# Архитектура операционной системы

**Process scheduling**

# Что такое процесс?

# Что такое процесс?

- Иллюзия персональной абстрактной машины
  - Адресное пространство
    - Страницы, таблицы
  - CPU
  - Файлы, и другие абстракции ОС
  - Состояние?
  - Стек?

- proc.h:52,  sched.h

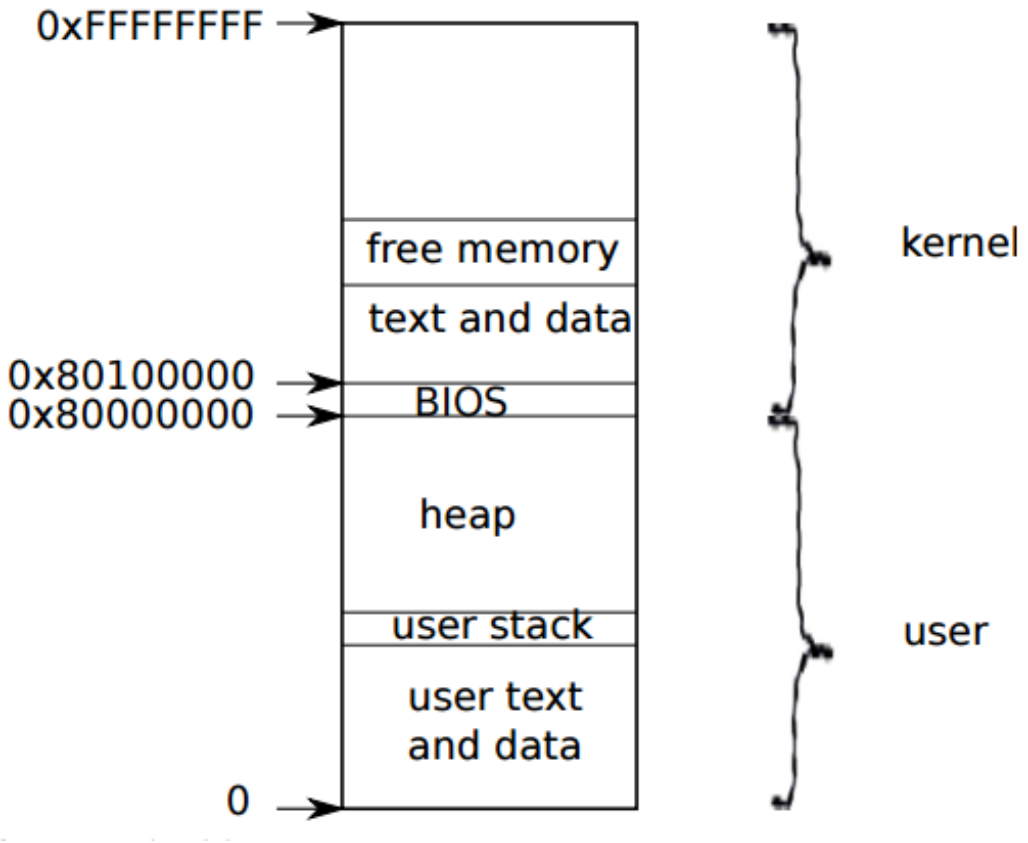
# Process context (xv6)

```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

```
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)
};
```

# Память процесса (layout)



# Что такое поток?

```
1  [||||||||                                          11.6%]   Tasks: 192, 906 thr; 1 running
2  [||||||||                                          10.5%]   Load average: 0.87 1.23 1.51
3  [||||||||||||||||||||||||                           37.4%]   Uptime: 02:04:32
4  [||||||||||                                        14.7%]
Mem[||||||||||||||||||||||||||||||||||||||||||||||  6.12G/19.5G]
Swp[                                                  0K/19.9G]

