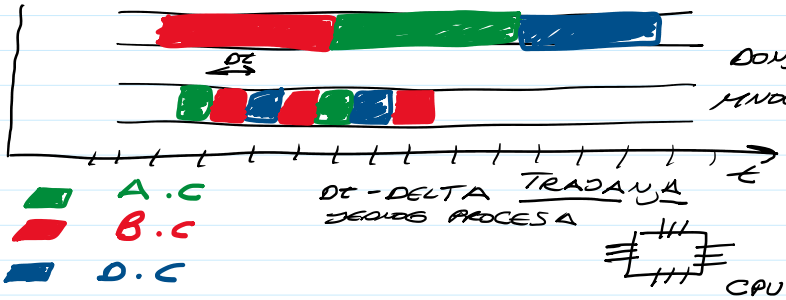
Friday, April 7, 2023 7:59 PM

PROCESI I RASPOREDIVAČ

- NA OS-U MOŽEMO IMATI VIŠE PROCESA (A I PARALELNO)
- PROGRAM SA SVIM SVOJIM PODACIMA, FAJLOVIM U MEMORIJI ČINI JEDAN PROCES



DOBAR PRISTUP JE MNOGO EFIKASNIJI I BOLJI ZA KORISNIKA

TIKOVI

dt - DELTA TRAJANJA JEDNOG PROCESA

CPU

- XV6 "MOŽE" DA RADI SA VIŠE PROCESORA
- TIKOVI - JEDINICA VREMENA NA MATIČNOJ PLOČI
- DA BI SE OMOGUĆILO OVO "CIJEPKANJE" PROCESA I NASTAVLJANJE MORAMO DA KORISTIMO SCHEDULER

