Memory-Management

CS3500 Operating Systems Jul-Nov 2025

Sanjeev Subrahmaniyan S B

EE23B102

1 Objective

The objective of this lab is to:

1. Print the page table entries, along with their hierarchy and the mapping between page table entry index, page table entry and the associated physical address when the first user process(the /init process) is run.
2. Add a utility to speed up the getpid() system call by mapping a common page of memory between the user and kernel address spaces, and store the process ID there. This is inspired by the system used for the trampoline page used to switch between the kernel and user spaces.
3. Add a pgaccess system call to identify the list of pages that have been accessed in a given range. The range is specified by providing the base address of the pages to be checked and a count of pages(which are after the base address and will be checked). A bit corresponding to the page number is set when the page has been accessed.

2 Printing Page Tables

2.1 Code

The changes made to implement the page table printing are made in the following files. The changes are also briefly justified.

2.1.1 kernel/exec.c

Inside the exec() function, which is used to execute a new process, I made a modification to create a custom string compare function to check if the process being executed is the /init process and then call the vmprint() function to print the page table.

```
// {...}
   oldpagetable = p->pagetable;
   p->pagetable = pagetable;
   p->sz = sz;
// {CHANGES BEGIN HERE}
  // this location is chosen because the memory image is done writing
  // section of code to check if this is the init
  char *target = "/init";
  int flag = 1;

  // custom implementation of strcmp
  for(int i = 0; path[i] && target[i]; i++){
    if(path[i] != target[i]){flag = 0; break;}
  }
  if(flag) vmprint(pagetable);
  // page table printing ends here
```

```
17
18 // {CHANGES END HERE}
19
20    p->trapframe->epc = elf.entry;  // initial program counter = main
21    p->trapframe->sp = sp; // initial stack pointer
22    proc_freepagetable(oldpagetable, oldsz);
23 // {...}
```

2.1.2 kernel/vm.c

The vmprint() function is implemented here.

```
1  // {CHANGES BEGIN HERE}
2  #ifdef LAB_PGTBL
3  void vmprint(pagetable_t pagetable){
4    // static runs only one time
5
6    static int level = 2; // level of the pagetable we are at
7    if(level == 2)printf("page table %p\n", pagetable);
8
9    // search each of the entry in the page table
10   for(int i = 0; i < 512; i++){
11     pte_t entry = pagetable[i];
12     // check if the pte is valid
13      if(entry & PTE_V){
14       uint64 pa = PTE2PA(entry);
15
16       // children are valid page tables
17       if(level == 2){
18         printf("..%d: pte %p pa %p\n", i, (void *)entry, (void *)pa);
19       }
20       else if(level == 1){
21         printf(".. ..%d: pte %p pa %p\n", i, (void *)entry, (void *)pa);
22       }
23       else{
24         printf(".. .. ..%d: pte %p pa %p\n", i, (void *)entry, (void *)pa);
25       }
26       if(level != 0 && (entry & (PTE_W | PTE_R | PTE_X)) == 0){
27         level--;
28         vmprint((pagetable_t)pa);
29         level++;
30       }
31     }
32   }
33 }
34 #endif
35 // {CHANGES END HERE}
```

2.2 Implementation Explanation

Here the pages are recursively checked if their valid bit is set. If the bit is set, the page is printed in the format specified in the handout, and the vmprint function is recursively called on the next level page that is referenced by the current page table entry. I have also utilised a static int level which preserves its value between calls and allows to make the indented prints for the different levels of the page table. The traversal of the pages is

identical to the freewalk() function: start from the base of the page table, check the page table entry, recurse into the next level page table if valid, otherwise continue on the same level.

2.3 Execution and Output

When the command "make qemu" is run in the docker container with the new repository, the following output is seen, showing the page table hierarchy being printed before the first process is run:

```
root@4c2cf3b9963f:/home/os-iitm/xv6-riscv# make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -global virtio-mmio.force-
legacy=false -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
page table 0x0000000087f4d000
 ..0: pte 0x0000000021fd2401 pa 0x0000000087f49000
 .. ..0: pte 0x0000000021fd2001 pa 0x0000000087f48000
 .. .. ..0: pte 0x0000000021fd281b pa 0x0000000087f4a000
 .. .. ..1: pte 0x0000000021fd1c17 pa 0x0000000087f47000
 .. .. ..2: pte 0x0000000021fd1807 pa 0x0000000087f46000
 .. .. ..3: pte 0x0000000021fd1417 pa 0x0000000087f45000
 ..255: pte 0x0000000021fd3001 pa 0x0000000087f4c000
 .. ..511: pte 0x0000000021fd2c01 pa 0x0000000087f4b000
 .. .. ..509: pte 0x0000000021fd5413 pa 0x0000000087f55000
 .. .. ..510: pte 0x0000000021fd5807 pa 0x0000000087f56000
 .. .. ..511: pte 0x000000002000180b pa 0x0000000080006000
init: starting sh
$
```

