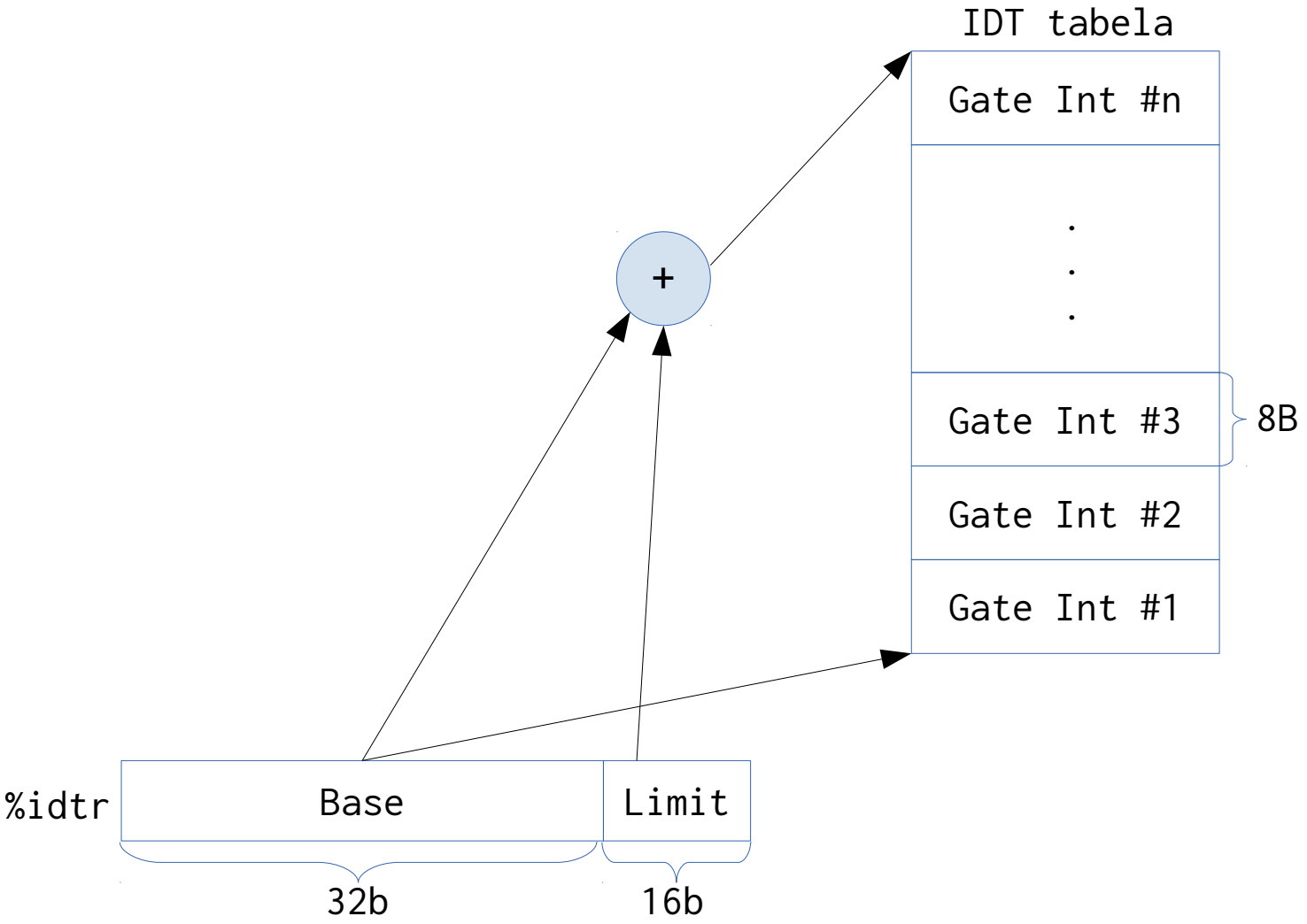
Operativni sistemi

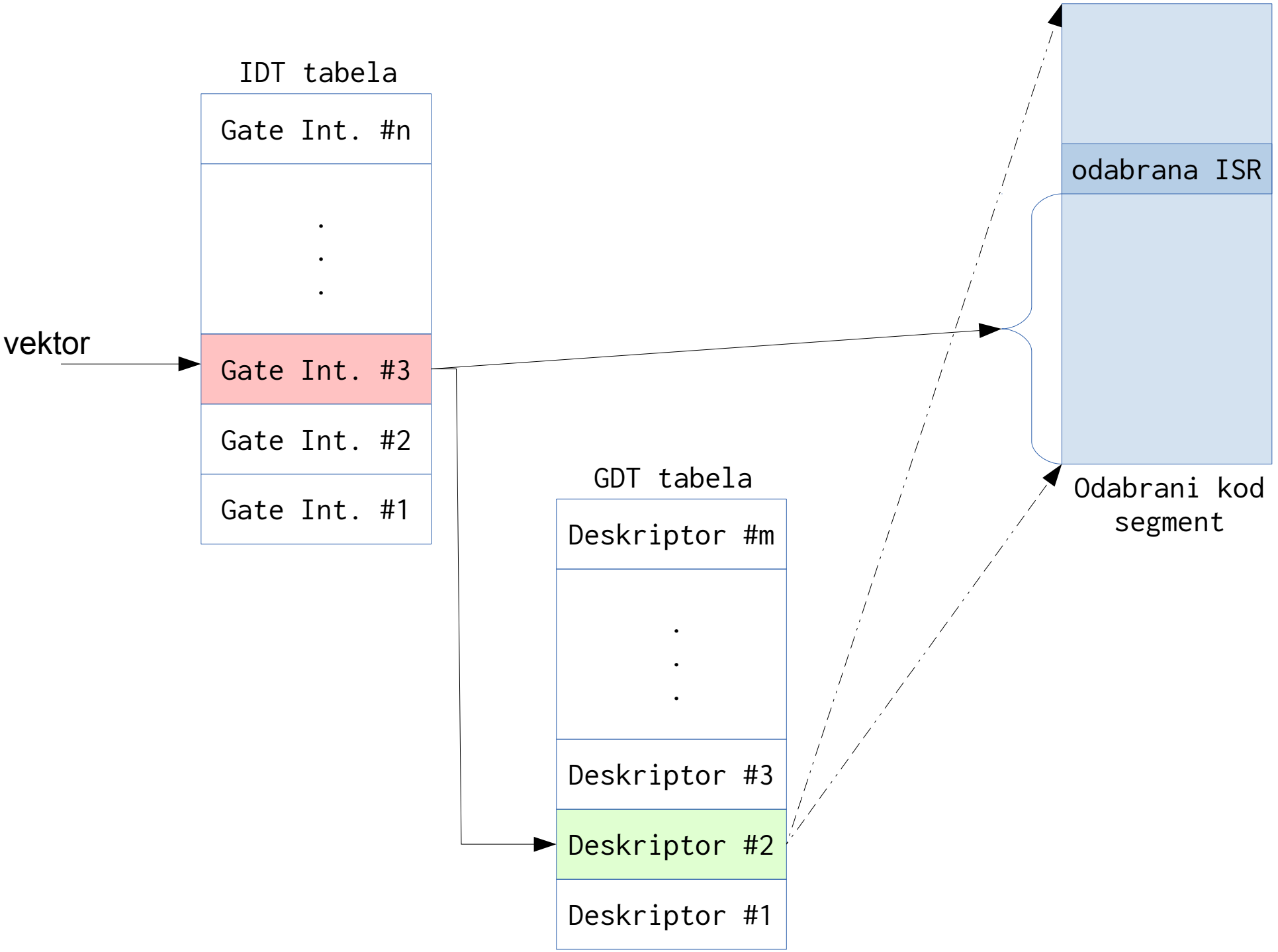
dr.sc. Amer Hasanović

Tretman iznimki i sistemskih poziva

Vektor prekida

- Svaki prekid, bez obzira na tip, asocira se sa brojem, tzv **vektor**, koji identificira konkretni prekid.
  - vektore `0-32` x86 rezervira za iznimke
  - ostali su konfigurabilni za hardver prekide i sistemske pozive
- Kada nastane prekid, na osnovu vrijednosti vektora bira se odgovorajaća rutina koja tretira prekid:
  - vektorom se indeksira IDT (Interrupt Descriptor Table), a iz dobivenog deskriptora (ili gate-a) čita se destinacijski `%cs` i `%eip`;
  - adresu IDT-a CPU drži u posebnom registru `%idtr`, čija se vrijednost mijenja instrukcijom `lidt`
  - operativni sistem u procesu inicijalizacije mora kreirati tabelu koja definira gate-e za sve moguće prekide, te istu učitati u svako aktivno jezgro CPU-a sa `lidt`