  PID  PPID  TGID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
    1     0     1 root       20   0  181M  6568  3984 S  0.0  0.0  0:02.16 /sbin/init splash
 5909     1  5909 root       20   0 11688  1484   884 S  0.0  0.0  0:00.81 ├─ /sbin/mount.exfat /dev/mmcblk0p1 /media/kkv/9C33-6BBD -o r
 4298     1  4298 uuidd      20   0 29380  1376  1184 S  0.0  0.0  0:00.00 ├─ /usr/sbin/uuidd --socket-activation
 3837     1  3837 root       20   0  340M 10292  8504 S  0.0  0.1  0:01.73 ├─ /usr/lib/upower/upowerd
 3846     1  3837 root       20   0  340M 10292  8504 S  0.0  0.1  0:00.33 │  ├─ /usr/lib/upower/upowerd
 3845     1  3837 root       20   0  340M 10292  8504 S  0.0  0.1  0:00.00 │  └─ /usr/lib/upower/upowerd
 3771     1  3771 root       20   0  358M  9252  6508 S  0.0  0.0  0:01.48 ├─ /usr/lib/udisks2/udisksd --no-debug
 3786     1  3771 root       20   0  358M  9252  6508 S  0.0  0.0  0:00.00 │  ├─ /usr/lib/udisks2/udisksd --no-debug
 3784     1  3771 root       20   0  358M  9252  6508 S  0.0  0.0  0:00.00 │  ├─ /usr/lib/udisks2/udisksd --no-debug
 3781     1  3771 root       20   0  358M  9252  6508 S  0.0  0.0  0:00.00 │  ├─ /usr/lib/udisks2/udisksd --no-debug
 3779     1  3771 root       20   0  358M  9252  6508 S  0.0  0.0  0:00.00 │  └─ /usr/lib/udisks2/udisksd --no-debug
 3303     1  3303 kkv        20   0  218M 24432  6512 S  0.0  0.1  0:33.01 ├─ /usr/bin/gnome-keyring-daemon --daemonize --login
 3437     1  3303 kkv        20   0  218M 24432  6512 S  0.0  0.1  0:03.50 │  ├─ /usr/bin/gnome-keyring-daemon --daemonize --login
 3413     1  3303 kkv        20   0  218M 24432  6512 S  0.0  0.1  0:00.00 │  ├─ /usr/bin/gnome-keyring-daemon --daemonize --login
 3304     1  3303 kkv        20   0  218M 24432  6512 S  0.0  0.1  0:00.00 │  └─ /usr/bin/gnome-keyring-daemon --daemonize --login
 3288     1  3288 kkv        20   0 45412  5092  4156 S  0.0  0.0  0:00.03 ├─ /lib/systemd/systemd --user
 3295  3288  3295 kkv        20   0 82348  2820     0 S  0.0  0.0  0:00.00 │  └─ (sd-pam)
 3223     1  3223 libvirt-d  20   0 49980  2568  2184 S  0.0  0.0  0:00.00 ├─ /usr/sbin/dnsmasq --conf-file=/var/lib/libvirt/dnsmasq/def
 3224  3223  3224 root       20   0 49952   384    12 S  0.0  0.0  0:00.22 │  └─ /usr/sbin/dnsmasq --conf-file=/var/lib/libvirt/dnsmasq/
 2729     1  2729 root       20   0 36588   464     0 S  0.0  0.0  0:00.00 ├─ /usr/sbin/xrdp-sesman
 2712     1  2712 xrdp       20   0 24056   456     0 S  0.0  0.0  0:00.00 ├─ /usr/sbin/xrdp
 2708     1  2708 root       20   0  329M 12964 10860 S  0.0  0.1  0:00.16 ├─ /opt/cisco/anyconnect/bin/vpnagentd
 2772     1  2708 root       20   0  329M 12964 10860 S  0.0  0.1  0:00.00 │  ├─ /opt/cisco/anyconnect/bin/vpnagentd
F1Help  F2Setup F3SearchF4FilterF5SortedF6CollapF7Nice -F8Nice +F9Kill  F10Quit
```

# Fork/exec/clone

# Code analysis

- main.c
- bootasm.S
- proc.c
- vm.c
- main.c  +54

# OS Main

```
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();        // kernel page table
  mpinit();          // detect other processors
  lapicinit();       // interrupt controller
  seginit();         // segment descriptors
  picinit();         // disable pic
  ioapicinit();      // another interrupt controller
  consoleinit();     // console hardware
  uartinit();        // serial port
  pinit();           // process table
  tvinit();          // trap vectors
  binit();           // buffer cache
  fileinit();        // file table
  ideinit();         // disk
  startothers();     // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();        // first user process
  mpmain();          // finish this processor's setup
}
```

