Lecture Note 5.3

Task 1: Implementing ps as a System Call in xv6-riscv

Step 0: Overview

Goal: Implement **ps** as a new system call in xv6-riscv so that when a user types **ps** in the shell, they see a list of processes (PID, state, name), similar to the **ps** command in Unix.

High-level design (simple version):

- Reuse the existing kernel function **procdump()** (which already walks the process table and prints PID, state, name).
- Expose that functionality through a new system call **ps()**.
- Add a small user program **ps**, that calls this system call.

Files we have to touch:

1. Kernel side

- kernel/proc.c (already has **procdump()**, used by our syscall)
- kernel/sysproc.c (implement **sys_ps**)
- kernel/syscall.h (add **SYS_ps** number)
- kernel/syscall.c (connect **SYS_ps** to **sys_ps**)

2. User side

- user/usys.pl → generates user/usys.S (add entry("**ps**"))
- user/user.h (add **int ps(void)**; prototype)
- user/ps.c (user program that calls **ps()**)

3. Build system

- Makefile (add **$U/_ps**\ to UPROGS)

Step 1: Where Does Process Information Live? (proc.c and proc.h)
Before adding the system call, students should know where process information is stored and how it is protected.

In xv6-riscv, every process is represented by a struct proc defined in kernel/proc.h, and all processes live in a global array **proc[NPROC]** defined in kernel/proc.c.

1.1 struct proc in kernel/proc.h
A simplified version of struct proc looks like this:

```
// kernel/proc.h

enum procstate { UNUSED, USED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

struct proc {
  struct spinlock lock;   // protects all fields below

  enum procstate state;   // current state
  void *chan;             // if non-zero, sleeping on this channel
  int killed;             // if non-zero, process has been killed
  int xstate;             // exit status
  int pid;                // process ID

  uint64 sz;              // size of process memory (bytes)
  pagetable_t pagetable;  // user page table
  struct trapframe *tf;   // saved user registers

  struct context context; // swtch() here to run process
  struct proc *parent;    // parent process

  struct file *ofile[NOFILE]; // open files
  struct inode *cwd;          // current directory
  char name[16];              // process name for debugging
};
```

Key fields for ps:
- **pid:** the unique process ID.
- **state:** the current state (RUNNING, RUNNABLE, SLEEPING, ZOMBIE, etc.).
- **name:** a short name of the process (usually the program name like "init", "sh", "ls").
- **lock:** a spinlock that must be held whenever the kernel reads or writes the other fields.

1.2 Process Table and procdump() in kernel/proc.c

The global process table and the debug function **procdump()** live in kernel/proc.c:

```
// kernel/proc.c

struct proc proc[NPROC];  // global process table
```

xv6 also includes a debug helper that prints all processes when you press Ctrl+P on the console:

```
void
procdump(void)
{
  struct proc *p;

  for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if(p->state != UNUSED) {
      char *s;
      switch(p->state) {
      case SLEEPING: s = "SLEEPING"; break;
      case RUNNABLE: s = "RUNNABLE"; break;
      case RUNNING:  s = "RUNNING";  break;
      case ZOMBIE:   s = "ZOMBIE";   break;
      default:       s = "USED";     break;
      }
      printf("%d %s %s\n", p->pid, s, p->name);
    }
    release(&p->lock);
  }
}
```

Important observations for students:

- **procdump()** iterates over every struct proc in **proc[]**.
- For each process p, it **acquires p->lock** before reading state, pid, and name.
- It skips UNUSED entries.
- It prints PID, state, and name, then **releases p->lock**.

Our **ps** system call will simply call **procdump()**, so we automatically reuse this safe and well-tested logic.

Step 2: Add a New Syscall Number in kernel/syscall.h

System calls in xv6 are identified by numbers defined in kernel/syscall.h. We must assign a unique number to **SYS_ps**.

```
// kernel/syscall.h
// System call numbers

#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_pause  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_ps     22      //new system call ps
```

Explanation:

- The new constant **SYS_ps** is the syscall number the kernel and user stubs will use.
- If your file has more syscalls, you can use the next free number instead of 22.