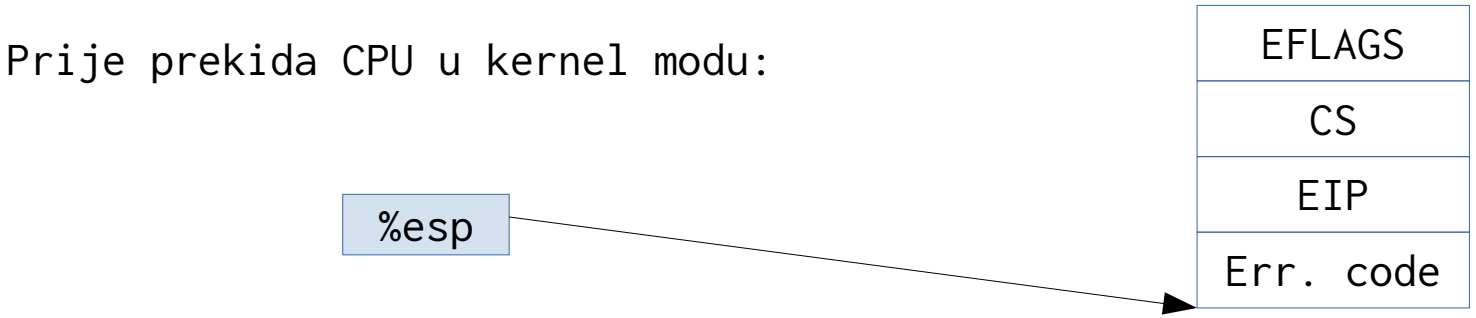

IDT tabela
Gate Int #n
.
.
.
Gate Int #3
8B
Gate Int #2
Gate Int #1
+
%idtr
Base
Limit
32b
16b

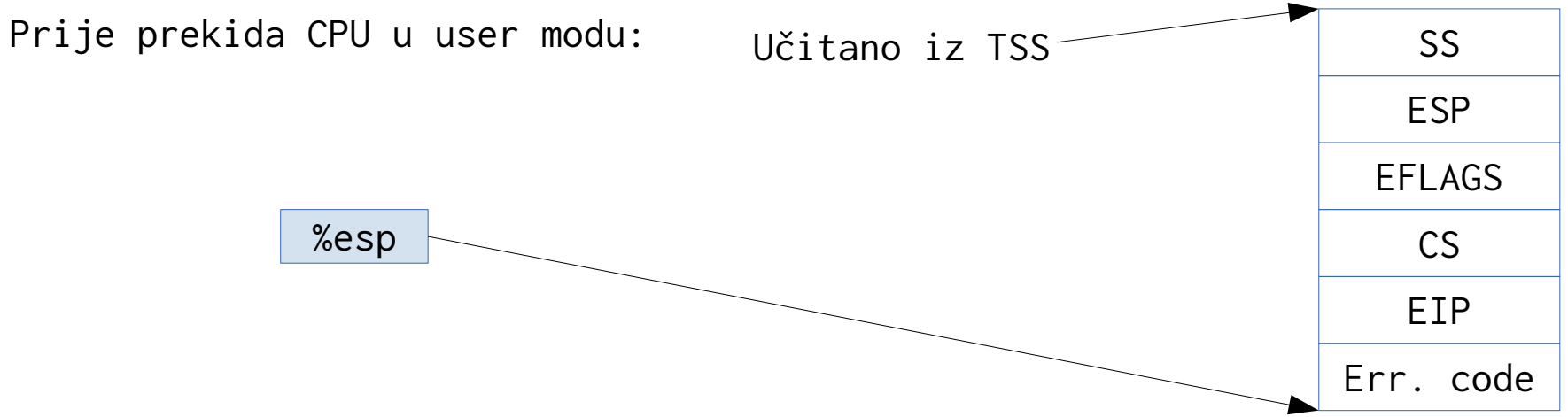

IDT tabela
Gate Int. #n
.
.
.
vektor
Gate Int. #3
Gate Int. #2
Gate Int. #1
GDT tabela
Deskriptor #m
.
.
.
Deskriptor #3
Deskriptor #2
Deskriptor #1
odabrana ISR
Odabrani kod segment