3 Speeding up getpid()

3.1 Code

The changes made for this part of the assignment are in the following files:

3.1.1 kernel/proc.h

Inside the struct proc, it is required to add the usyscall struct as one of the elements for the process to be able to access it.

```
1 struct proc {
2     struct spinlock lock;
3 // ...
4 // {CHANGES BEGIN HERE}
5 struct usyscall *usys;
6 // {CHANGES END HERE}
7 // ...
8 };
```

3.1.2 kernel/proc.c

This is the file containing the code for allocation of processes and their page tables, freeing the memory associated with those processes and such. There are a number of changes made in this file:

```
1 static struct proc*
2 allocproc(void)
```

```
{
// ...
found:
// ...
if((p->trapframe = (struct trapframe *)kalloc()) == 0){
freeproc(p);
release(&p->lock);
return 0;
}

// {CHANGES BEGIN HERE}
// allocate the page for usyscall
if((p->usys = (struct usyscall *)kalloc()) == 0){
freeproc(p);
release(&p->lock);
return 0;
}
// clear the memory and write the usys struct
memset(p->usys, 0, PGSIZE);
p->usys->pid = p->pid;
// {CHANGES END HERE}

// An empty user page table.
p->pagetable = proc_pagetable(p);
if(p->pagetable == 0){
freeproc(p);
release(&p->lock);
return 0;
}
}
```

Here, a page is allocated for storing the struct usys, just as is done for the trapframe page and the empty user page table following it. This is where the struct will be stored, and the pid is stored in the struct.

```
static void
freeproc(struct proc *p)
{
  if(p->trapframe)
    kfree((void*)p->trapframe);
  p->trapframe = 0;
  if(p->pagetable)
    proc_freepagetable(p->pagetable, p->sz);
// {CHANGES BEGIN HERE}
  // free and zero out the usys entry in the proc struct
  if(p->usys){
    kfree((void*)p->usys);
  }
  p->usys = 0;
// {CHANGES END HERE}
  p->pagetable = 0;
  p->sz = 0;
  p->pid = 0;
  p->parent = 0;
  p->name[0] = 0;
  p->chan = 0;
  p->killed = 0;
  p->xstate = 0;
  p->state = UNUSED;
}
```

When the process is being freed, all the pages mapped to the process must also be freed. In a manner similar to which the user page table and the trapframe pages are freed, the usys entry's page table is also freed, and the corresponding entry in the proc struct is zeroed out so another process can later use the struct without a problem.

```
pagetable_t
proc_pagetable(struct proc *p)
{
  pagetable_t pagetable;

  pagetable = uvmcreate();
  if(pagetable == 0)
    return 0;

  if(mappages(pagetable, TRAMPOLINE, PGSIZE,
              (uint64)trampoline, PTE_R | PTE_X) < 0){
    uvmfree(pagetable, 0);
    return 0;
  }

  if(mappages(pagetable, TRAPFRAME, PGSIZE,
              (uint64)(p->trapframe), PTE_R | PTE_W) < 0){
    uvmunmap(pagetable, TRAMPOLINE, 1, 0);
    uvmfree(pagetable, 0);
    return 0;
  }
// {CHANGES BEGIN HERE}
  // map the usys page in user space with one read and user perms
  if(mappages(pagetable, USYSCALL, PGSIZE,
              (uint64)(p->usys), PTE_R | PTE_U) < 0){
  uvmunmap(pagetable, TRAMPOLINE, 1, 0);
  uvmunmap(pagetable, TRAPFRAME, 1, 0);
  return 0;
  }
// {CHANGES END HERE}
  return pagetable;
}
```

After the page table has been allocated, it has to be mapped to the corresponding physical address, which is USYSCALL for the usys page. This is performed inside the proc_pagetable(), which also maps the page tables for the trampoline and trapframe. Note that appropriate error actions(unmapping other pages) are added when the mapping fails. Also note that the permissions are set to only R and U, so that user processes can access and read there pages.