RASPOREDIVAČ

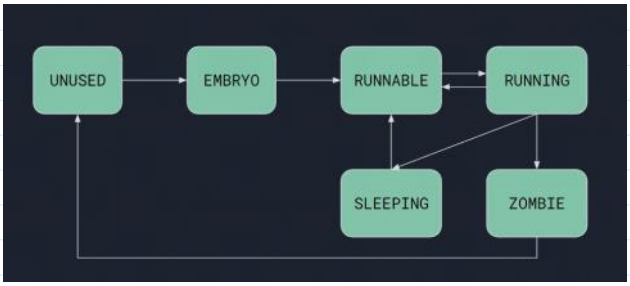
XV6 MOŽE DA IMA MAX 64 AKTIVNA PROCESA

STRUKTURA PROC OPISUJE AKTIVNI PROCES

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// 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
  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)
};
```

CONTEXT SLUŽI ZA PAMĆENJE NAREDNE JEDNOG PROCESA

BITNO ZA NASTAVAK ILI STOPIRANJE PROCESA U CPU



ŽIVOTNI VIJEK PROCESA U XV6

UNUSED - PROCES KOJI UOPŠTE NIJE BIO UPOTREBLJEN

AKO ZA NEKI PROCES IMAMO PROCSTATE[UNUSED], TO ZNAČI DA U NIZU OD 64 PROCESA ZA XV6, MOŽEMO DA GA ZAMIJENIMO SA NEKIM DRUGIM PROCESOM

EMBRYO - NAGLAŠAVAMO DA ĆEMO PROCES DA KORISTIMO

RUNNABLE ⟷ RUNNING, PRELAZIMO IZ JEDNOG U DRUGI

- SCHEDULER KORISTI ROUND ROBIN ALGORITAM KROZ NIZ PROCESA

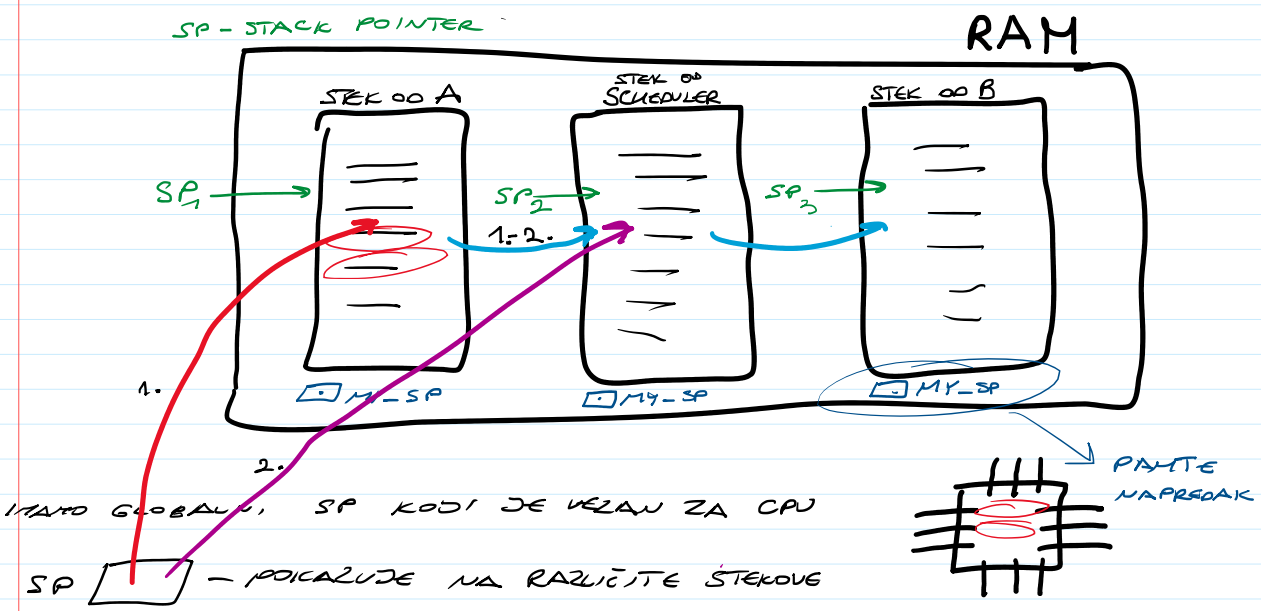
- exit() FUNKCIJA STAVLJA RUNNING PROCES U ZOMBIE STANJE

- SVI PROCESI U XV6 IMAJU PARENT-CHILD ODNOSE.

- U TOJ HIJERARHIJI PRVI PROCES JESTE ZAPRAVO

INIT PROCES, POSLIJE/ISPOD NJEGA IDE SHELL PROCES

$



IMAMO GLOBALNI SP KOJI JE VEZAN ZA CPU

SP — POKAZUJE NA RAZLIČITE STEKOVE

TRAP METODA JE HANDLER SVIH PREKIDA

TRAP.C

```
    // Force process exit if it has been killed and is in user space.
    // (If it is still executing in the kernel, let it keep running
    // until it gets to the regular system call return.)
    if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
        exit();

    // Force process to give up CPU on clock tick.
    // If interrupts were on while locks held, would need to check nlock.
    if(myproc() && myproc()->state == RUNNING &&
       tf->trapno == T_IRQ0+IRQ_TIMER)
        yield();

    // Check if the process has been killed since we yielded
    if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
        exit();
}
```

```
struct {
    struct spinlock lock;
    struct proc proc[NPROC];
} ptable;
```

64 STAVKI

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
    acquire(&ptable.lock);  //DOC: yieldlock
    myproc()->state = RUNNABLE;
    sched();
    release(&ptable.lock);
}
```

PROC. C

```
void
sched(void)
{
    int intena;
    struct proc *p = myproc();

    if(!holding(&ptable.lock))
        panic("sched ptable.lock");
    if(mycpu()->ncli != 1)
        panic("sched locks");
    if(p->state == RUNNING)
        panic("sched running");
    if(readeflags()&FL_IF)
        panic("sched interruptible");
    intena = mycpu()->intena;
    swtch(&p->context, mycpu()->scheduler);
    mycpu()->intena = intena;
}
```