xv6 i IDT

- `xv6` inicijalizira `IDT` u funkciji `tvinit`
- Vrijednost svakog pojedinačnog gate-a unutar IDT postavlja se putem makroa `SETGATE`
- `SETGATE(gate, istrap, sel, off, d)`
  - `gate` – red u IDT tabeli koji se definira
  - `istrap` – primanje hardver prekida tokom tretmana datog prekida (1 → da, 0 → ne)
  - `sel` – asocirani segment deskriptor za dati prekid (%cs)
  - `off` – adresa asoricranog ISR (%eip)
  - `d` – DPL polje unutar gate-a koje definira minimalni nivo privilegija potreban da se prekid asociran sa datim vektorom generira softverski putem `int` instrukcije

- `xv6` definira ISR rutine za sve prekide u fajlu `vectors.S`
  - Obratiti pažnju da hardver na steku generira `Error code` polje za prekide asocirane sa vektorima `8`, `10-14`, `17`, dok za ostale vektore njihovi ISR na stek stavljaju umjetni `Error code`





trap.c

```
struct gatedesc idt[256];

extern uint vectors[];

void tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

  initlock(&tickslock, "time");
}
```

```
#define T_SYSCALL       64
```

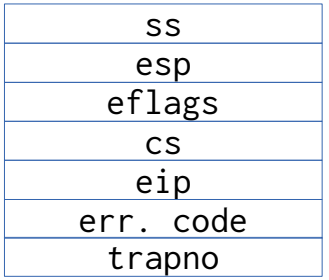
vectors.S

```
.data
.globl vectors
vectors:
  .long vector0
  .long vector1
  .long vector2
  .long vector3
  .long vector4
  .long vector5
  .long vector6
  .long vector7
  .long vector8
…

  .long vector254
  .long vector255
```

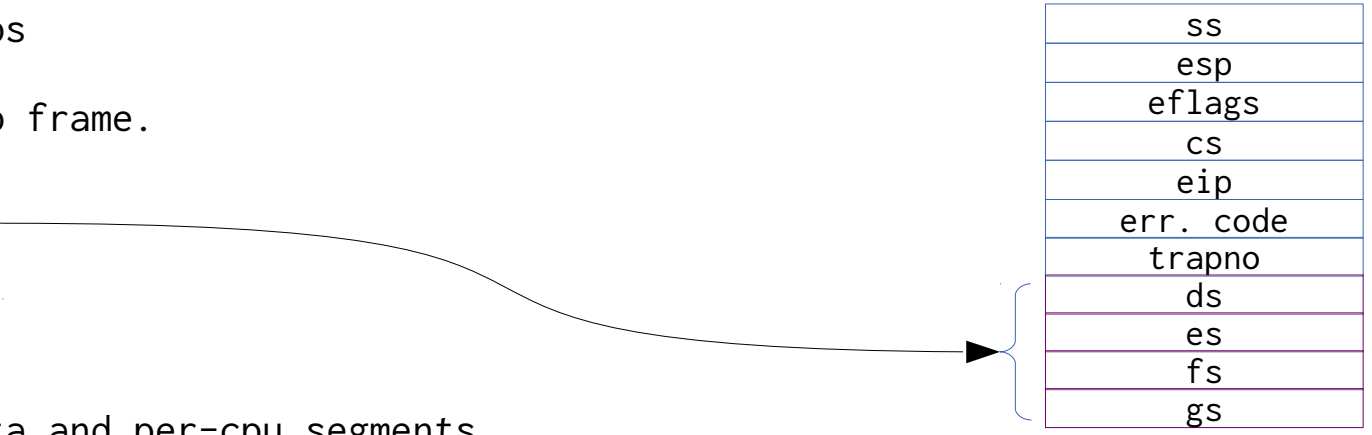
```
.globl vector0
vector0:
  pushl $0
  pushl $0
  jmp alltraps
.globl vector1
vector1:
  pushl $0
  pushl $1
  jmp alltraps
…

vector8:
  pushl $8
  jmp alltraps
```

```
…

.globl vector254
vector254:
  pushl $0
  pushl $254
  jmp alltraps
.globl vector255
vector255:
  pushl $0
  pushl $255
  jmp alltraps
```

