

xv6©-rev10  
(Copyright Frans Kaashoek, Robert Morris, and Russ Cox.)  
Scheduler

Carmi Merimovich

Tel-Aviv Academic College

November 20, 2017

## Context

```
1219 kinit1(end, P2V(4*1024*1024)); // phys page allocation
    kvmalloc(); // kernel page table
    :
1222 seginit(); // set up segments
    :
1224 pinit(); // process table
    :
    :
1226 kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must copy
1227 userinit(); // first user process
    mpmain();
```

## Auxiliary context

- The primary processor begins its C code in **main()**.
- The auxiliary processors begins their C code in **mpenter()**.
- The state on entering either **main()** or **mpenter()** is the same.
- There is a separate stack of each processor.

```
1241 static void mpenter(void) {  
    switchkvm();  
    seginit();  
    lapicinit();  
    mpmain();  
}
```

## mycpu()

```
2436 struct cpu* mycpu(void) {  
    int apicid, i;  
  
    if (readeflags() & FL_IF)  
        panic("mycpu_called_with_interrupts_enabled\n");  
  
    apicid = lapicid();  
    for (i = 0; i < ncpu; ++i) {  
        if (cpus[i].apicid == apicid)  
            return &cpus[i];  
    }  
    panic("unknown_apicid\n");  
}
```

## mpmain()

```
1252 static void mpmain(void) {  
    cprintf("cpu%d: _starting _%d\n", cpuid(), cpuid());  
    idtinit(); // load idt register  
    xchg(&(mycpu()->started), 1); // tell startothers()  
    scheduler(); // start running processes  
}
```

## myproc()

```
2456 struct proc *myproc(void) {  
    struct cpu *c;  
    struct proc *p;  
    pushcli();  
    c = mycpu();  
    p = c->proc;  
    popcli();  
    return p;  
}
```

## scheduler

2758

```
void scheduler(void) {  
    struct proc *p;  
    struct cpu *c = mycpu();  
    c->proc = 0;  
    for(;;) { sti();  
        acquire(&ptable.lock);  
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {  
            if (p->state != RUNNABLE) continue;  
  
            c->proc = p;  
            switchvm(p);  
            p->state = RUNNING;  
            swtch(&c->scheduler, p->context);  
            switchkvm();  
  
            c->proc = 0;  
        }  
        release(&ptable.lock);  
    }
```

## scheduler() operation

- For each proc struct `p` with state `RUNNABLE` the following is executed:
  - `c->proc = p;`
  - `switchvm()`.
  - `swtch()`.
  - `switchkvm()`.
  - `c->proc=NULL.`



## swtch()

3058

```
.globl swtch
swtch:
    movl 4(%esp), %eax
    movl 8(%esp), %edx

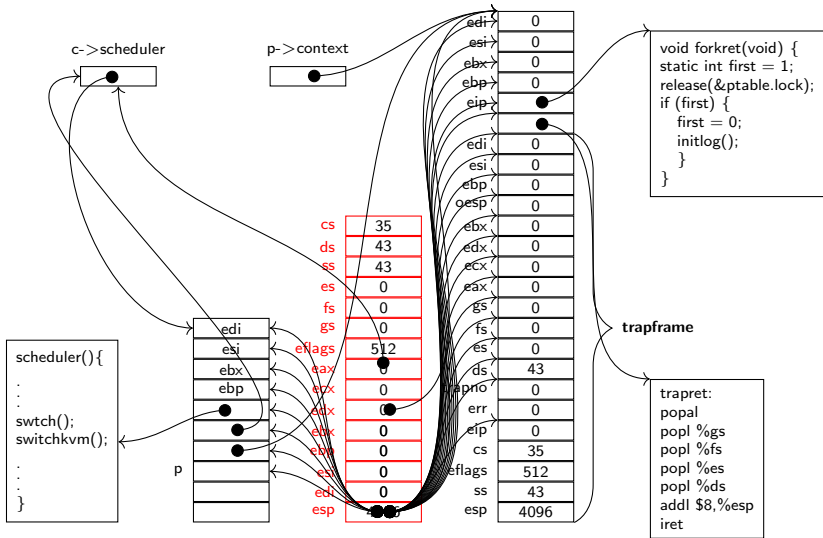
    pushl %ebp
    pushl %ebx
    pushl %esi
    pushl %edi

    movl %esp, (%eax)
    movl %edx, %esp
```

```
    popl %edi
    popl %esi
    popl %ebx
    popl %ebp
    ret
```

The eip field of context is generated by the instruction calling swtch.

## switch() operation



## Context switch

- Calling **switch()**:
  - Creates a **context** structure on the current stack.
  - Stores the **context** structure address created in the first argument.
  - Load the **context** structure pointed to by the second argument.
- We are switching KERNEL contexts.
- User mode context of a process is loaded by the kernel side of the process.

