

Operativni sistemi

dr.sc. Amer Hasanović

Preemptivna koordinacija procesa

- Nakon što se obavi promjena konteksta i kernel preda kontrolu nekom procesu, proces dobija potpunu kontrolu nad nekom CPU jezgrom.
- Da bi kernel ponovo preuzeo kontrolu nad istom CPU jezgrom potrebno je da tajmer, integriran direktno na jezgri, generira prekide u jednakim vremenskim intervalima.
 - xv6 u funkciji `lapicinit` inicijalizira tajmer za periodične prekide frekvencijom 100Hz;
 - xv6 u funkciji `trap` tretira tajmer prekid na način da poziva funkciju `yield`, čime se trenutni proces stavlja u `RUNNABLE` stanje, prelazi na scheduler stek i izvršava funkcija `scheduler`.
 - tokom tretmana tajmer prekida inkrementira se i globalna varijabla `ticks`

```

void trap(struct trapframe *tf) {
    ...

    switch(tf->trapno){
    case T_IRQ0 + IRQ_TIMER:
        if(cpu->id == 0){
            acquire(&tickslock);
            ticks++;
            wakeup(&ticks);
            release(&tickslock);
        }
        lapiceoi();
        break;

        ...

    }

    ...

    if(proc && proc->killed && (tf->cs&3) == DPL_USER)
        exit();

    if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
        yield();

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

```

Kreiranje novih procesa

- Svi procesi, sem `init` procesa, kreiraju se pomoću sistemskog poziva `fork`
 - kada neki proces pozove `fork`, kreira se novi proces:
 - proces koji je pozvao `fork`, naziva se *roditelj* (parent), kreirani proces je *dijete* (child);
 - dijete dobija svoj adresni prostor u kome su sve stranice kopirane iz roditelja;
 - dijete će biti u `RUNNABLE` stanju;
 - `fork` vraća `PID` djeteta u roditelj procesu, a vrijednost `0` u dijete procesu.

```

int fork(void)
{
    int i, pid;
    struct proc *np;
    if((np = allocproc()) == 0)
        return -1;

    if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
        kfree(np->kstack);
        np->kstack = 0;
        np->state = UNUSED;
        return -1;
    }
    np->sz = proc->sz;
    np->parent = proc;
    *np->tf = *proc->tf;

    np->tf->eax = 0;

    for(i = 0; i < NOFILE; i++)
        if(proc->ofile[i])
            np->ofile[i] = filedup(proc->ofile[i]);
    np->cwd = idup(proc->cwd);

    safestrcpy(np->name, proc->name, sizeof(proc->name));

    pid = np->pid;

    acquire(&ptable.lock);
    np->state = RUNNABLE;
    release(&ptable.lock);

    return pid;
}

```

- Nakon fork dijete izvršava isti program kao i roditelj.
- U adresni prostor procesa moguće je učitati neki program iz fajl sistema pomoću sistemskog poziva `exec`
 - `exec(char *path, char **argv)`
 - `path` – staza u fajl sistemu gdje se nalazi izvršni fajl
 - `argv` – niz parametara za funkciju `main` od pozvanog programa
- Nakon izvršenja sistemskog poziva `exec`, adresni prostor trenutnog procesa je u potpunosti izmjenjen i počinje se izvršavati prva instrukcija u funkciji `main` upravo učitano programa.