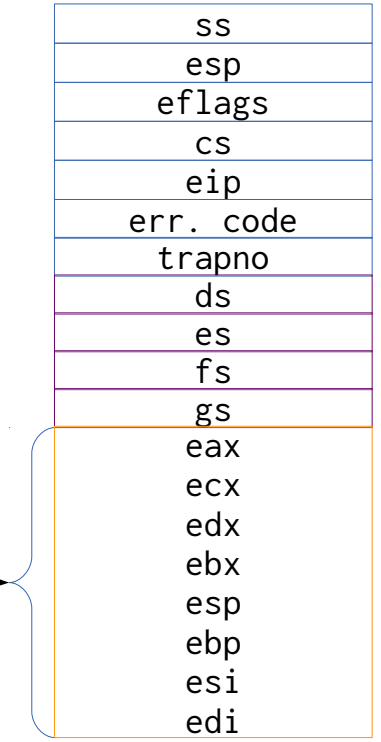
```
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # 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
  addl $4, %esp

  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```



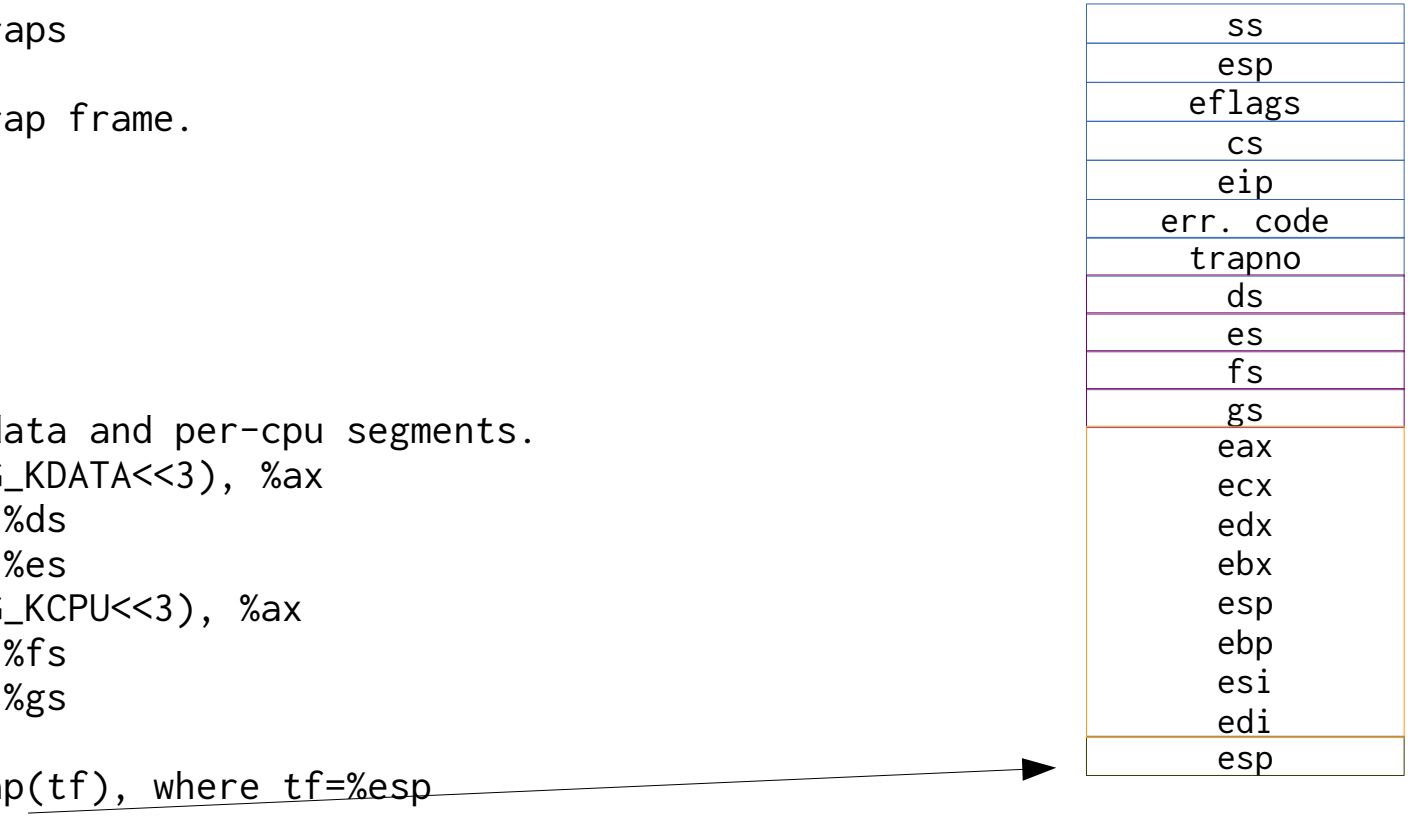
```
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # 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
  addl $4, %esp