## Kernel context seems too small

- Where are **eax**, **ecx**, **edx**????
  - The **gcc** calling conventions deals with them!
- Where are **cs**, **ds**, and **ss**????
  - The base and limit fields are identical across all kernel sides.
  - So, they need to be loaded only on each kernel entering.
- Where is **gdt**????
  - The address and size fields are different across processors.
  - The base and limit **MUST NOT** change between kernel sides on the same CPU.
  - Since **gdt** is privileged, it needs to be loaded **ONLY** on kernel initialization.

## If switching to user mode is expected:

- The `tr` register should contain the index of a TSS descriptor.
- The TSS descriptor should point to a `taskstate` structure.
- The `ss0` and `esp0` fields should point to a valid kernel stack top.
- The above is ESSENTIAL for proper interrupt service in user mode.

## switchvm

```
1860 void switchvm(struct proc *p) {
    pushcli();
    mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A,
                                &mycpu()->ts,
                                sizeof(mycpu()->ts)-1, 0);
    mycpu()->gdt[SEG_TSS].s = 0;
    mycpu()->ts.ss0 = SEG_KDATA<<3;
    mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
    mycpu()->ts.iomb = (ushort) 0xFFFF;
    ltr(SEG_TSS << 3);
    if (p->pgdir == 0)
        panic("switchvm: _no_pgdir");
    lcr3(v2p(p->pgdir)); // switch to new address space
    popcli();
}
```

## taskstate (hardware structure)

link	
esp0	
	ss0
esp1	
	ss1
esp2	
	ss2
cr3	
eip	
eflags	
eax	
ecx	
edx	
ebx	
esp	
ebp	
esi	
edi	

	es
	cs
	ss
	ds
	fs
	gs
	ldt
	t
iomb	

## taskstate in C

851

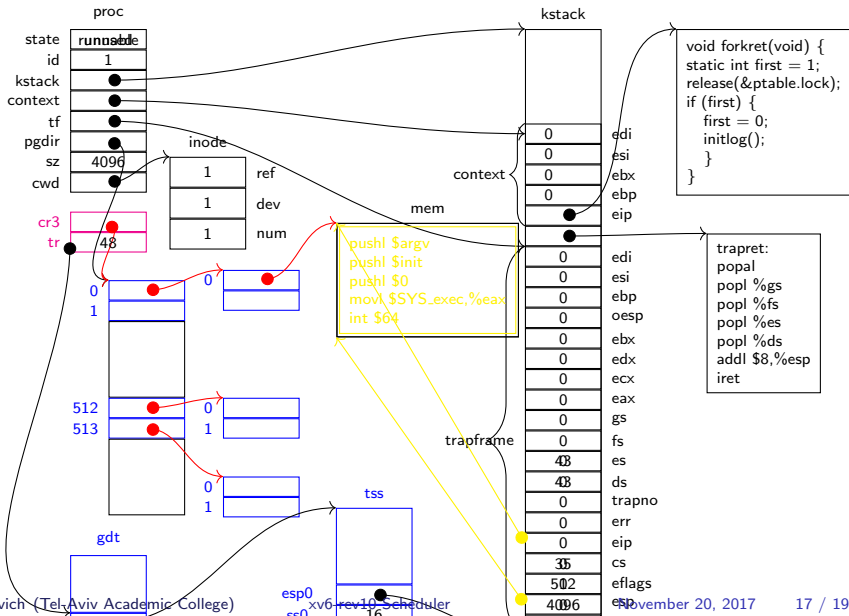
```
struct taskstate
{
    uint link;
    uint esp0;
    ushort ss0;
    ushort padding1;
    uint *esp1;
    ushort ss1;
    ushort padding2;
    uint *esp2;
    ushort ss2;
    ushort padding3;
    void *cr3;
    uint *eip;
    uint eflags;
    uint eax;
    uint ecx;
```

```
    uint edx;
    uint ebx;
    uint *esp;
    uint *ebp;
    uint esi;
    uint edi;
    ushort es;
    ushort padding4;
    ushort cs;
    ushort padding5;
    ushort ss;
    ushort padding6;
    ushort ds;
    ushort padding7;
    ushort fs;
    ushort padding8;
```

```
    ushort gs;
    ushort padding9;
    ushort ldt;
    ushort padding10;
    ushort t;
    ushort iomb;
};
```



# switchvm



## co-routines

- The scheduler switches to a process by using:

```
2478      swtch(&c->scheduler , p->context );
```

- A process leaves the cpu by returning to the scheduler using:

```
2516      swtch(&p->context , mycpu()->scheduler );
```

- We have here co-routines.

## Event driven kernel

- At this point there is no more LINEAR EXECUTION of the kernel.
- The kernel as of now is EVENT DRIVEN.