```c
121 userinit(void)
122 {
123   struct proc *p;
124   extern char _binary_initcode_start[], _binary_initcode_size[];
125
126   p = allocproc();
127
128   initproc = p;
129   if((p->pgdir = setupkvm()) == 0)
130     panic("userinit: out of memory?");
131   inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
132   p->sz = PGSIZE;
133   memset(p->tf, 0, sizeof(*p->tf));
134   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
135   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
136   p->tf->es = p->tf->ds;
137   p->tf->ss = p->tf->ds;
138   p->tf->eflags = FL_IF;
139   p->tf->esp = PGSIZE;
140   p->tf->eip = 0;  // beginning of initcode.S
141
142   safestrcpy(p->name, "initcode", sizeof(p->name));
143   p->cwd = namei("/");
144
145   // this assignment to p->state lets other cores
146   // run this process. the acquire forces the above
147   // writes to be visible, and the lock is also needed
148   // because the assignment might not be atomic.
149   acquire(&ptable.lock);
150
151   p->state = RUNNABLE;
152
153   release(&ptable.lock);
154 }
```

Создание контекста процесса

Создание памяти процесса

```c
181 // Load the initcode into address 0 of pgdir.
182 // sz must be less than a page.
183 void
184 inituvm(pde_t *pgdir, char *init, uint sz)
185 {
186   char *mem;
187
188   if(sz >= PGSIZE)
189     panic("inituvm: more than a page");
190   mem = kalloc();
191   memset(mem, 0, PGSIZE);
192   mappages(pgdir, 0, PGSIZE, V2P(mem), PTE_W|PTE_U);
193   memmove(mem, init, sz);
194 }
```

Запуск

```
10
11 // Interrupt descriptor table (shared by all CPUs).
12 struct gatedesc idt[256];
13 extern uint vectors[];  // in vectors.S: array of 256 entry pointers
14 struct spinlock tickslock;
15 uint ticks;
```

**trap.c**

