1. What tools does the OS include? Or is it only the kernel?

R: It is not only the kernel, the XV6 OS include some system calls that are used by some procedure that needs them. The following System calls that are located in the file “syscall.c” are the aforementioned ones:

extern int sys\_chdir(void);

extern int sys\_close(void);

extern int sys\_dup(void);

extern int sys\_exec(void);

extern int sys\_exit(void);

extern int sys\_fork(void);

extern int sys\_fstat(void);

extern int sys\_getpid(void);

extern int sys\_kill(void);

extern int sys\_link(void);

extern int sys\_mkdir(void);

extern int sys\_mknod(void);

extern int sys\_open(void);

extern int sys\_pipe(void);

extern int sys\_read(void);

extern int sys\_sbrk(void);

extern int sys\_sleep(void);

extern int sys\_unlink(void);

extern int sys\_wait(void);

extern int sys\_write(void);

extern int sys\_uptime(void);

static int (\*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,

};

1. What critical section protection mechanisms implement?

R: The XV6 OS implement “Mutual exclusion spin locks”; this code is located within the file “spinlock.c”. This is actually the code:

#include "types.h"

#include "defs.h"

#include "param.h"

#include "x86.h"

#include "memlayout.h"

#include "mmu.h"

#include "proc.h"

#include "spinlock.h"

Void initlock(struct spinlock \*lk, char \*name)

{

lk->name = name;

lk->locked = 0;

lk->cpu = 0;

}

// Acquire the lock.

// Loops (spins) until the lock is acquired.

// Holding a lock for a long time may cause

// other CPUs to waste time spinning to acquire it.

void

acquire(struct spinlock \*lk)

{

pushcli(); // disable interrupts to avoid deadlock.

if(holding(lk))

panic("acquire");

// The xchg is atomic.

// It also serializes, so that reads after acquire are not

// reordered before it.

while(xchg(&lk->locked, 1) != 0)

;

// Record info about lock acquisition for debugging.

lk->cpu = cpu;

getcallerpcs(&lk, lk->pcs);

}

// Release the lock.

void

release(struct spinlock \*lk)

{

if(!holding(lk))

panic("release");

lk->pcs[0] = 0;

lk->cpu = 0;

// The xchg serializes, so that reads before release are

// not reordered after it. The 1996 PentiumPro manual (Volume 3,

// 7.2) says reads can be carried out speculatively and in

// any order, which implies we need to serialize here.

// But the 2007 Intel 64 Architecture Memory Ordering White

// Paper says that Intel 64 and IA-32 will not move a load

// after a store. So lock->locked = 0 would work here.

// The xchg being asm volatile ensures gcc emits it after

// the above assignments (and after the critical section).

xchg(&lk->locked, 0);

popcli();

}

// Record the current call stack in pcs[] by following the %ebp chain.

void

getcallerpcs(void \*v, uint pcs[])

{

uint \*ebp;

int i;

ebp = (uint\*)v - 2;

for(i = 0; i < 10; i++){

if(ebp == 0 || ebp < (uint\*)KERNBASE || ebp == (uint\*)0xffffffff)

break;

pcs[i] = ebp[1]; // saved %eip

ebp = (uint\*)ebp[0]; // saved %ebp

}

for(; i < 10; i++)

pcs[i] = 0;

}

// Check whether this cpu is holding the lock.

int

holding(struct spinlock \*lock)

{

return lock->locked && lock->cpu == cpu;

}

// Pushcli/popcli are like cli/sti except that they are matched:

// it takes two popcli to undo two pushcli. Also, if interrupts

// are off, then pushcli, popcli leaves them off.

void

pushcli(void)

{

int eflags;

eflags = readeflags();

cli();

if(cpu->ncli++ == 0)

cpu->intena = eflags & FL\_IF;

}

void

popcli(void)

{

if(readeflags()&FL\_IF)

panic("popcli - interruptible");

if(--cpu->ncli < 0)

panic("popcli");

if(cpu->ncli == 0 && cpu->intena)

sti();

}

1. With which techniques (TSL, Peterson's, etc) does the operating system do this protection?

R: It uses an atomic operation helped by a fuction called “xchg” as it can be seen in the code below:

while(xchg(&lk->locked, 1) != 0)

;

1. What information is stored in the process table?

R: The information that is stored in the process table is the spinlock of the processes and an array of “n” processes. (something that can be seen in the “proc.c” file)

struct {

struct spinlock lock;

struct proc proc[NPROC];

} ptable;

Each process has its information as you can see in this code here below: (something that can be seen in the “proc.h” file)

// Per-process state

struct proc {

uint sz; // Size of process memory (bytes)

pde\_t\* pgdir; // Page table

char \*kstack; // Bottom of kernel stack for this process

enum procstate state; // Process state

volatile int pid; // Process ID

struct proc \*parent; // Parent process

struct trapframe \*tf; // Trap frame for current syscall

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

void \*chan; // If non-zero, sleeping on chan

int killed; // If non-zero, have been killed

struct file \*ofile[NOFILE]; // Open files

struct inode \*cwd; // Current directory

char name[16]; // Process name (debugging)

};

1. Which are the process states in this OS?

R: The process states are define within the file “proc.c”. Those are these ones:

static char \*states[] = {

[UNUSED] "unused",

[EMBRYO] "embryo",

[SLEEPING] "sleep ",

[RUNNABLE] "runble",

[RUNNING] "run ",

[ZOMBIE] "zombie"

};

1. Up to how many processes can be run simultaneously by the operating system?

R: Due to the fact that there can be running as many processes as there are processors, since there are 10 cpu define, at most 10 processes can be run at the same time.

uint pc[10];

located on “proc.c”.

1. Modify fork so that it searches for an available space in the process table starting from the end of it and finishing at the first element.

R: Within the “proc.c” there is a method called “allocproc”, its fuction is to looking for an avaible space in the process table, but its actually start from the start of it, so in order to invert this, all I had to do was change a single line in the next for, so it start to looking from th end to the start.

Before:

for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)

if(p->state == UNUSED)

goto found;

release(&ptable.lock);

return 0;

After:

for(p = &ptable.proc[NPROC-1] ; p <0; p--)

if(p->state == UNUSED)

goto found;

release(&ptable.lock);

return 0;