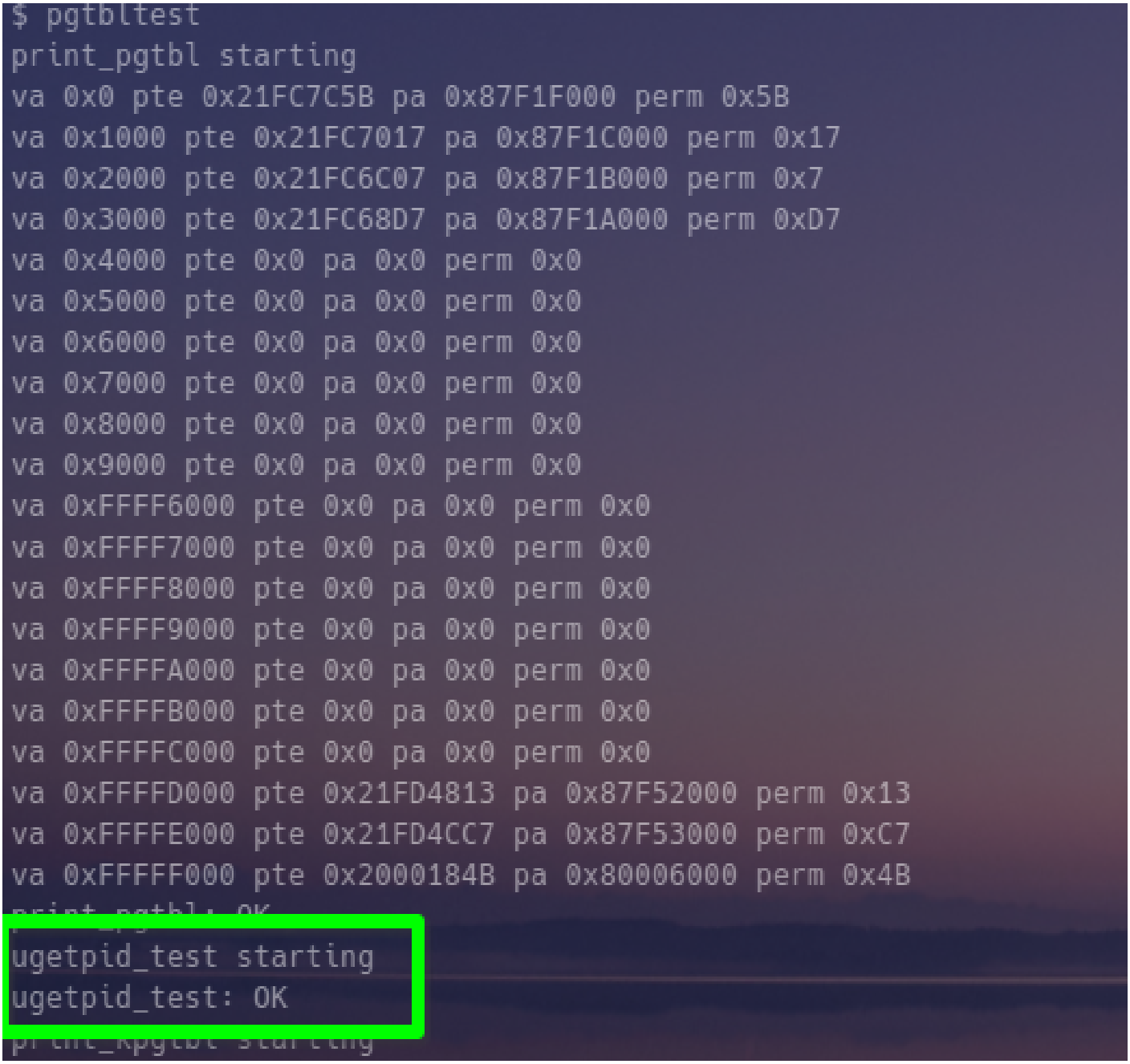
```
void
proc_freepagetable(pagetable_t pagetable, uint64 sz)
{
  uvmunmap(pagetable, TRAMPOLINE, 1, 0);
  uvmunmap(pagetable, TRAPFRAME, 1, 0);
// {CHANGES BEGIN HERE}
  uvmunmap(pagetable, USYSCALL, 1, 0);
// {CHANGES END HERE}
  uvmfree(pagetable, sz);
}
```

When the page tables that are allocated to a process are freed, the one for usyscall must also be done. This is

done in a manner identical to that for trampoline and trapframes.

3.2 Execution and Output

The code passes the tests given as part of the code mentioned in the handout.



4 Page Access Checks

4.1 Code

Changes were made to the following files to complete this part of the assignment.

