实例分析

9.30 第四部分

刘卓轩、杨程远、卜秋实、陆润宇

系统调用的流程

- 用户态程序init中调用了fork，wait等系统调用。
- 下面以fork为例，分析系统调用的流程:

```c
// init: The initial user-level program

#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

char *argv[] = { "sh", 0 };

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");
  }
}
```

usys.S

```asm
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
```

vectors.S

```
# generated by vectors.pl - do not edit
# handlers
.globl alltraps
.globl vector0
vector0:
  pushl $0
  pushl $0
  jmp alltraps
.globl vector1
vector1:
  pushl $0
  pushl $1
  jmp alltraps
.globl vector2
vector2:
  pushl $0
  pushl $2
  jmp alltraps
.globl vector3
vector3:
  pushl $0
  pushl $3
  jmp alltraps
.globl vector4
vector4:
  pushl $0
  pushl $4
  jmp alltraps
.globl vector5
vector5:
  pushl $0
  pushl $5
  jmp alltraps
```

trapasm.S

```
1  #include "mmu.h"
2
3    # vectors.S sends all traps here.
4  .globl alltraps
5  alltraps:
6    # Build trap frame.
7    pushl %ds
8    pushl %es
9    pushl %fs
10   pushl %gs
11   pushal
12
13   # Set up data segments.
14   movw $(SEG_KDATA<<3), %ax
15   movw %ax, %ds
16   movw %ax, %es
17
18   # Call trap(tf), where tf=%esp
19   pushl %esp
20   call trap
21   addl $4, %esp
22
23   # Return falls through to trapret...
24 .globl trapret
25 trapret:
26   popal
27   popl %gs
28   popl %fs
29   popl %es
30   popl %ds
31   addl $0x8, %esp  # trapno and errcode
32   iret
33
```

trap.c > ...

```
36  void
37  trap(struct trapframe *tf)
38  {
39    if(tf->trapno == T_SYSCALL){
40      if(myproc()->killed)
41        exit();
42      myproc()->tf = tf;
43      syscall();
44      if(myproc()->killed)
45        exit();
46      return;
47    }
48
49    switch(tf->trapno){
50  >   case T_IRQ0 + IRQ_TIMER: ···
59  >   case T_IRQ0 + IRQ_IDE: ···
63  >   case T_IRQ0 + IRQ_IDE+1: ···
66  >   case T_IRQ0 + IRQ_KBD: ···
70  >   case T_IRQ0 + IRQ_COM1: ···
74      case T_IRQ0 + 7:
75  >   case T_IRQ0 + IRQ_SPURIOUS: ···
81    //PAGEBREAK: 13
82    default:
```

syscall.c > ...

```
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  };
130
131  void
132  syscall(void)
133  {
134    int num;
135    struct proc *curproc = myproc();
136
137    num = curproc->tf->eax;
138    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
139      curproc->tf->eax = syscalls[num]();
140    } else {
141      cprintf("%d %s: unknown sys call %d\n",
142              curproc->pid, curproc->name, num);
143      curproc->tf->eax = -1;
```

sysproc.c > ...

```
1   #include "types.h"
2   #include "x86.h"
3   #include "defs.h"
4   #include "date.h"
5   #include "param.h"
6   #include "memlayout.h"
7   #include "mmu.h"
8   #include "proc.h"
9
10  int
11  sys_fork(void)
12  {
13    return fork();
14  }
15
16  int
17  sys_exit(void)
18  {
19    exit();
20    return 0;  // not reached
21  }
22
```

proc.c > ...

```
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
```

proc.c > ...

```
74   allocproc(void)
75   {
76     struct proc *p;
77     char *sp;
78
79     acquire(&ptable.lock);
80
81     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
82       if(p->state == UNUSED)
83         goto found;
84
85     release(&ptable.lock);
86     return 0;
87
88   found:
89     p->state = EMBRYO;
90     p->pid = nextpid++;
91
92     release(&ptable.lock);
93
94     // Allocate kernel stack.
95     if((p->kstack = kalloc()) == 0){
96       p->state = UNUSED;
97       return 0;
98     }
99     sp = p->kstack + KSTACKSIZE;
100
101    // Leave room for trap frame.
102    sp -= sizeof *p->tf;
103    p->tf = (struct trapframe*)sp;
104
105    // Set up new context to start executing at forkret,
106    // which returns to trapret.
107    sp -= 4;
108    *(uint*)sp = (uint)trapret;
109
110    sp -= sizeof *p->context;
111    p->context = (struct context*)sp;
112    memset(p->context, 0, sizeof *p->context);
113    p->context->eip = (uint)forkret;
114
115    return p;
116  }
```

