ICS143A: Principles of Operating Systems

Lecture 13: Context switch

Anton Burtsev
November, 2017
- User mode
- Two stacks
  - Kernel and user
  - Kernel stack is empty
- Page table
- GDT
Interrupt path

User stack of a process (can grow up to 2GBs)

Code, data, heap

Timer: IRQ0 -> vector 32

Interrupt Vector #

Kernel Stack of a process (4K)

User state (saved by hardware)

EBP → Process

Last stack frame

Argument 1
Argument 2
Calling EIP ++
Old EBP
Local variables
Saved local values, e.g. push EAX, etc

SS
ESP
EFLAGS
CS
EIP

Kernel Stack of a process (4K)

GDT

IDT

Page table
Level 1

Level 2

0 - 4KB
4 - 4KB

2GB - 2GB + 4MB

0 - 4K
4K - 8K

(4MB-4K) - 4MB

TSS

timer

Kernel code

vector 32

CR3: pt

TSS: tss

CS: gdt

IDT: idt

EIP: <kernel>

SS: #2

CS: #1

SS: #2

GDT: gdt

ESP: <kernel>
Where does IDT (entry 32) point to?

vector32:
    pushl $0       // error code
    pushl $32      // vector #
    jmp alltraps

• Automatically generated
• From vectors.pl
  • vector.S
Kernel stack after interrupt

User state (saved by hardware)

- SS
- ESP
- EFLAGS
- CS
- EIP
- 0
- 32

Kernel Stack of a process (4K)

Call stack: vector32()
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # Set up data and per-cpu segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es
  movw $(SEG_KCPU<<3), %ax
  movw %ax, %fs
  movw %ax, %gs

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
Kernel stack after interrupt

User state (saved by hardware)

Vector32

Alltraps

ESP

Kernel Stack of a process (4K)

Trap frame

Call stack: vector32() alltraps()
alltraps:

# Build trap frame.
pushl %ds
pushl %es
pushl %fs
pushl %gs
pushal

# Set up data and per-cpu segments.
movw $(SEG_KDATA<<3), %ax
movw %ax, %ds
movw %ax, %es
movw $(SEG_KCPU<<3), %ax
movw %ax, %fs
movw %ax, %gs

# Call trap(tf), where tf=%esp
pushl %esp
call trap
3351 trap(struct trapframe *tf) {
...
3363   switch(tf->trapno){
3364     case T_IRQ0 + IRQ_TIMER:
3365       if(cpu->id == 0){
3366         acquire(&tickslock);
3367         ticks++;
3368         wakeup(&ticks);
3369         release(&tickslock);
3370       }
3372     break;
...
3423     if(proc && proc->state == RUNNING
3424       && tf->trapno == T_IRQ0+IRQ_TIMER)
3424       yield();
3351 trap(struct trapframe *tf)
3352 {
...
3363   switch(tf->trapno){
3364   case T_IRQ0 + IRQ_TIMER:
3365     if(cpu->id == 0){
3366       acquire(&tickslock);
3367       ticks++;
3368       wakeup(&ticks);
3369       release(&tickslock);
3370     }
3372   break;
...
3423   if(proc && proc->state == RUNNING
3424       && tf->trapno == T_IRQ0+IRQ_TIMER)
3424     yield();
Invoke the scheduler

```c
2777    yield(void)
2778    {
2779         acquire(&ptable.lock);
2780         proc->state = RUNNABLE;
2781         sched();
2782         release(&ptable.lock);
2783    }
```
2758 sched(void) {
...
2771   swtch(&proc->context, cpu->scheduler);
...
2773 }
Stack inside `swtch()` and its two arguments (passed on the stack)

Call stack:
- `vector32()`
- `alltraps()`
- `trap()`
- `yield()`
- `sched()`
- `switch(&proc->context, cpu->scheduler)`

User state (saved by hardware)
- SS
- ESP
- EFLAGS
- CS
- EIP
- 0
- 32
- DS
- ES
- FS
- GS
- All registers
- ESP
- EIP (alltraps)
- ...
- EIP (trap)
- ...
- EIP (yield)
- ...
- &proc->context
- cpu->scheduler
- EIP (sched)