4.1.1 kernel/defs.h

```
int                 pgaccess(void*, int, void*);
```

The function prototype as given in the handout is added to this header file.

4.1.2 kernel/riscv.h

```
#define PTE_A (1L << 6) // used to check if the page has been accessed
```

The mask to extract the page Accessed bit is defined in this header file. This is known by looking at the RISCV detailed specification and the xv6 book. The idea is that when the page table entry is logically anded with this mask, the result conveys if the bit was initially set or not.

4.1.3 kernel/syscall.c

```
// ...
extern uint64 sys_pageaccess(void);
// ...
static uint64 (*syscalls[])(void) = {
// ...
[SYS_pgaccess] sys_pgaccess,
// ...
}
```

These entries are made to allow the syscall handler call the appropriate system call by examining the arguments in the registers when an ecall is made. These changes are the same when any new system call is added to the xv6 operating system.

4.1.4 kernel/sysproc.c

```
uint64
sys_pgaccess(void)
{
  uint64 va;
  argaddr(0, &va);

  int pgcount;
  argint(1, &pgcount);

  uint64 buffer;
  argaddr(2, &buffer);

  return pgaccess((void *)va, pgcount, (void *)buffer);
}
```

A wrapper function is required to copy the arguments from the user space to kernel space and call the function used to handle the system call. These argument copies are done with the argint() and argaddr() functions, and the pgaccess() function is called with these copied arguments.

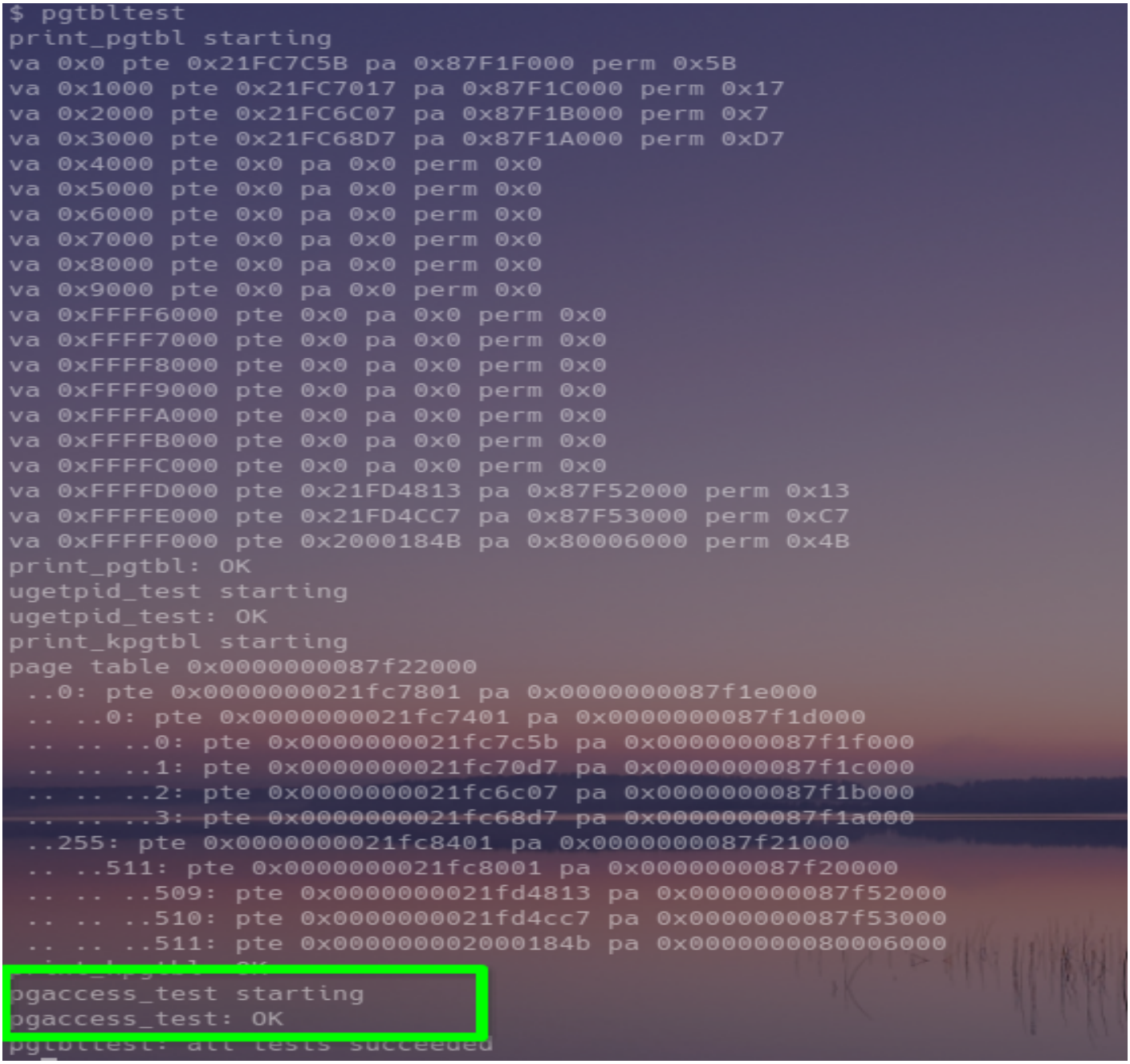
4.1.5 kernel/vm.c

```c
int pgaccess(void *va, int n, void *buf){
  uint64 mask = 0;
  pte_t *entry;
  struct proc *p = myproc();

  if(n > 64) return -1; // limit on number of max scannable pages

  for(int i = 0; i < n; i++){
    entry = walk(p->pagetable, (uint64)va + i * PGSIZE, 0);
    if(entry == 0) continue; // this would mean the page is not mapped

    // when page is mapped, check for accessed
    if(*entry & PTE_A){
      mask |= (1 << i); // setting the corresponding mask bit nicely

      // clear the accessed bit to prevent forever loops
      *entry &= ~(PTE_A);
    }
  }
  if(copyout(p->pagetable, (uint64)buf, (char *)&mask, sizeof(mask)) < 0){
    return -1;
  }

  return 0;
}
```