  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```



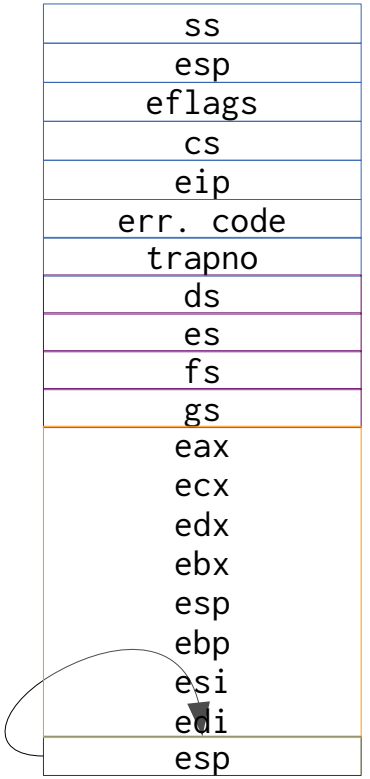
```
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # 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
  addl $4, %esp

  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```



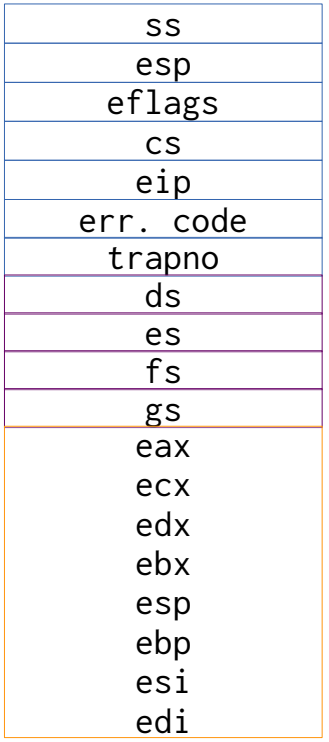
```
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # 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
  addl $4, %esp

  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```



```
.globl 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
  addl $4, %esp

  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```



```
struct trapframe {

  uint edi;
  uint esi;
  uint ebp;
  uint oesp;      // useless & ignored
  uint ebx;
  uint edx;
  uint ecx;
  uint eax;


  ushort gs;
  ushort padding1;
  ushort fs;
  ushort padding2;
  ushort es;
  ushort padding3;
  ushort ds;
  ushort padding4;
  uint trapno;


  uint err;
  uint eip;
  ushort cs;
  ushort padding5;
  uint eflags;


  uint esp;
  ushort ss;
  ushort padding6;
};
```



```
void trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL) {
    if(proc->killed)
      exit();
    proc->tf = tf;
    syscall();
    if(proc->killed)
      exit();
    return;
  }
  switch(tf->trapno) {
  case T_IRQ0 + IRQ_TIMER:
    if(cpu->id == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_IDE:
    ideintr();
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_IDE+1:
    // Bochs generates spurious IDE1 interrupts.
    break;
  case T_IRQ0 + IRQ_KBD:
    kbdintr();
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_COM1:
    uartintr();
    lapiceoi();
    break;
  case T_IRQ0 + 7:
  case T_IRQ0 + IRQ_SPURIOUS:
    cprintf("cpu%d: spurious interrupt at %x:%x\n",
            cpu->id, tf->cs, tf->eip);
    lapiceoi();
    break;
```

tretman sistemskih poziva

tretman hardver prekida