Kernel Stack of a process (4K)
Trap frame

Context
- EIP (line: 2479)
- EBP
- EBX
- ESI
- EDI
swtch:
movl 4(%esp), %eax  # **old
movl 8(%esp), %edx  # *new

# Save old callee-save registers
pushl %ebp
pushl %ebx
pushl %esi
pushl %edi

# Switch stacksh
movl %esp, (%eax) # *old = %esp
movl %edx, %esp   # %esp = new

# Load new callee-save registers
popl %edi
popl %esi
popl %ebx
popl %ebp
ret

swtch()

void swtch(struct context **old,
           struct context *new);

• First argument:
  • A pointer to a pointer to a context
  • That we're going to save

• Second argument:
  • A pointer to a context
  • We're going to restore
2093 struct context {
2094    uint edi;
2095    uint esi;
2096    uint ebx;
2097    uint ebp;
2098    uint eip;
2099 };
swtch:

2958 movl 4(%esp), %eax
2959 movl 8(%esp), %edx

2961 # Save old callee-save registers
2962 pushl %ebp
2963 pushl %ebx
2964 pushl %esi
2965 pushl %edi

2968 # Switch stacksh
2969 movl %esp, (%eax)
2970 movl %edx, %esp

2972 # Load new callee-save registers
2973 popl %edi
2974 popl %esi
2975 popl %ebx
2976 popl %ebp
2977 ret

2093 struct context {
2094   uint edi;
2095   uint esi;
2096   uint ebx;
2097   uint ebp;
2098   uint eip;
2099 };

Stack inside swtch() and its two arguments (passed on the stack)

User state (saved by hardware)
- SS
- ESP
- EFLAGS
- CS
- EIP
- 0
- 32
- DS
- ES
- FS
- GS
- All registers

vector32
- ESP
- EIP (vector32)
- ...
- EIP (alltraps)

alltraps
- ...
- EIP (trap)

Call stack:
- vector32()
- alltraps()
- trap()
- yield()
- sched()
- switch(&proc->context, cpu->scheduler)

Kernel Stack of a process (4K)
Trap frame

Context
- EIP (line: 2479)
- EBP
- EBX
- ESI
- EDI
Context is always top of some stack

Kernel Stack of a process (4K)

User state (saved by hardware)
- SS
- ESP
- EFLAGS
- CS
- EIP
- 0
- 32
- DS
- ES
- FS
- GS
- All registers
- ESP
- EIP (alltraps)
- ...
- EIP (trap)
- ...
- EIP (yield)
- ...
- &proc->context
- cpu->scheduler
- EIP (sched)

Trap frame

Kernel Stack of a boot process (4K)

- EIP (main)
- ...
- EIP (mpmain)
- ...
- &proc->context
- cpu->scheduler
- EIP (scheduler)
- EBP
- EBX
- ESI
- EDI

Context
Context is always top of some stack, why?

- How does initialization of each CPU end?
1251 static void
1252 mpenter(void)
1253 {
1254   switchkvm();
1255   seginit();
1256   lapicinit();
1257   mpmain();
1258 }
// Common CPU setup code.
static void mpmain(void)
{
cprintf("cpu%d: starting\n", cpu->id);
idtinit(); // load idt register
xchg(&cpu->started, 1);
scheduler(); // start running processes
We ended boot by starting a scheduler
Remember the stack of the boot process?

Kernel Stack of a boot process (4K)

- EIP (main)
- ...
- EIP (mpmain)
- ...

How does scheduler start?

- Chooses next process to run
- Switches to it
  - From the current context
• So when the scheduler context switched the first time
  
  2478  swtch(&cpu->scheduler,  
           proc->context);

• It saved its own context
  
  &cpu->scheduler

• And restored a context of the first process
  
  proc->context
This is how stack looked like when scheduler() invoked swtch() for the first time.