Step 3: Connect SYS_ps to sys_ps in kernel/syscall.c

kernel/syscall.c contains the system call dispatch table. We must:

1. Declare **sys_ps()** as an external function.
2. Add an entry mapping **SYS_ps** to **sys_ps** in the **syscalls[] array**.

```
// kernel/syscall.c
extern uint64 sys_fork(void);
extern uint64 sys_exit(void);
extern uint64 sys_wait(void);
extern uint64 sys_pipe(void);
extern uint64 sys_read(void);
extern uint64 sys_kill(void);
extern uint64 sys_exec(void);
extern uint64 sys_fstat(void);
extern uint64 sys_chdir(void);
extern uint64 sys_dup(void);
extern uint64 sys_getpid(void);
extern uint64 sys_sbrk(void);
extern uint64 sys_sleep(void);
extern uint64 sys_uptime(void);
extern uint64 sys_open(void);
extern uint64 sys_write(void);
extern uint64 sys_mknod(void);
extern uint64 sys_unlink(void);
extern uint64 sys_link(void);
extern uint64 sys_mkdir(void);
extern uint64 sys_close(void);
extern uint64 sys_ps(void);   // new declaration

static uint64 (*syscalls[])(void) = {
  [SYS_fork]   sys_fork,
  [SYS_exit]   sys_exit,
  [SYS_wait]   sys_wait,
  [SYS_pipe]   sys_pipe,
  [SYS_read]   sys_read,
  [SYS_kill]   sys_kill,
  [SYS_exec]   sys_exec,
  [SYS_fstat]  sys_fstat,
  [SYS_chdir]  sys_chdir,
  [SYS_dup]    sys_dup,
  [SYS_getpid] sys_getpid,
  [SYS_sbrk]   sys_sbrk,
  [SYS_sleep]  sys_sleep,
  [SYS_uptime] sys_uptime,
  [SYS_open]   sys_open,
  [SYS_write]  sys_write,
  [SYS_mknod]  sys_mknod,
  [SYS_unlink] sys_unlink,
  [SYS_link]   sys_link,
  [SYS_mkdir]  sys_mkdir,
  [SYS_close]  sys_close,
  [SYS_ps]     sys_ps,       // new table entry
};
```

Now, when a user program issues a syscall with number **SYS_ps**, the kernel will call **sys_ps()**.

Step 4: Implement sys_ps in kernel/sysproc.c
kernel/sysproc.c holds the implementations of many system calls (sys_getpid, sys_pipe, etc.). We add a new function **sys_ps()**, that simply calls procdump().

```
// kernel/sysproc.c
#include "types.h"
#include "riscv.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "spinlock.h"
#include "proc.h"

uint64
sys_ps(void)
{
  procdump();  // print the process table to the console
  return 0;
}
```

Explanation:
- **sys_ps()** runs in kernel mode.
- It reuses **procdump()** from proc.c, which already **acquires p->lock** and walks through **proc[]** safely.
- Returning 0 is enough because the main "result" of this syscall is the printed output.

Step 5: Generate the User-Space Stub via user/usys.pl

xv6 uses user/usys.pl to generate user/usys.S, which contains assembly stubs for each system call. Your usys.pl looks like this:

```
#!/usr/bin/perl -w

# Generate usys.S, the stubs for syscalls.

print "# generated by usys.pl - do not edit\n";

print "#include \"kernel/syscall.h\"\n";

sub entry {
    my $prefix = "sys_";
    my $name = shift;
    if ($name eq "sbrk") {
       print ".global $prefix$name\n";
       print "$prefix$name:\n";
    } else {
       print ".global $name\n";
       print "$name:\n";
    }
    print " li a7, SYS_${name}\n";
    print " ecall\n";
    print " ret\n";
}

entry("fork");
entry("exit");
entry("wait");
entry("pipe");
entry("read");
entry("write");
entry("close");
entry("kill");
entry("exec");
entry("open");
entry("mknod");
entry("unlink");
entry("fstat");
entry("link");
entry("mkdir");
entry("chdir");
entry("dup");
entry("getpid");
entry("sbrk");
entry("pause");
entry("uptime");
```

To add ps, simply append:

entry("ps");

So the bottom of user/usys.pl becomes:

```
entry("getpid");
entry("sbrk");
entry("pause");
entry("uptime");
entry("ps");   // new entry for ps
```

When you run make, usys.pl regenerates user/usys.S, and you will now find a stub like:

```
.global ps
ps:
 li a7, SYS_ps
 ecall
 ret
```

This stub allows C code in user space to call **ps()** as a normal function, which will execute the ecall instruction with a7 = SYS_ps.