USLOVI KOJI SE NE TREBAJU DESITI :D

SWTCH.S (ASEMBLI)

```
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
# Save the current registers on the stack, creating
# a struct context, and save its address in *old.
# Switch stacks to new and pop previously-saved registers.

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

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

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

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

[ESP+4] <- OLD

NEW

MIJENJANJE VRIJEDNOSTI

POPUJEMO SA SCHEDULERA

SRŽ XV6 OPERATIVNOG SISTEMA JE BESKONAČNA PETLJA

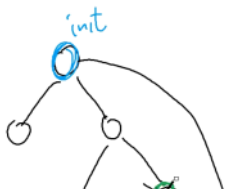
```
void
scheduler(void)
{
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0;

    for(;;){
        // Enable interrupts on this processor.
        sti();

        // Loop over process table looking for process to run.
        acquire(&ptable.lock);
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->state != RUNNABLE)
                continue;

            // Switch to chosen process.  It is the process's job
            // to release ptable.lock and then reacquire it
            // before jumping back to us.
            c->proc = p;
            switchuvm(p);
            p->state = RUNNING;

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

            // Process is done running for now.
            // It should have changed its p->state before coming back.
            c->proc = 0;
        }
        release(&ptable.lock);

    }
}
```

"POD ZNACIMA NALOGA DA SE TELEPORTUJEM"

PAUZA -11-

1:26:00

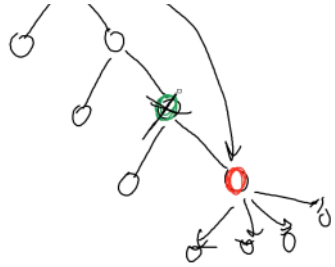
```
allocproc(void)
{
    struct proc *p;
    char *sp;

    acquire(&ptable.lock);

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        if(p->state == UNUSED)
            goto found;

    release(&ptable.lock);
    return 0;

found:
    p->state = EMBRYO;
    p->pid = nextpid++;

    release(&ptable.lock);

    // Allocate kernel stack.
    if((p->kstack = kalloc()) == 0){
        p->state = UNUSED;
        return 0;
    }
    sp = p->kstack + KSTACKSIZE;

    // 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.
```



14-2008: Vežbe 10 5.4.2023. (raf.edu.rs)

Init.c

```
int
main(void)
{
    int pid, wpid;

    if(getpid() != 1){
        fprintf(2, "init: already running\n");
        exit();
    }

    if(open("/dev/console", O_RDWR) < 0){
        mknod("/dev/console", 1, 1);
        open("/dev/console", O_RDWR);
    }
    dup(0);  // stdout
    dup(0);  // stderr

    for(;;){
        printf("init: starting sh\n");
        pid = fork();
        if(pid < 0){
```



```
    dup(0);  // stdout
    dup(0);  // stderr

    for(;;){
        printf("init: starting sh\n");
        pid = fork();
        if(pid < 0){
            printf("init: fork failed\n");
            exit();
        }
        if(pid == 0){
            exec("/bin/sh", argv);
            printf("init: exec sh failed\n");
            exit();
        }
        while((wpid=wait()) >= 0 && wpid != pid)
            printf("zombie!\n");
    }
}
```

Exit funkcija mnogo bitna

```
begin_op();
iput(curproc->cwd);
end_op();
curproc->cwd = 0;

acquire(&ptable.lock);

// Parent might be sleeping in wait().
wakeup1(curproc->parent);

// Pass abandoned children to init.
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == curproc){
        p->parent = initproc;
        if(p->state == ZOMBIE)
            wakeup1(initproc);
    }
}

// Jump into the scheduler, never to return.
curproc->state = ZOMBIE;
sched();
panic("zombie exit");
```