```
29 void
30 idtinit(void)
31 {
32   lidt(idt, sizeof(idt));
33 }
```

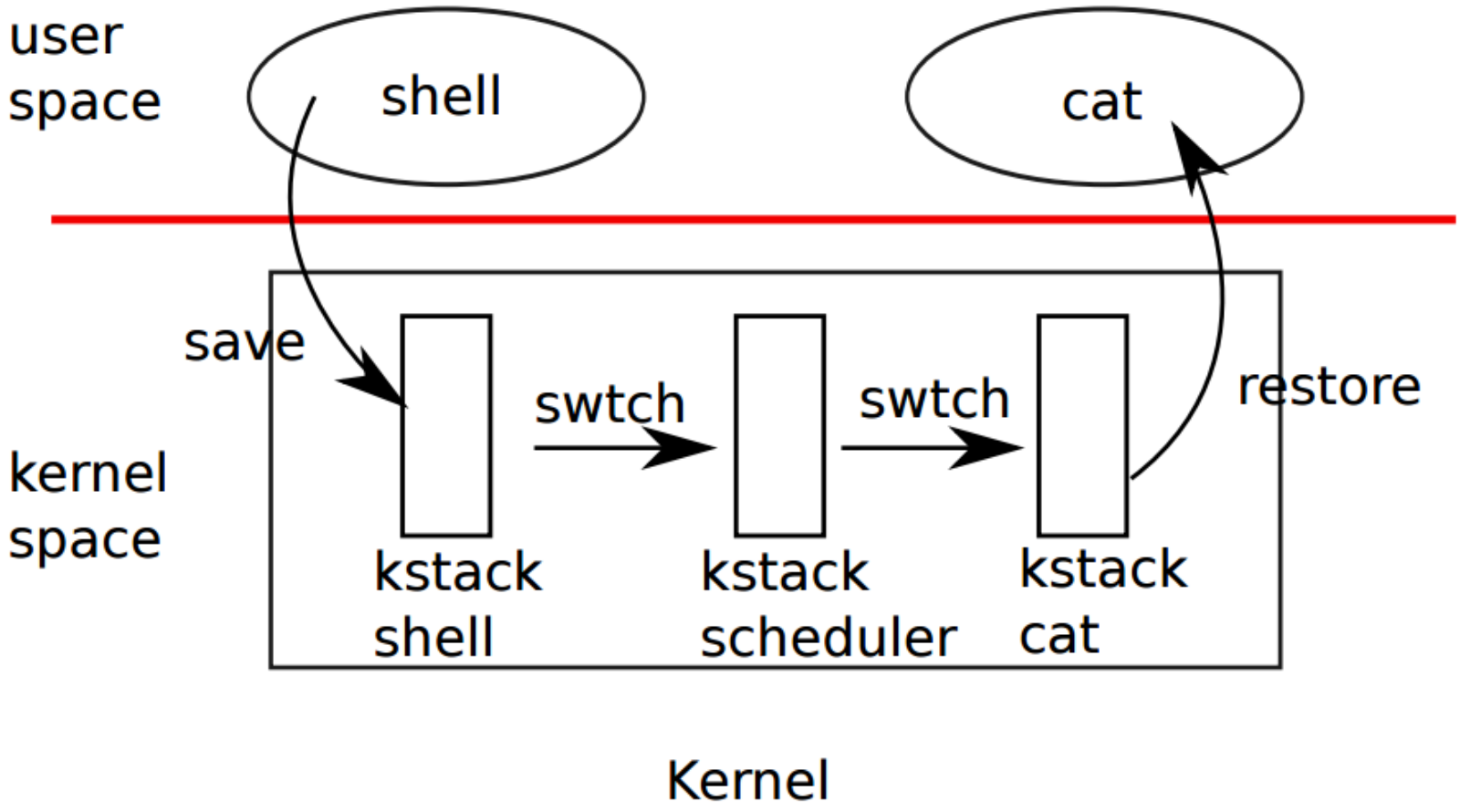
```
50 // Common CPU setup code.
51 static void
52 mpmain(void)
53 {
54   cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
55   idtinit();        // load idt register
56   xchg(&(mycpu()->started), 1); // tell startothers() we're up
57   scheduler();      // start running processes
58 }
```