Step 6: Add the User-Space Prototype in user/user.h

The header user/user.h contains prototypes for user-visible system calls. To avoid compiler warnings and to make **ps()** callable from user programs, add:

```
// user/user.h

int ps(void);   // prototype for the new ps system call
```

Step 7: Write the User Program user/ps.c

Now we create a simple user program named **ps.c** that just calls the **ps()** system call and then exits.

```
// user/ps.c
#include "kernel/types.h"
#include "user/user.h"

int
main(void)
{
  ps();     // call the ps system call
  exit(0);
}
```

Explanation:

- **ps()** is the user stub generated from usys.S.
- It triggers the ecall instruction with **a7 = SYS_ps**.
- The kernel handles that by calling **sys_ps()**, which calls **procdump()**.

Step 8: Add ps to the Build in the Makefile

The top-level Makefile controls which user programs get built into the xv6 file system. Find the UPROGS definition and add **$U/_ps** to it.

```
UPROGS=\
	$U/_cat\
	$U/_echo\
	$U/_forktest\
	$U/_grep\
	$U/_init\
	$U/_kill\
	$U/_ln\
	$U/_ls\
	$U/_mkdir\
	$U/_rm\
	$U/_sh\
	$U/_stressfs\
	$U/_usertests\
	$U/_grind\
	$U/_wc\
	$U/_zombie\
	$U/_logstress\
	$U/_forphan\
	$U/_dorphan\
	$U/_ps\
```

Adding **$U/_ps** ensures that user/ps.c is compiled, linked, and packed into the xv6 disk image as a program named ps.

Step 9: Rebuild xv6 and Test ps

After all changes are made:

1. Clean and rebuild xv6:

```
make clean
make qemu
```

2. Once xv6 boots and you get the shell prompt, run:

```
$ ps
```

You should see the output similar to:

```
1 sleep  init
2 sleep  sh
5 run    ps
```

Extra explanation: How This Uses proc.c, proc.h, struct proc, and Locks
Even though we only touched syscall-related files and a small user program, the core logic lives in proc.c and proc.h.

1. Process representation:
- Each process is represented by a struct proc defined in kernel/proc.h.
- The important fields for ps are pid, state, name, and lock.

2. Process table:
- All processes live in the global array struct proc proc[NPROC]; in kernel/proc.c.
- The scheduler, fork, exit, sleep, wakeup, and other functions all iterate over this array.

3. Locks and concurrency:
- Each struct proc has a spinlock (p->lock).
- Any kernel code that reads or writes process fields must first acquire(&p->lock) and release(&p->lock) afterwards.
- This prevents race conditions when multiple CPUs modify the same process at the same time.

4. procdump() as the core of ps:
- procdump() in proc.c walks through proc[], locks each process, checks p->state, and prints pid, state, and name.
- By calling procdump() inside sys_ps(), our ps syscall reuses the same safe, lock-aware traversal of the process table.

5. End-to-end chain for ps:
- User types ps → shell runs user program ps (user/ps.c).
- ps() in user space (generated from usys.S) sets a7 = SYS_ps and executes ecall.
- Kernel sees SYS_ps and dispatches to sys_ps() via syscall.c.
- sys_ps() calls procdump().
- procdump() iterates over struct proc proc[NPROC], uses p->lock to protect access, and prints the info.

Task 2: Implementing sleep(int) System Call in xv6-riscv

Step 0: Overview

Goal:

By the end of this lecture, you should be able to:

- Explain how system calls work in xv6-riscv (user → kernel → user).
- Implement a new system call, sleep(int n), that blocks a process for n timer ticks.
- Wire the syscall correctly through:
  - kernel/syscall.h
  - kernel/syscall.c
  - kernel/sysproc.c
  - user/user.h
  - user/usys.pl
  - user/Makefile and user/sleep.c

Big picture: sleep system call pipeline

We want to support this call from the user space:

```
int sleep(int n);   // n = number of ticks
```

Usage in the xv6 shell:

```
$ sleep 100   // sleep for ~1 second if ticks = 100 per second
```

Conceptual flow:

```
user program (sleep.c)      calls      sleep(n)
        |
        ▼
user stub in usys.S         sets       a7 = SYS_sleep, a0 = n, then ecall
        |
        ▼
kernel trap handler         calls      syscall()
        |
        ▼
syscall() dispatch table    calls      sys_sleep()
        |
        ▼
sys_sleep()                 blocks process using ticks and sleep()/wakeup()
```

Files we will touch (xv6-riscv)

You will mainly modify or create:

1. Kernel side
   – kernel/syscall.h (add **SYS_sleep** number)
   – kernel/syscall.c (connect **SYS_sleep** to **sys_sleep**)
   – kernel/sysproc.c (implement **sys_sleep)**
2. User side
   – user/user.h (add **int sleep(int)**; prototype)
   – user/usys.pl → generates user/usys.S (add entry("**sleep**"))
   – user/sleep.c (user program that calls **sleep(int))**
3. Build System
   – user/Makefile (add **$U/_sleep**\ to UPROGS)

Step 1: Add SYS_sleep in kernel/syscall.h

Open: kernel/syscall.h

You should see entries like:

```
// kernel/syscall.h
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_pause  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_ps     22
#define Sys_sleep  23   //new system call sleep
```

Explanation:

- The new constant **SYS_sleep** is the syscall number the kernel and user stubs will use.
- If your file has more syscalls, you can use the next free number instead of 23.