- 返回流程：fork -> sys_fork ->syscall -> trap -> alltraps

proc.c > ...

```
181  fork(void)
182  {
213    pid = np->pid;
214
215    acquire(&ptable.lock);
216
217    np->state = RUNNABLE;
218
219    release(&ptable.lock);
220
221    return pid;
222  }
```

syscall.c > ...

```
131  void
132  syscall(void)
133  {
134    int num;
135    struct proc *curproc = myproc();
136
137    num = curproc->tf->eax;
138    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
139      curproc->tf->eax = syscalls[num]();
140    } else {
141      cprintf("%d %s: unknown sys call %d\n",
142              curproc->pid, curproc->name, num);
143      curproc->tf->eax = -1;
144    }
145  }
146
```

sysproc.c > ...

```
10   int
11   sys_fork(void)
12   {
13     return fork();
14   }
15
```

trap.c > ...

```
36   void
37   trap(struct trapframe *tf)
38   {
39     if(tf->trapno == T_SYSCALL){
40       if(myproc()->killed)
41         exit();
42       myproc()->tf = tf;
43       syscall();
44       if(myproc()->killed)
45         exit();
46       return;
47     }
```

- iret后回到用户态usys中，再回到init里

trapasm.S

```
#include "mmu.h"

  # vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # Set up data segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es

  # 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

```

usys.S

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

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
```

init.c > ...

```
  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");
  }
}
```

1.为什么init.c中的fork是usys.S中定义的fork系统调用，而不是proc.c中定义的fork函数？

用户态程序

```
168 UPROGS=\
169     _cat\
170     _echo\
171     _forktest\
172     _grep\
173     _init\
174     _kill\
175     _ln\
176     _ls\
177     _mkdir\
178     _rm\
179     _sh\
180     _stressfs\
181     _usertests\
182     _wc\
183     _zombie\
184
```

```
148 _%: %.o $(ULIB)
149     $(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $@ $^
150     $(OBJDUMP) -S $@ > $*.asm
151     $(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d; s/
152
```

```
146 ULIB = ulib.o usys.o printf.o umalloc.o
147
```

```
185 fs.img: mkfs README $(UPROGS)
186     ./mkfs fs.img README $(UPROGS)
187
```

文件系统镜像

对象文件链接成kernel

Makefile

```
1   OBJS = \
2       bio.o\
3       console.o\
4       exec.o\
5       file.o\
6       fs.o\
7       ide.o\
8       ioapic.o\
9       kalloc.o\
10      kbd.o\
11      lapic.o\
12      log.o\
13      main.o\
14      mp.o\
15      picirq.o\
16      pipe.o\
17      proc.o\
18      sleeplock.o\
19      spinlock.o\
20      string.o\
21      swtch.o\
22      syscall.o\
23      sysfile.o\
24      sysproc.o\
25      trapasm.o\
26      trap.o\
27      uart.o\
28      vectors.o\
29      vm.o\
```

```
123 kernel: $(OBJS) entry.o entryother initcode kernel.ld
124     $(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary ir
125     $(OBJDUMP) -S kernel > kernel.asm
126     $(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' >
127
```

