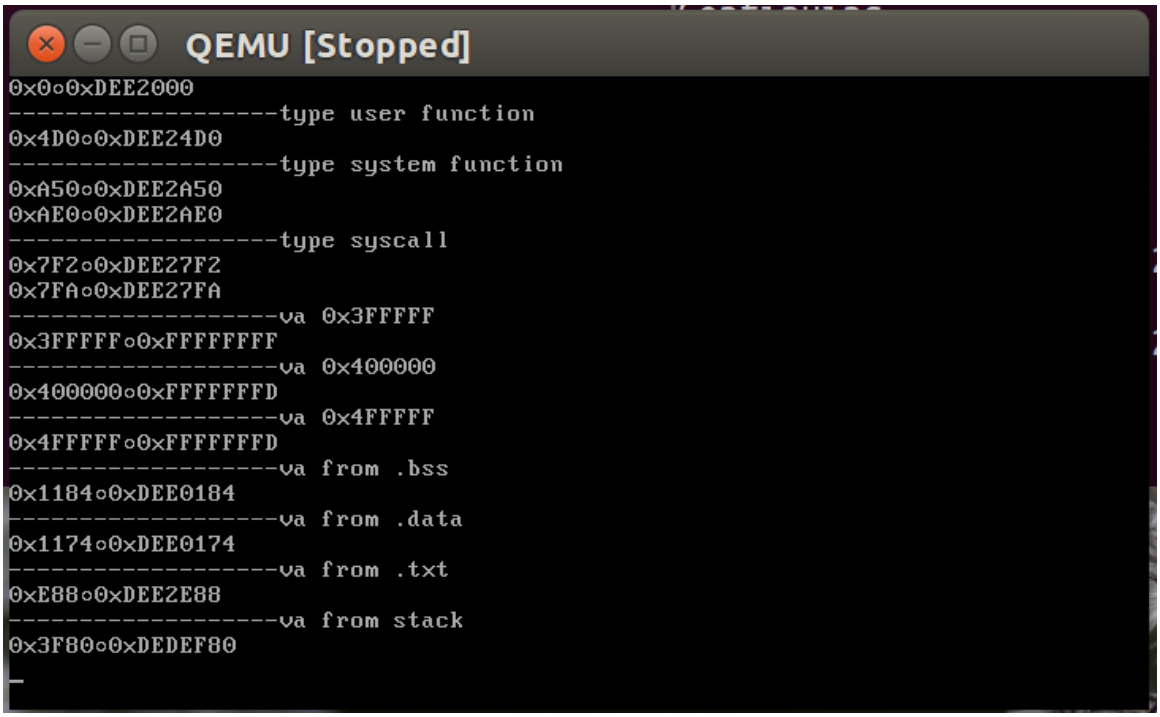
In this test program, we test virtual address that point different part of the memory, includes heap, stack, .data, .text and system call. We can find that the virtual address with different type has different page directory index+ page table index, like *heap* is 0xB, *.data* and *.bss* is 0x1, *.txt* is 0x0 and *stack* is 0x3; and for the translation result, va with different PDE+PTE are in different physical page, and the continue va also translate to continue pa.

And for any PTE does not present, uv2p return -1 (like 0x3FFFFF tranced to 0xFFFFFFFF)

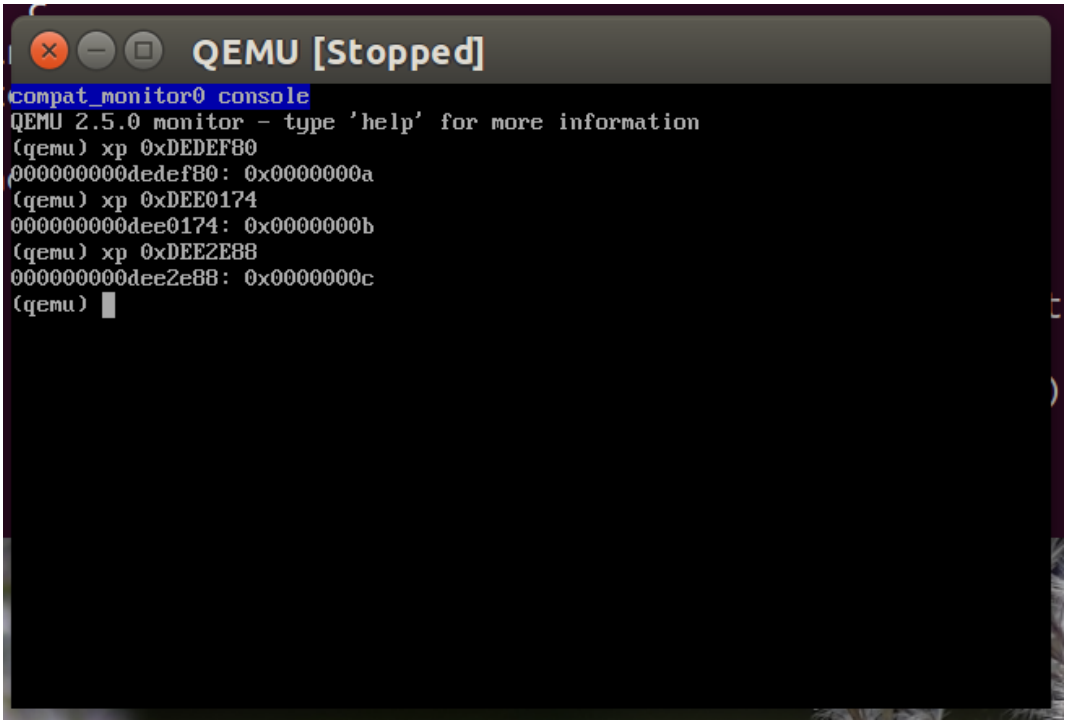
for any address with length bigger than 22 bits, return -3 (0x400000 & 0x4FFFFF both tranced to 0xFFFFFFFD) does not meet the situation that PDE miss.

Verify PA's correctness outside XV6:

QEMU has monitor mode which could get the context of a given physical address. When we ran the test program in xv6, we got the following result;



And as the code showed upside, we know that the value of 0xDEDEF80 should be 10 (0xA), 0xDEE2E88 should be 12 (0xC), and 0xDEE0174 should be 11 (0xB), and use command xp pa in QEMU monitor to verify the values.



The values of each variable were matched, so the translations should be correct.
(use qemu to verify the pa need remove "-serial mon:stdio" from the qemu parameter, and need use gdb to set break point before the user program exit. This modify did not show in the submitted xv6 source code.)