```
280 // Per-CPU process scheduler.
281 // Each CPU calls scheduler() after setting itself up.
282 // Scheduler never returns.  It loops, doing:
283 //  - choose a process to run
284 //  - swtch to start running that process
285 //  - eventually that process transfers control
286 //      via swtch back to the scheduler.
287 void
288 scheduler(void)
289 {
290   struct proc *p;
291
292   for(;;){
293     // Enable interrupts on this processor.
294     sti();
295
296     // Loop over process table looking for process to run.
297     acquire(&ptable.lock);
298     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
299       if(p->state != RUNNABLE)
300         continue;
301
302       // Switch to chosen process.  It is the process's job
303       // to release ptable.lock and then reacquire it
304       // before jumping back to us.
305       proc = p;
306       switchuvm(p);
307       p->state = RUNNING;
308       swtch(&cpu->scheduler, p->context);
309       switchkvm();
310
311       // Process is done running for now.
312       // It should have changed its p->state before coming back.
313       proc = 0;
314     }
315     release(&ptable.lock);
316
317   }
318 }
```

# Много[задачность]

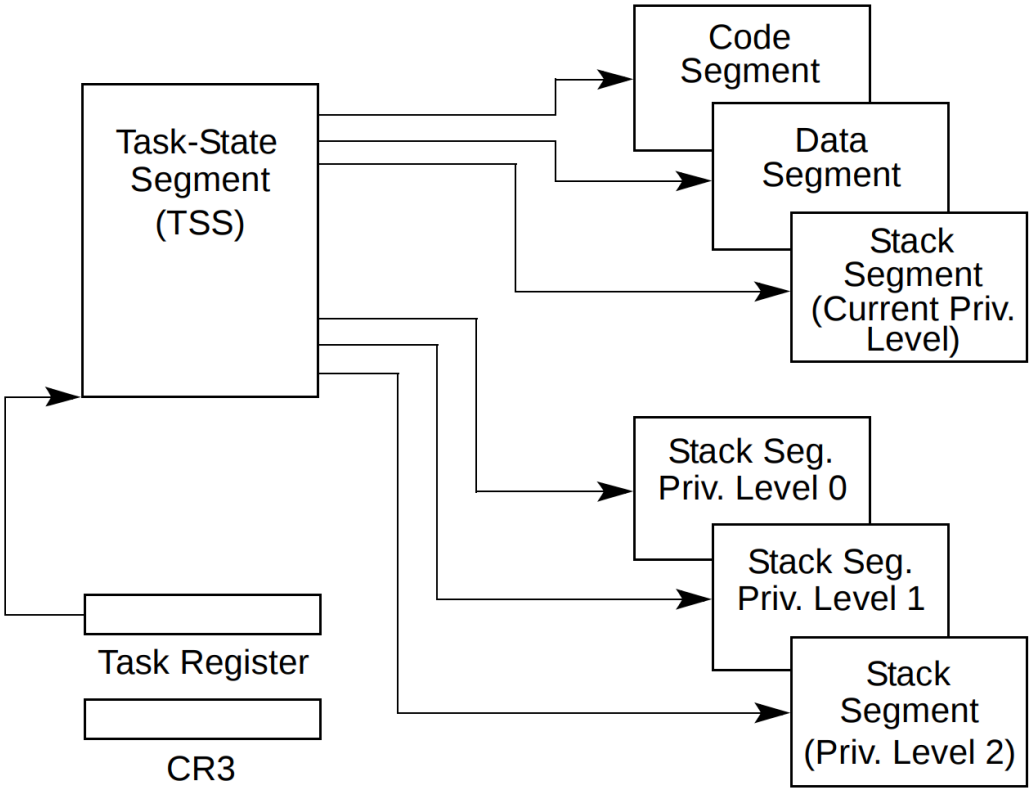
- Прерывание таймера
- Смена контекста
- План блокировок, при наличии нескольких CPU
- Освобождение ресурсов при завершении процесса

# Переключение контекста



# switchkvm/switchuvm

```c
// Switch h/w page table register to the kernel-only page table,
// for when no process is running.
void
switchkvm(void)
{
  lcr3(V2P(kpgdir));   // switch to the kernel page table
}
```




```c
// Switch TSS and h/w page table to correspond to process p.
void
switchuvm(struct proc *p)
{
  if(p == 0)
    panic("switchuvm: no process");
  if(p->kstack == 0)
    panic("switchuvm: no kstack");
  if(p->pgdir == 0)
    panic("switchuvm: no pgdir");

  pushcli();
  mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
                                sizeof(mycpu()->ts)-1, 0);
  mycpu()->gdt[SEG_TSS].s = 0;
  mycpu()->ts.ss0 = SEG_KDATA << 3;
  mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
  // setting IOPL=0 in eflags *and* iomb beyond the tss segment limit
  // forbids I/O instructions (e.g., inb and outb) from user space
  mycpu()->ts.iomb = (ushort) 0xFFFF;
  ltr(SEG_TSS << 3);
  lcr3(V2P(p->pgdir));  // switch to process's address space
  popcli();
}
```

# CPU state

```
// Per-CPU state
struct cpu {
  uchar id;                    // Local APIC ID; index into cpus[] below
  struct context *scheduler;   // swtch() here to enter scheduler
  struct taskstate ts;         // Used by x86 to find stack for interrupt
  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 before pushcli?

  // Cpu-local storage variables; see below
  struct cpu *cpu;
  struct proc *proc;           // The currently-running process.
};
```

# Переключение контекста (+1)

- Замечания:
  - Процесс:
    - Свой набор регистров

```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

    - Свой стек в ядре
- **Каждый CPU имеет свой scheduler thread**

```
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;
}
```

```
void
scheduler(void)
{
  struct proc *p;

  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.
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();

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

# swtch

```
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.

