Operating Systems

Lecture: Context switch

Anton Burtsev
A bit of recap...
When OS context switches between processes?
When OS context switches between processes?

- Timer interrupt preempts the current process
- A process enters the kernel with a system call and has to wait on some resource
  - E.g., write to a pipe, but the pipe is full
- The process voluntarily yields CPU with the `yield()` system call
Let's look at timer interrupt
3351 trap(struct trapframe *tf) {

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;

if(proc && proc->state == RUNNING
    && tf->trapno == T_IRQ0+IRQ_TIMER)
3424     yield();
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 && tf−>trapno == T_IRQ0+IRQ_TIMER)
3424     yield();
Invoke the scheduler

    2777    yield(void)
    2778    {
    2779         acquire(&ptable.lock);
    2780         proc->state = RUNNABLE;
    2781         sched();
    2782         release(&ptable.lock);
    2783    }
Start the context switch

2758 sched(void)
2759 {
...
2771   swtch(&proc->context,
          cpu->scheduler);
...
2773 }
But what do you think needs to happen inside `switch()`?
What should be in the box? i.e., what is the state of the process?
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
  struct context *context; // swtch() here to run
  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)
};
What about general registers?
Save them on the stack

2093 struct context {
2094   uint edi;
2095   uint esi;
2096   uint ebx;
2097   uint ebp;
2098   uint eip;
2099 };
Process 1

Process 2
Back to the timer interrupt path
(keep track of what happens to the stack, lets see how everything gets packed in the “box”!)
- **User mode**
- **Two stacks**
  - Kernel and user
  - Kernel stack is empty
- Page table
- GDT
Timer interrupt

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

Code, data, heap

Timer Interrupt
Interrupt path

User state (saved by hardware):
- SS
- ESP
- EFLAGS
- CS
- EIP

User stack of a process (can grow up to 2GBs):
- Argument 1
- Argument 2
- Calling EIP ++
- Old EBP
- Local variables
- Saved local values, e.g., push EAX, etc

Last stack frame

Kernel Stack of a process (4K)

ESI

Page table
- Level 1
- Level 2

Interrupt Vector #

GDT
- NULL: 0x0
- KCODE: 0 - 4GB
- KDATA: 0 - 4GB
- _K_CPU: 4 bytes
- CODE: 0 - 4GB
- DATA: 0 - 4GB
- TSS: sizeof(tss)

IDT

TSS

Vector code

Timer:IRQ0 -> vector 32
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)

vector32

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

vector32
- DS
- ES
- FS
- GS
- All registers
- ESP

ESP

Kernel Stack of a process (4K)

Call stack: vector32()
alltraps()

Trap frame
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
trap(struct trapframe *tf) {
  ...
  switch(tf->trapno) {
  case T_IRQ0 + IRQ_TIMER:
    if(cpu->id == 0) {
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    break;
  ...
  if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();
}
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     }
3371   break;
3372   break;
...
3423   if(proc && proc−>state == RUNNING
3424        && tf−>trapno == T_IRQ0+IRQ_TIMER)
3425      yield();
2777  yield(void)
2778  {
2779     acquire(&ptable.lock);
2780     proc->state = RUNNABLE;
2781     sched();
2782     release(&ptable.lock);
2783  }
2758 sched(void) 
2759  {
 ... 
2771  swtch(&proc->context, 
       cpu->scheduler);
 ... 
2773  }
Stack inside `swtch()` and its two arguments (passed on the stack)

<table>
<thead>
<tr>
<th>Call stack:</th>
<th>vector32()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>alltraps()</td>
</tr>
<tr>
<td></td>
<td>trap()</td>
</tr>
<tr>
<td></td>
<td>yield()</td>
</tr>
<tr>
<td></td>
<td>sched()</td>
</tr>
<tr>
<td></td>
<td>switch(&amp;proc-&gt;context, cpu-&gt;scheduler)</td>
</tr>
</tbody>
</table>

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
Remember you have two stacks (P1 and P2)

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)

vector32

alltraps

trap

tyield

sched

Kernel Stack of a process (4K)

Kernel Stack of a boot process (4K)

Trap frame

Proc

Context

P1

P2
The context switch function should pack everything what is left in the box and switch stacks, i.e., save the pointer to the old stack (P1) and load the new stack (P2)
What is left unsaved?
Callee saved registers

User state (saved by hardware)
- SS
- ESP
- EFLAGS
- CS
- EIP
- 0
- 32

vector32
- DS
- ES
- FS
- GS
- All registers
- ESP
- EIP (alltraps)
- ...
- EIP (trap)

alltraps
- ... EIP (yield)

trap
- ...

yield
- ...

sched
- &proc->context
- cpu->scheduler
- EIP (sched)

Kernel Stack of a process (4K)