Kernel Stack of a boot process (4K):
- EIP (main)
- ...
- EIP (mpmain)
- ...
- &cpu->scheduler
- proc->context
- EIP (scheduler)

What is this context?
This is how stack looked like when scheduler() invoked swtch() for the first time.

- **What is this context?**
- It's the context of the first process scheduler decides to run.
Save context of the scheduler
2958  swtch:
2959  movl 4(%esp), %eax  // struct context **old
2960  movl 8(%esp), %edx  // struct context *new
2961
2962 # Save old callee-save registers
2963  pushl %ebp
2964  pushl %ebx
2965  pushl %esi
2966  pushl %edi
2967
2968 # Switch stacksh
2969  movl %esp, (%eax)  // load current context (top of current stack) into
2970      // the memory location pointed by *old
2971  movl %edx, %esp  // set stack to be equal to *new (the top of the new context)
2972
2973 # Load new callee-save registers
2974  popl %edi
2975  popl %esi
2976  popl %ebx
2977  popl %ebp
2978  ret
This is why the context is the top of some stack

- Initially it was the stack of `mpenter()`
  - On which scheduler started
- Then first process...
  - Then scheduler again
  - And the next process...
Back to main context switch: so context is always top of some stack
Currently the *new context is the stack of the scheduler

We switch to the scheduler,

It runs on the stack of the boot process
2958  swtch:
2959  movl 4(%esp), %eax   // struct context **old
2960  movl 8(%esp), %edx   // struct context *new

2961
2962  # Save old callee-save registers
2963  pushl %ebp
2964  pushl %ebx
2965  pushl %esi
2966  pushl %edi

2967
2968  # Switch stacks
2969  movl %esp, (%eax)   // load current context (top of current stack) into
2970  movl %edx, %esp     // the memory location pointed by *old
2971
2972  # Load new callee-save registers
2973  popl %edi
2974  popl %esi
2975  popl %ebx
2976  popl %ebp
2977  ret
And now: exit from swtch()
Where does this swtch() return?
Where does this switch() return?

- Scheduler
- After all remember
  - We started with timer interrupt
  - Entered the kernel
  - Entered schedule()
  - Entered switch
- And are currently on our way from the process into the scheduler
What does scheduler do?

- Chooses next process to run
- Switches to it
What does stack look like when scheduler() invokes swtch()?

Kernel Stack of a boot process (4K)

EIP (main)
...
EIP (mpmain)
...
&cpu->scheduler
proc->context
EIP (scheduler)

What is this context?

CPU
Scheduler
Proc
Context

Context

EIP (sched)
EBP
EBX
ESI
EDI
What does stack look like when scheduler() invokes swtch()?

- What is this context?
- Right the context of the next process to run
• We save the context of the scheduler again
• Restore the context of the next process
• Remember, from inside the scheduler we invoked `swtch()` as

```
2478  swtch(&cpu->scheduler,
            proc->context);
```

• Hence, we save context of the scheduler into

```
&cpu->scheduler
```

• And restore

```
proc->context
```
Stacks and context inside `swtch()`

Kernel Stack of a boot process (4K)

User state (saved by hardware)

System Stack of a process (4K)

Trap frame

Context
Exiting back to user-level

- Stack inside sched()
- Normal returns until back to alltrap()
alltraps(): Exiting back into user level process
Stack after trap() returns

User state (saved by hardware)

vector32

alltraps

ESP

Kernel Stack of a process (4K)

Trap frame
alltraps(): exiting

- Restore all registers
- Exit into user
  - iret

3004 alltraps:
...
3020 # Call trap(tf), where tf=%esp
3021 pushl %esp
3022 call trap
3023 addl $4, %esp
3024
3025 # Return falls through to trapret...
3026 .globl trapret
3027 trapret:
3028 popal
3029 poopl %gs
3030 poopl %fs
3031 poopl %es
3032 poopl %ds
3033 addl $0x8, %esp # trapno and errcode
3034 iret
We're back to where we started, but in a new process.
Summary

- We switch between processes now
Thank you