Step 2: Implement sys_sleep in kernel/sysproc.c

2.1 Required includes and externs

At the top of kernel/sysproc.c you should already have:

```
#include "types.h"
#include "riscv.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "spinlock.h"
#include "proc.h"
```

We will also need:

```
extern uint ticks;
extern struct spinlock tickslock;
```

2.2 The final working version of sys_sleep

Important: In this environment, argint was effectively used as if it returns void, so we do not check argint's return value. We just call it to fill n and then validate n ourselves.

```
extern uint ticks;
extern struct spinlock tickslock;

uint64
sys_sleep(void)
{
  int n = 0;
  uint ticks0;

  // Read the first syscall argument (ticks) into n.
  // Here we do not compare the return of argint; we just use it to store n.
  argint(0, &n);

  // Defensive check: non-positive sleep duration does nothing.
  if(n <= 0)
    return 0;

  acquire(&tickslock);
  ticks0 = ticks;

  while(ticks - ticks0 < (uint)n){
    // If this process has been killed while sleeping, abort.
    if(myproc()->killed){
      release(&tickslock);
      return -1;
    }
    // Sleep on the address of ticks while holding tickslock.
    sleep(&ticks, &tickslock);
    // When sleep() returns, tickslock has been re-acquired.
  }

  release(&tickslock);
  return 0;
}
```

2.3 What is happening inside sys_sleep?

- argint(0, &n); reads the 0-th syscall argument into n.
- ticks is a global counter of timer ticks.
- ticks0 = ticks; remembers the starting tick count.
- The loop runs until ticks - ticks0 >= n.
- sleep(&ticks, &tickslock); puts the process to sleep on &ticks.
- myproc()->killed is checked to abort if the process is killed.

Step 3: Connect SYS_sleep to sys_sleep in kernel/syscall.c

kernel/syscall.c contains the system call dispatch table. We must:

1. Declare **sys_ps()** as an external function.
2. Add an entry mapping **SYS_ps** to **sys_ps** in the **syscalls[] array**.

```
// kernel/syscall.c
extern uint64 sys_fork(void);
extern uint64 sys_exit(void);
extern uint64 sys_wait(void);
extern uint64 sys_pipe(void);
extern uint64 sys_read(void);
extern uint64 sys_kill(void);
extern uint64 sys_exec(void);
extern uint64 sys_fstat(void);
extern uint64 sys_chdir(void);
extern uint64 sys_dup(void);
extern uint64 sys_getpid(void);
extern uint64 sys_sbrk(void);
extern uint64 sys_sleep(void);
extern uint64 sys_uptime(void);
extern uint64 sys_open(void);
extern uint64 sys_write(void);
extern uint64 sys_mknod(void);
extern uint64 sys_unlink(void);
extern uint64 sys_link(void);
extern uint64 sys_mkdir(void);
extern uint64 sys_close(void);
extern uint64 sys_ps(void);
extern uint64 sys_sleep(void);   // new declaration

static uint64 (*syscalls[])(void) = {
  [SYS_fork]   sys_fork,
  [SYS_exit]   sys_exit,
  [SYS_wait]   sys_wait,
  [SYS_pipe]   sys_pipe,
  [SYS_read]   sys_read,
  [SYS_kill]   sys_kill,
  [SYS_exec]   sys_exec,
  [SYS_fstat]  sys_fstat,
  [SYS_chdir]  sys_chdir,
  [SYS_dup]    sys_dup,
  [SYS_getpid] sys_getpid,
  [SYS_sbrk]   sys_sbrk,
  [SYS_sleep]  sys_sleep,
  [SYS_uptime] sys_uptime,
  [SYS_open]   sys_open,
  [SYS_write]  sys_write,
  [SYS_mknod]  sys_mknod,
  [SYS_unlink] sys_unlink,
```

```
  [SYS_link]   sys_link,
  [SYS_mkdir]  sys_mkdir,
  [SYS_close]  sys_close,
  [SYS_ps]     sys_ps,
  [SYS_sleep]  sys_sleep,  // new table entry
};
```

Now, when a user program issues a syscall with number **SYS_sleep**, the kernel will call **sys_sleep(int)**.