The function is implemented in this file. The idea used is a simple for loop to iterate over all the possible pages in the given ranges. The page table entry corresponding to a virtual address is obtained using the walk() function. If such a page table entry is found to be valid, its Accessed bit is tested by using the mask defined in riscv.h. If the page is found to be accessed, the corresponding bit is set in the mask register(which will be returned to the user through a safe copy) and the accessed flag is reset. Additionally, a check is performed on the number of pages passed as arguments to prevent problems.

4.2 Execution and Output

The code passes the tests given in the testing file.


$ pgtbltest
print_pgtbl starting
va 0x0 pte 0x21FC7C5B pa 0x87F1F000 perm 0x5B
va 0x1000 pte 0x21FC7017 pa 0x87F1C000 perm 0x17
va 0x2000 pte 0x21FC6C07 pa 0x87F1B000 perm 0x7
va 0x3000 pte 0x21FC68D7 pa 0x87F1A000 perm 0xD7
va 0x4000 pte 0x0 pa 0x0 perm 0x0
va 0x5000 pte 0x0 pa 0x0 perm 0x0
va 0x6000 pte 0x0 pa 0x0 perm 0x0
va 0x7000 pte 0x0 pa 0x0 perm 0x0
va 0x8000 pte 0x0 pa 0x0 perm 0x0
va 0x9000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFF6000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFF7000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFF8000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFF9000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFFA000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFFB000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFFC000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFFD000 pte 0x21FD4813 pa 0x87F52000 perm 0x13
va 0xFFFFE000 pte 0x21FD4CC7 pa 0x87F53000 perm 0xC7
va 0xFFFFF000 pte 0x2000184B pa 0x80006000 perm 0x4B
print_pgtbl: OK
ugetpid_test starting
ugetpid_test: OK
print_kpgtbl starting
page table 0x0000000087f22000
..0: pte 0x0000000021fc7801 pa 0x0000000087f1e000
.. ..0: pte 0x0000000021fc7401 pa 0x0000000087f1d000
.. .. ..0: pte 0x0000000021fc7c5b pa 0x0000000087f1f000
.. .. ..1: pte 0x0000000021fc70d7 pa 0x0000000087f1c000
.. .. ..2: pte 0x0000000021fc6c07 pa 0x0000000087f1b000
.. .. ..3: pte 0x0000000021fc68d7 pa 0x0000000087f1a000
..255: pte 0x0000000021fc8401 pa 0x0000000087f21000
.. ..511: pte 0x0000000021fc8001 pa 0x0000000087f20000
.. .. ..509: pte 0x0000000021fd4813 pa 0x0000000087f52000
.. .. ..510: pte 0x0000000021fd4cc7 pa 0x0000000087f53000
.. .. ..511: pte 0x000000002000184b pa 0x0000000080006000
pgaccess_test starting
pgaccess_test: OK