```
93  xv6.img: bootblock kernel
94      dd if=/dev/zero of=xv6.img count=10000
95      dd if=bootblock of=xv6.img conv=notrunc
96      dd if=kernel of=xv6.img seek=1 conv=notrunc
97
```

1.为什么init.c中的fork是usys.S中定义的fork系统调用，而不是proc.c中定义的fork函数？

总结：

init.c是用户态程序，单独编译，与用户库ULIB（包括usys）链接，所以它调用的fork是usys.S中定义的。

内核程序proc.c，也是单独编译的，且和其他内核程序一同链接。

虽然proc.c和usys.S中都定义了fork，但它们从编译、链接甚至到制作镜像都是毫无交叉，所以init.c调用的fork当然是usys.S中定义的用户态系统调用函数fork。

3. 子进程是如何从RUNNABLE转换到RUNNING状态的?

init.c

```
32          exit();
33        }
34        while((wpid=wait()) >= 0 && wpid != pid)
35          printf(1, "zombie!\n");
36      }
37    }
38
```

proc.c

```
308
309     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
310     sleep(curproc, &ptable.lock);  //DOC: wait-sleep
311   }
312 }
313
```

proc.c

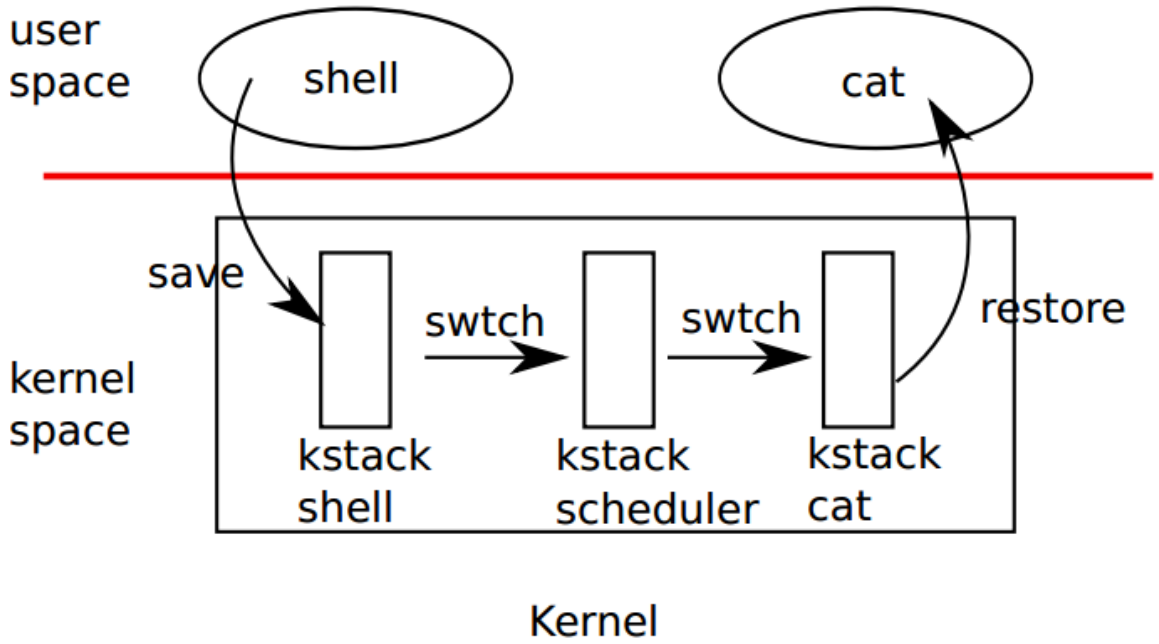
```
431   // guaranteed that we won't miss any wakeup
432   // (wakeup runs with ptable.lock locked),
433   // so it's okay to release lk.
434   if(lk != &ptable.lock){  //DOC: sleeplock0
435     acquire(&ptable.lock);  //DOC: sleeplock1
436     release(lk);
437   }
438   // Go to sleep.
439   p->chan = chan;
440   p->state = SLEEPING; //
441
442   sched();
```

proc.c > ...

```
379     intena = mycpu()->intena;
380     swtch(&p->context, mycpu()->scheduler);
381     mycpu()->intena = intena;
382   }
383
```

proc.c > ...