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

Proc

Context
- EIP (line: 2479)
- EBP
- EBX
- ESI
- EDI
swtch: save registers on the stack

```
2958  swtch:
2959  movl 4(%esp), %eax
2960  movl 8(%esp), %edx

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

2967  # Switch stacksh
2968  movl %esp, (%eax)
2969  movl %edx, %esp

2971  # Load new callee-save registers
2972  popl %edi
2973  popl %esi
2974  popl %ebx
2975  popl %ebp
2976  popl %ebp
2977  ret
```

```
2093 struct context {
2094   uint edi;
2095   uint esi;
2096   uint ebx;
2097   uint ebp;
2098   uint eip;
2099  };
```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
Main trick: context is always saved on the top of a stack
And the context switch just saves the old context and loads the new
Save the pointer to the stack of the old process (P1) inside the proc data structure (in the context field)
swtch:
2959 movl 4(%esp), %eax       // struct context **old
2960 movl 8(%esp), %edx       // struct context *new

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

# Switch stacks
2969 movl %esp, (%eax)       // load current context (top of current stack) into
                              // the memory location pointed by *old
2970 movl %edx, %esp         // set stack to be equal to *new (the top of the new context)

# Load new callee-save registers
2973 popl %edi
2974 popl %esi
2975 popl %ebx
2976 popl %ebp
2977 ret
Now you can simply load the new context (P2) into ESP and continue returning on that new stack.
Save context of the old process

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

Kernel Stack of a process (4K)

Trap frame

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

Proc

Context

Context

ESP
swtch(): load next context

- Load address of the next context (it's in %edx) into %esp
Load stack of the next process

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

vector32
- ESP
- EIP (alltraps)
- ...
- EIP (trap)
- ...
- EIP (yield)
- ...
- &proc->context
cpu->scheduler
- EIP (sched)
- EBP
- EBX
- ESI
- EDI

Kernel Stack of a process (4K)

Trap frame

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

Proc

Context

Context

ESP
Remember: The context switch function should just save the pointer to the old stack (P1) and load the new stack (P2)
swtch:
   movl 4(%esp), %eax
   movl 8(%esp), %edx

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

# Switch stacks
   movl %esp, (%eax)
   movl %edx, %esp

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

Now: exit from swtch()
Where does this switch() return?
Context is always top of some stack...ok, but how?

- How does initialization of each CPU end?
main(void)
{
    kinit1(end, P2V(4*1024*1024)); // phys page allocator
    kvmalloc(); // kernel page table
    mpinit(); // detect other processors
    seginit(); // segment descriptors
    tvinit(); // trap vectors
    userinit(); // first user process
    mpmain(); // finish this processor’s setup
}
// 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 the scheduler
scheduler(void)
{
  for(;;){
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();
      proc = 0;
    }
  }
}

Scheduler()

- Chooses next process to run
- Switches to it
  - From the current context
struct cpu {
    uchar apicid; // Local APIC
    struct context *scheduler; // swtch() here to enter scheduler
    struct taskstate ts; // TSS
    struct segdesc gdt[NSEGS]; // x86 global descriptor table
    volatile uint started; // Has the CPU started?
    int ncli; // Depth of pushcli nesting.
    int intena; // Were interrupts enabled ...
    struct proc *proc; // The process running on this cpu
};

extern struct cpu cpus[NCPU];
extern int ncpu;
This is how the stack looked after boot finished, i.e., inside `mpmain()`

Kernel Stack of a boot process (4K)

<table>
<thead>
<tr>
<th>EIP (main)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>EIP (mpmain)</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
• So when the scheduler context switched the first time
  2478  swtch(&cpu->scheduler, 
           proc->context);
• We save the current context of the scheduler into:
  &cpu->scheduler
• And restore the 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)

Which process is this context?

Context:
- EIP (sched)
- EBP
- EBX
- ESI
- EDI
This is how stack looked like when scheduler() invoked switch() for the first time.

- Which process is this context?
- It's the context of the first process scheduler decides to run.
allocproc(void)
{
  ...
  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
  ...
}

Context is configured as top of the stack when new process is created inside allocproc() function

• Remember exec()?
Save context of the scheduler
swtch:

2959 movl 4(%esp), %eax  // struct context **old
2960 movl 8(%esp), %edx  // struct context *new

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

2968 # Switch stacks
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 # Load new callee-save registers
2973 popl %edi
2974 popl %esi
2975 popl %ebx
2976 popl %ebp
2977 ret
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 the context switch
(we now know we return into the scheduler)
Where does this `swtch()` 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)

Stack (context) of the scheduler

CPU

Scheduler

Proc

Context

Context

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

- Where does the proc->context point?
What does stack look like when scheduler() invokes swtch()?

- Where does the `proc->context` point?
  - Right the context (stack) of the next process to run
• We save the context of the scheduler
• 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 contexts inside the `swtch()`
Exiting back to user-level

- Stack of the process after context switch, i.e., inside sched()
- Return as usual all the way to alltrap()
alltraps(): exit into user-level
We're back to where we started, but in a new process.
Summary

- We switch between processes now
Thank you