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  2017-10-27 16:52:36.000296000 -0500

@@ -356,6 +356,21 @@

return (char\*)P2V(PTE\_ADDR(\*pte));

}

+

+

+uint uva2pa(pde\_t \*pgdir,char \*uva)

+{

+ int offset = 0xfff&((uint)uva);

+ uva = (char\*)PGROUNDDOWN((uint)uva);

+ pte\_t \*pte;

+ pte = walkpgdir(pgdir, uva, 0);

+ 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.

## 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.

## Test & Result:

Test program: testup.c:

#include "types.h"

#include "stat.h"

#include "user.h"

int bss;

int data\_ =11;

const int txt=12;

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));

    testS(10);

    free(test);

    free(test1);

exit();

}

Result

$ testup

VA PA

------ ----------

-------------------va from heap

0xBFF8 0xDFF8FF8

-------------------va doesnt exist

0xC020 0xFFFFFFFF

-------------------contiune va in heap

0xB050 0xDFF8050

0xB054 0xDFF8054

0xB078 0xDFF8078

0xB1E0 0xDFF81E0

0xBFF0 0xDFF8FF0

0xBFF4 0xDFF8FF4

-------------------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

0x440 0xDEE2440

-------------------type system function

0x9C0 0xDEE29C0

0xA50 0xDEE2A50

-------------------type syscall

0x762 0xDEE2762

0x76A 0xDEE276A

-------------------va from .bss

0x1094 0xDEE0094

-------------------va from .data

0x1084 0xDEE0084

-------------------va from .txt

0xD98 0xDEE2D98

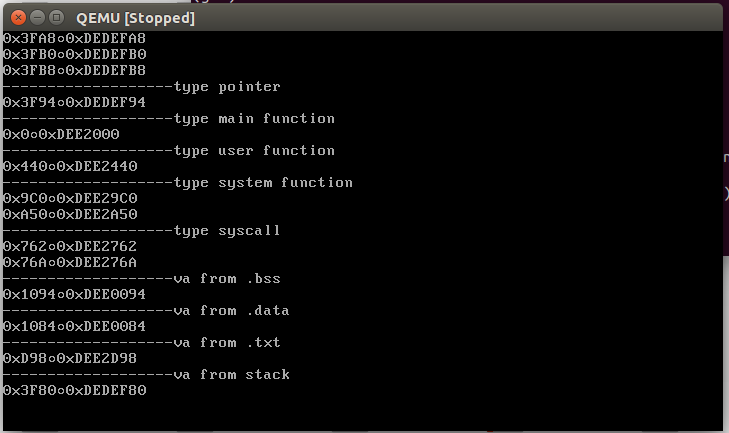
-------------------va from stack

0x3F80 0xDEDEF80

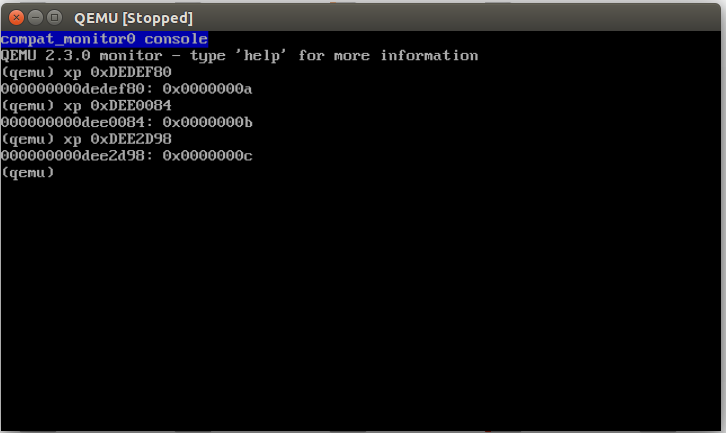
In this test program, we test virtual address that point different part of the memory, includes heap, stack, .data, .text and system call. We can find that the virtual address with different type has different page directory index+ page table index, like *heap* is 0xB, .*data* and *.bss* is 0x1, *.txt* 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.

#### 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), 0xDEE2D98 should be 12 (0xC), and 0xDEE0084 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 command line, and need gdb to ste break point before the user program exit. This modify did not show in the submitted xv6 source code.)