```
322   void
323   scheduler(void)
324   {
325     struct proc *p;
326     struct cpu *c = mycpu();
327     c->proc = 0;
328
329     for(;;){
330       // Enable interrupts on this processor.
331       sti();
332
333       // Loop over process table looking for process to run.
334       acquire(&ptable.lock);
335       for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
336         if(p->state != RUNNABLE)
337           continue;
338
339         // Switch to chosen process.  It is the process's job
340         // to release ptable.lock and then reacquire it
341         // before jumping back to us.
342         c->proc = p;
343         switchuvm(p);
344         p->state = RUNNING;
345
346         swtch(&(c->scheduler), p->context);
```



swtch.S

```asm
# 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

```

reference: xv6-book-rev11

4. main/init.c调用fork后，是父进程先返回还是子进程先返回？

mycpu()，myproc()

```
219  ifndef CPUS
220  CPUS := 1
221  endif
222  QEMUOPTS = -drive file=fs.img,index=1,media=disk,format=raw -drive file=
223
224  qemu: fs.img xv6.img
225      $(QEMU) -serial mon:stdio $(QEMUOPTS)
```

```c
// init: The initial user-level program

#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

char *argv[] = { "sh", 0 };

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");
  }
}
```

非抢占下，父进程先返回

5. 对于父进程和子进程，fork返回的pid相同吗？

```
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){…
190    }
191
192    // Copy process state from proc.
193  >   if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){…
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++)…
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  }
```

```
17   int nextpid = 1;
18   extern void forkret(void);
19   extern void trapret(void);
```

```
68   //PAGEBREAK: 32
69   // Look in the process table for an UNUSED proc.
70   // If found, change state to EMBRYO and initialize
71   // state required to run in the kernel.
72   // Otherwise return 0.
73   static struct proc*
74   allocproc(void)
75   {
76     struct proc *p;
77     char *sp;
78
79     acquire(&ptable.lock);
80
81   >   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)…
85     release(&ptable.lock);
86     return 0;
87
88   found:
89     p->state = EMBRYO;
90     p->pid = nextpid++;
91
```

子进程的中断栈的eax被设置为0

```
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){…
190    }
191
192    // Copy process state from proc.
193  >  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){…
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++)…
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  }
```

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

```
396  void
397  forkret(void)
398  {
399    static int first = 1;
400    // Still holding ptable.lock from scheduler.
401    release(&ptable.lock);
402
403    if (first) {
404      // Some initialization functions must be run in the context
405      // of a regular process (e.g., they call sleep), and thus cannot
406      // be run from main().
407      first = 0;
408      iinit(ROOTDEV);
409      initlog(ROOTDEV);
410    }
411
412    // Return to "caller", actually trapret (see allocproc).
413  }
```

6. 子进程返回后，加载的程序是什么程序?

init.c > ...

```
20    dup(0);  // stderr
21
22    for(;;){
23      printf(1, "init: starting sh\n");
24      pid = fork();
25      if(pid < 0){
26        printf(1, "init: fork failed\n");
27        exit();
28      }
29      if(pid == 0){
30        exec("sh", argv);
31        printf(1, "init: exec sh failed\n");
32        exit();
33      }
34      while((wpid=wait()) >= 0 && wpid != pid)
35        printf(1, "zombie!\n");
36    }
37  }
```

XV6和linux调度算法分析

C proc.c > ...

```
322  void
323  scheduler(void)
324  {
325    struct proc *p;
326    struct cpu *c = mycpu();
327    c->proc = 0;
328
329    for(;;){
330      // Enable interrupts on this processor.
331      sti();
332
333      // Loop over process table looking for process to run.
334      acquire(&ptable.lock);
335      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
336        if(p->state != RUNNABLE)
337          continue;
338
339        // Switch to chosen process.  It is the process's job
340        // to release ptable.lock and then reacquire it
341        // before jumping back to us.
342        c->proc = p;
343        switchuvm(p);
344        p->state = RUNNING;
345
346        swtch(&(c->scheduler), p->context);
```

朴素的查表法