kod tzv init procesa učitano u *userinit*

```
#include "syscall.h"          initcode.S
#include "traps.h"

# exec(init, argv)
.globl start
start:
    pushl $argv
    pushl $init
    pushl $0
    movl $SYS_exec, %eax
    int $T_SYSCALL

# for(;;) exit();
exit:
    movl $SYS_exit, %eax
    int $T_SYSCALL
    jmp exit

# char init[] = "/init\0";
init:
    .string "/init\0"

# char *argv[] = { init, 0 };
.p2align 2
argv:
    .long init
    .long 0
```

```
char *argv[] = { "sh", 0 };          init.c

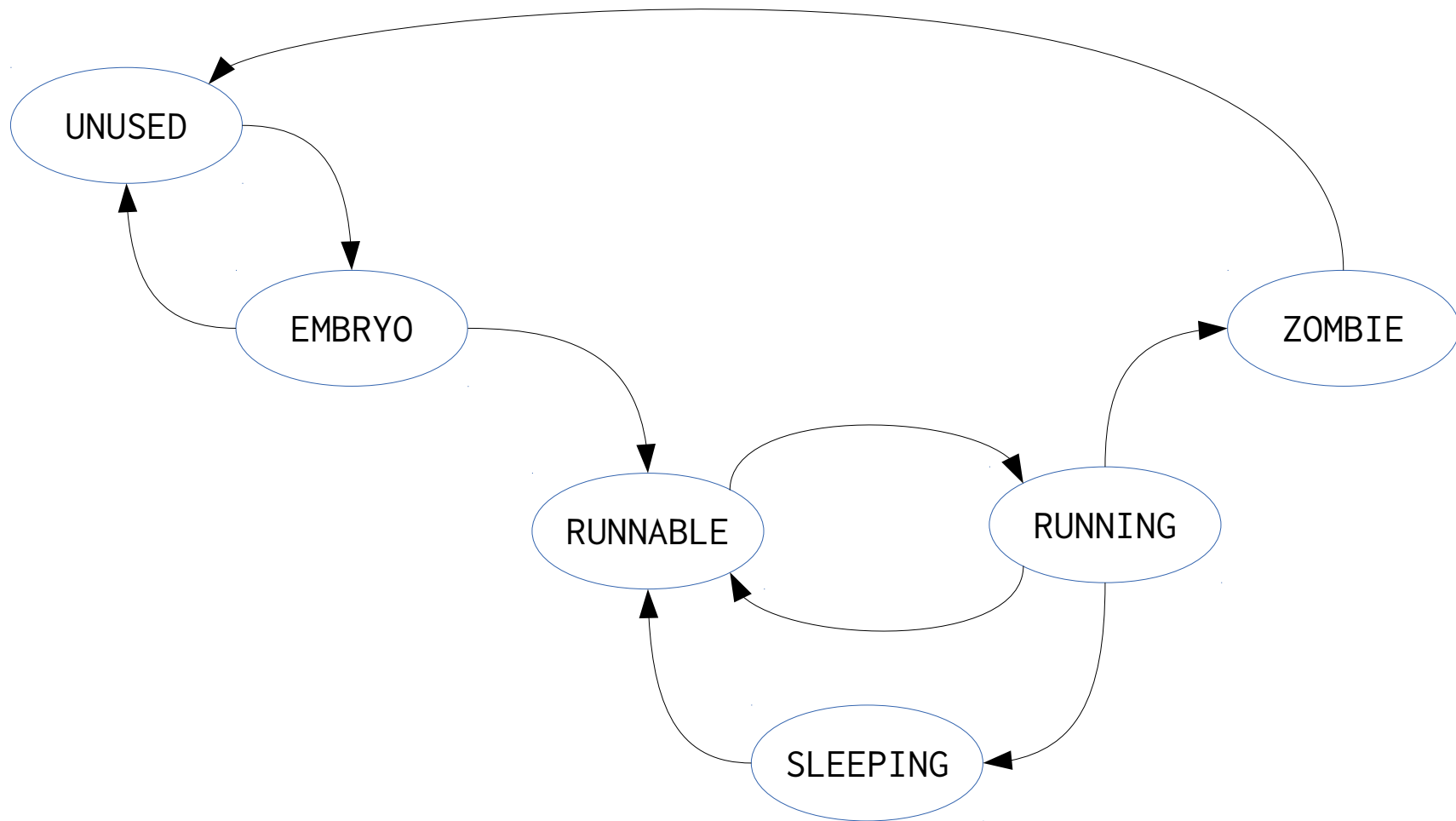
int main(void)
{
    int pid, wpid;

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

    for(;;){
        printf(1, "init: starting sh\n");
        pid = fork();
        if(pid < 0){
            printf(1, "init: fork failed\n");
            exit();
        }
        if(pid == 0){
            exec("sh", argv);
            printf(1, "init: exec sh failed\n");
            exit();
        }
        while((wpid=wait()) >= 0 && wpid != pid)
            printf(1, "zombie!\n");
    }
}
```