.globl swtch
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
```

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

## syscall.h

```
2 #define SYS_fork    1
3 #define SYS_exit    2
4 #define SYS_wait    3
5 #define SYS_pipe    4
6 #define SYS_read    5
7 #define SYS_kill    6
8 #define SYS_exec    7
9 #define SYS_fstat   8
10 #define SYS_chdir   9
11 #define SYS_dup     10
12 #define SYS_getpid  11
13 #define SYS_sbrk    12
14 #define SYS_sleep   13
15 #define SYS_uptime  14
16 #define SYS_open    15
17 #define SYS_write   16
18 #define SYS_mknod   17
19 #define SYS_unlink  18
20 #define SYS_link    19
21 #define SYS_mkdir   20
22 #define SYS_close   21
```

## syscall.c

```
85 extern int sys_chdir(void);
86 extern int sys_close(void);
87 extern int sys_dup(void);
88 extern int sys_exec(void);
89 extern int sys_exit(void);
90 extern int sys_fork(void);
91 extern int sys_fstat(void);
92 extern int sys_getpid(void);
93 extern int sys_kill(void);
94 extern int sys_link(void);
95 extern int sys_mkdir(void);
96 extern int sys_mknod(void);
97 extern int sys_open(void);
98 extern int sys_pipe(void);
99 extern int sys_read(void);
100 extern int sys_sbrk(void);
101 extern int sys_sleep(void);
102 extern int sys_unlink(void);
103 extern int sys_wait(void);
104 extern int sys_write(void);
105 extern int sys_uptime(void);
106
107 static int (*syscalls[])(void) = {
108 [SYS_fork]    sys_fork,
109 [SYS_exit]    sys_exit,
110 [SYS_wait]    sys_wait,
111 [SYS_pipe]    sys_pipe,
112 [SYS_read]    sys_read,
113 [SYS_kill]    sys_kill,
114 [SYS_exec]    sys_exec,
115 [SYS_fstat]   sys_fstat,
116 [SYS_chdir]   sys_chdir,
117 [SYS_dup]     sys_dup,
118 [SYS_getpid]  sys_getpid,
119 [SYS_sbrk]    sys_sbrk,
120 [SYS_sleep]   sys_sleep,
121 [SYS_uptime]  sys_uptime,
122 [SYS_open]    sys_open,
123 [SYS_write]   sys_write,
124 [SYS_mknod]   sys_mknod,
125 [SYS_unlink]  sys_unlink,
126 [SYS_link]    sys_link,
127 [SYS_mkdir]   sys_mkdir,
128 [SYS_close]   sys_close,
129 };
```

## sysproc.c

```
10 int
11 sys_fork(void)
12 {
13   return fork();
14 }
```

```
177 // Create a new process copying p as the parent.
178 // Sets up stack to return as if from system call.
179 // Caller must set state of returned proc to RUNNABLE.
180 int
181 fork(void)
182 {
183   int i, pid;
184   struct proc *np;
185   struct proc *curproc = myproc();
186
187   // Allocate process.
188   if((np = allocproc()) == 0){
189     return -1;
190   }
191
192   // Copy process state from proc.
193   if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
194     kfree(np->kstack);
195     np->kstack = 0;
196     np->state = UNUSED;
197     return -1;
198   }
199   np->sz = curproc->sz;
200   np->parent = curproc;
201   *np->tf = *curproc->tf;
202
203   // Clear %eax so that fork returns 0 in the child.
204   np->tf->eax = 0;
205
206   for(i = 0; i < NOFILE; i++)
207     if(curproc->ofile[i])
208       np->ofile[i] = filedup(curproc->ofile[i]);
209   np->cwd = idup(curproc->cwd);
210
211   safestrcpy(np->name, curproc->name, sizeof(curproc->name));
212
213   pid = np->pid;
214
215   acquire(&ptable.lock);
216
217   np->state = RUNNABLE;
218
219   release(&ptable.lock);
220
221   return pid;
222 }
```

## usys.S

```
1 #include "syscall.h"
2 #include "traps.h"
3
4 #define SYSCALL(name) \
5   .globl name; \
6   name: \
7     movl $SYS_ ## name, %eax; \
8     int $T_SYSCALL; \
9     ret
10
11 SYSCALL(fork)
12 SYSCALL(exit)
13 SYSCALL(wait)
```