Step 4: Add the User-Space Prototype in user/user.h

The header user/user.h contains prototypes for user-visible system calls. To avoid compiler warnings and to make **sleep(int)** callable from user programs, add:

```
// user/user.h

int sleep(int);   // prototype for the new ps system call
```

Step 5: Generate the RISC-V syscall stub via user/usys.pl

Open: user/usys.pl. The script looks like:

```
#!/usr/bin/perl -w

# Generate usys.S, the stubs for syscalls.

print "# generated by usys.pl - do not edit\n";

print "#include \"kernel/syscall.h\"\n";

sub entry {
    my $prefix = "sys_";
    my $name = shift;
    if ($name eq "sbrk") {
       print ".global $prefix$name\n";
       print "$prefix$name:\n";
    } else {
       print ".global $name\n";
       print "$name:\n";
    }
    print " li a7, SYS_${name}\n";
    print " ecall\n";
    print " ret\n";
}

entry("fork");
entry("exit");
entry("wait");
entry("pipe");
entry("read");
entry("write");
entry("close");
entry("kill");
```

```
entry("exec");
entry("open");
entry("mknod");
entry("unlink");
entry("fstat");
entry("link");
entry("mkdir");
entry("chdir");
entry("dup");
entry("getpid");
entry("sbrk");
entry("pause");
entry("uptime");
entry("ps");
```

To add ps, simply append:
entry("sleep");
So the bottom of user/usys.pl becomes:

```
ntry("getpid");
entry("sbrk");
entry("pause");
entry("uptime");
entry("ps");
entry("sleep")  // new entry for sleep
```

When you run make, usys.pl regenerates user/usys.S, and you will now find a stub like:

```
.global sleep
sleep:
  li a7, SYS_sleep   # syscall number in a7
  ecall              # trap into kernel
  ret
```

Step 6: Write the user program user/sleep.c

Create user/sleep.c with:

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int
main(int argc, char *argv[])
{
  int n;

  if(argc != 2){
    fprintf(2, "Usage: sleep ticks\n");
    exit(1);
  }

  n = atoi(argv[1]);

  if(n < 0){
    fprintf(2, "sleep: negative value not allowed\n");
    exit(1);
  }

  if(sleep(n) < 0){
    fprintf(2, "sleep: failed\n");
    exit(1);
  }

  exit(0);
}
```

Step 7: Add sleep to the Build in the Makefile

The top-level Makefile controls which user programs get built into the xv6 file system. Find the UPROGS definition and add **$U/_sleep** to it.

```
UPROGS=\
       $U/_cat\
       $U/_echo\
       $U/_forktest\
       $U/_grep\
       $U/_init\
       $U/_kill\
       $U/_ln\
       $U/_ls\
       $U/_mkdir\
       $U/_rm\
       $U/_sh\
       $U/_stressfs\
       $U/_usertests\
       $U/_grind\
       $U/_wc\
       $U/_zombie\
```

```
      $U/_logstress\
      $U/_forphan\
      $U/_dorphan\
      $U/_ps\
       $U/_sleep\
```

Adding **$U/_sleep** ensures that user/ps.c is compiled, linked, and packed into the xv6 disk image as a program named ps.

Step 8: Build and test in QEMU

From the root of your xv6-riscv project:

```
$ make clean
$ make qemu
```

Once xv6 boots and shows the shell prompt:

```
$ sleep 1
$ sleep 100
$ sleep 100; echo hello
hello
```

Here, sleep 1 will pause very briefly (1 tick). sleep 100 will pause for approximately 1 second if there are 100 ticks per second. sleep 100; echo done shows that the shell resumes after the sleep finishes.

Summary:

1. System call path (RISC-V):

```
sleep.c (user)        →   sleep() stub (usys.S)
    arg n             →   a0 = n, a7 = SYS_sleep, ecall
    |
    ▼
trap handler          →   syscall()
    |
    ▼
syscall()             →   syscalls[SYS_sleep] = sys_sleep
    |
    ▼
sys_sleep() (kernel)  →   uses ticks, tickslock, sleep(), wakeup()
```

2. Key RISC-V details:

- ecall instruction invokes the syscall.
- a7 holds the syscall number (SYS_sleep).
- a0..a5 hold arguments.
- Result is returned in a0.