Dijagram promjene stanja procesa

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



sleep i wakeup

- Često se pojavljuju situacije u kojim nastavak izvršenja koda nekog procesa nema smisla dok se ne ispuni neki uslov npr:
 - procesiranje informacije iz paketa koji treba biti primljen sa mreže;
 - izvršenje nekog algoritma nakon N sekundi;
 - izvršenje nekog algoritma nakon što drugi proces okonča svoje izvršenje, itd...
- Kernel funkcija `sleep` “uspavljuje” trenutni proces na nekom kanalu (obično adresa neke kernel strukture)
- Kernel funkcija `wakeup` “budi” sve procese uspavane na nekom kanalu.

```
void sleep(void *chan, struct spinlock *lk) {
    if(proc == 0)
        panic("sleep");

    if(lk == 0)
        panic("sleep without lk");

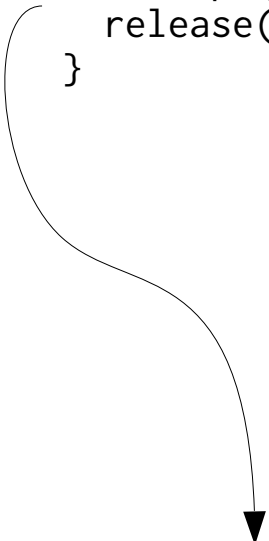
    if(lk != &ptable.lock){
        acquire(&ptable.lock);
        release(lk);
    }

    // Go to sleep.
    proc->chan = chan;
    proc->state = SLEEPING;
    sched();

    // Tidy up.
    proc->chan = 0;

    // Reacquire original lock.
    if(lk != &ptable.lock){
        release(&ptable.lock);
        acquire(lk);
    }
}
```

```
void wakeup(void *chan) {
    acquire(&ptable.lock);
    wakeup1(chan);
    release(&ptable.lock);
}
```



```
static void wakeup1(void *chan) {
    struct proc *p;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        if(p->state == SLEEPING && p->chan == chan)
            p->state = RUNNABLE;
}
```

- npr. sistemski poziv `sys_sleep` uspavljuje proces za vrijeme od `n` prekida periodičnog tajmera i to na kanalu `&ticks`
- prilikom tretmana tajmer prekida, a nakon inkrementiranja globalne varijable `ticks`, pozivom `wakup` bude se svi uspavani procesi na kanalu `&ticks`

```
int sys_sleep(void) {
    int n;
    uint ticks0;

    if(argint(0, &n) < 0)
        return -1;

    acquire(&tickslock);
    ticks0 = ticks;
    while(ticks - ticks0 < n){
        if(proc->killed){
            release(&tickslock);
            return -1;
        }
        sleep(&ticks, &tickslock);
    }
    release(&tickslock);

    return 0;
}
```

Sistemiški poziv `exit`

- Da bi se ispravno terminirao, proces treba da pozove sistemski poziv `exit`
 - sa wakeup budi eventualno uspavanog roditelja;
 - ukoliko trenutni proces ima djecu, njihov roditelj postaje `initproc` (pointer na prvi proces tj `init`);
 - stanje trenutnog procesa postaje `ZOMBIE`

```

void exit(void) {
    struct proc *p;
    int fd;

    if(proc == initproc)
        panic("init exiting");

    ...

    acquire(&ptable.lock);

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

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

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

```

Sistemiški poziv wait

- Ukoliko nakon fork, roditelj želi da sačeka kraj izvršenja koda djeteta, tada treba da pozove sistemski poziv wait:
 - wait pronalazi prvi proces koji je u stanju ZOMBIE i čiji je roditelj proces koji je pozvao wait.
 - dealocira resurse od pronađenog procesa;
 - postavlja pronađeni proces u UNUSED stanje
 - wait vraća PID pronađenog procesa ili -1 ako ne pronađe niti jedno dijete.
 - ako niti jedno dijete nije u ZOMBIE stanju, wait poziva sleep na kanalu proc.

```

int wait(void) {
    struct proc *p;
    int havekids, pid;
    acquire(&ptable.lock);
    for(;;){
        // Scan through table looking for zombie children.
        havekids = 0;
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->parent != proc)
                continue;
            havekids = 1;
            if(p->state == ZOMBIE){
                // Found one.
                pid = p->pid;
                kfree(p->kstack);
                p->kstack = 0;
                freevm(p->pgdir);
                p->state = UNUSED;
                p->pid = 0;
                p->parent = 0;
                p->name[0] = 0;
                p->killed = 0;
                release(&ptable.lock);
                return pid;
            }
        }
        // No point waiting if we don't have any children.
        if(!havekids || proc->killed){
            release(&ptable.lock);
            return -1;
        }
        // Wait for children to exit. (See wakeup1 call in proc_exit.)
        sleep(proc, &ptable.lock); //DOC: wait-sleep
    }
}

```

Sistemiški poziv kill

- Bilo koji proces može prijevremeno terminirati neki drugi proces sistemskim pozivom kill, i to sa argumentom PID od procesa koji se terminira.

```
int kill(int pid) {
    struct proc *p;

    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->pid == pid){
            p->killed = 1;
            // Wake process from sleep if necessary.
            if(p->state == SLEEPING)
                p->state = RUNNABLE;
            release(&ptable.lock);
            return 0;
        }
    }
    release(&ptable.lock);
    return -1;
}
```