```
// nastavak sa prethodnog slajda
  default:
    if(proc == 0 || (tf->cs&3) == 0) {
      // In kernel, it must be our mistake.
      cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
              tf->trapno, cpu->id, tf->eip, rcr2());
      panic("trap");
    }
    // In user space, assume process misbehaved.
    cprintf("pid %d %s: trap %d err %d on cpu %d "
            "eip 0x%x addr 0x%x--kill proc\n",
            proc->pid, proc->name, tf->trapno, tf->err, cpu->id, tf->eip,
            rcr2());
    proc->killed = 1;
  }
```

tretman iznimki

sistemski pozivi

- `xv6` koristi vektor `64` (`T_SYSCALL`) za signalizaciju sistemskog poziva.
- ukoliko neka aplikacija želi da pozove neku od kernel funkcija eksportovanih u korisnički prostor kroz mehanizam sistemskih poziva, aplikacija treba da obavi korake:
  - broj sistemskog poziva koji `xv6` asocira sa željenom kernel funkcijom postaviti u registar `%eax`;
  - argumente za kernel funkciju postaviti na stek aplikacije;
  - izvršiti sistemski poziv sa instrukcijom: `int 64`

- Svaki sistemski poziv ima dva dijela:
  - funkciju za pripremu poziva unutar korisničkog prostora:
    - kompajlira se i linka skupa sa aplikacijom;
    - `xv6` relevantni fajlovi: `user.h` i `usys.S`
  - funkciju koja predstavlja implementaciju sistemskog poziva
    - ove funkcije kompajliraju se i linkaju sa ostalim kernel funkcijama;
    - svaka funkcija treba da dobije identifikator koji predstavlja broj sistemskog poziva;
    - `xv6` relevantni fajlovi: `syscall.c` i `sysproc.c`

Sigurnost sistemskog poziva

- Argumente za sistemski poziv aplikacija ostavlja na svom steku.
  - kada se aktivira kernel nakon prekida kod sistemskog poziva, on ima pristup i kernel i user dijelu AP-a procesa;
  - prije upotrebe argumenata, kernel mora provjeriti validnost istih jer eventualna iznimka (npr page fault) u kernel prostoru može izazvati zaustavljanje cijelog OS-a
    - xv6 za provjeru argumenata koristi funkcije: `argint`, `argptr`, `argstr` itd..

```
int argint(int n, int *ip)
{
  return fetchint(proc->tf->esp + 4 + 4*n, ip);
}
```

```
int fetchint(uint addr, int *ip)
{
  if(addr >= proc->sz || addr+4 > proc->sz)
    return -1;
  *ip = *(int*)(addr);
  return 0;
}
```

user.h

```
// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
int write(int, void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(char*, int);
int mknod(char*, short, short);
int unlink(char*);
int fstat(int fd, struct stat*);
int link(char*, char*);
int mkdir(char*);
int chdir(char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
```

usys.S

```
.globl fork; fork: movl $1, %eax; int $64; ret
.globl exit; exit: movl $2, %eax; int $64; ret
.globl wait; wait: movl $3, %eax; int $64; ret
.globl pipe; pipe: movl $4, %eax; int $64; ret
.globl read; read: movl $5, %eax; int $64; ret
.globl write; write: movl $16, %eax; int $64; ret
.globl close; close: movl $21, %eax; int $64; ret
.globl kill; kill: movl $6, %eax; int $64; ret
.globl exec; exec: movl $7, %eax; int $64; ret
.globl open; open: movl $15, %eax; int $64; ret
.globl mknod; mknod: movl $17, %eax; int $64; ret
.globl unlink; unlink: movl $18, %eax; int $64; ret
.globl fstat; fstat: movl $8, %eax; int $64; ret
.globl link; link: movl $19, %eax; int $64; ret
.globl mkdir; mkdir: movl $20, %eax; int $64; ret
.globl chdir; chdir: movl $9, %eax; int $64; ret
.globl dup; dup: movl $10, %eax; int $64; ret
.globl getpid; getpid: movl $11, %eax; int $64; ret
.globl sbrk; sbrk: movl $12, %eax; int $64; ret
.globl sleep; sleep: movl $13, %eax; int $64; ret
.globl uptime; uptime: movl $14, %eax; int $64; ret
```

syscall.c

```
void syscall(void)
{
  int num;
  num = proc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    proc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            proc->pid, proc->name, num);
    proc->tf->eax = -1;
  }
}
```

```
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,
};
```

```
#define SYS_getpid 11
```

sysproc.c

```
int sys_getpid(void)
{
  return proc->pid;
}
```