```
static inline int idle_policy(int policy)
{
	return policy == SCHED_IDLE;
}

static inline int fair_policy(int policy)
{
	return policy == SCHED_NORMAL || policy == SCHED_BATCH;
}

static inline int rt_policy(int policy)
{
	return policy == SCHED_FIFO || policy == SCHED_RR;
}

static inline int dl_policy(int policy)
{
	return policy == SCHED_DEADLINE;
}
```

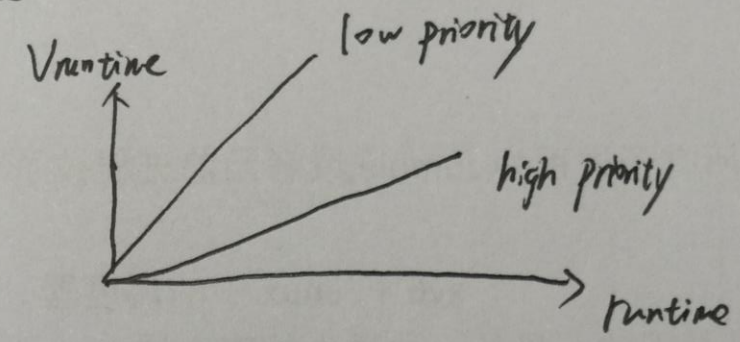
```
static inline bool valid_policy(int policy)
{
	return idle_policy(policy) || fair_policy(policy) ||
		rt_policy(policy) || dl_policy(policy);
}

static inline int task_has_rt_policy(struct task_struct *p)
{
	return rt_policy(p->policy);
}

static inline int task_has_dl_policy(struct task_struct *p)
{
	return dl_policy(p->policy);
}
```

这段代码中涉及到了linux 4.12.10中主要的几种调度策略，其中SCHED_NORMAL和SCHED_BATCH是针对普通任务（即没有对任务设置完成的deadline）的调度策略，SCHED_FIFO和SCHED_RR和SCHED_DEADLINE是针对实时进程的调度策略，SCHED_IDLE则负责在系统空闲时调用idle进程。

run_queue PCB → vruntime



CFS: choose process of the least vruntime

$$runtime_{(i)} = \frac{cycle \times w_i}{\sum w}$$

$$vruntime = \frac{runtime \times 1024}{w}$$

$$\Rightarrow$$

$$Vruntime = \frac{cycle \times w \times 1024}{(\sum w) \times w} = \frac{1024 \times cycle}{\sum w}$$

以下是fair_policy中进行平衡的一个片段（CFS完全公平调度算法的一个部分，摘自linux-4.12.10\kernel\sched\fair.c第7949行）

```
/*
 * Check this_cpu to ensure it is balanced within domain. Attempt to move
 * tasks if there is an imbalance.
 */
static int load_balance(int this_cpu, struct rq *this_rq,
                        struct sched_domain *sd, enum cpu_idle_type idle,
                        int *continue_balancing)
{
        int ld_moved, cur_ld_moved, active_balance = 0;
        struct sched_domain *sd_parent = sd->parent;
        struct sched_group *group;
        struct rq *busiest;
        struct rq_flags rf;
        struct cpumask *cpus = this_cpu_cpumask_var_ptr(load_balance_mask);

        struct lb_env env = {
                .sd             = sd,
                .dst_cpu        = this_cpu,
                .dst_rq         = this_rq,
                .dst_grpmask    = sched_group_cpus(sd->groups),
                .idle           = idle,
                .loop_break     = sched_nr_migrate_break,
                .cpus           = cpus,
                .fbq_type       = all,
                .tasks          = LIST_HEAD_INIT(env.tasks),
        };

        cpumask_and(cpus, sched_domain_span(sd), cpu_active_mask);

        schedstat_inc(sd->lb_count[idle]);

redo:
        if (!should_we_balance(&env)) {
                *continue_balancing = 0;
                goto out_balanced;
        }

        group = find_busiest_group(&env);
        if (!group) {
                schedstat_inc(sd->lb_nobusyg[idle]);
                goto out_balanced;
        }

        busiest = find_busiest_queue(&env, group);
        if (!busiest) {
                schedstat_inc(sd->lb_nobusyq[idle]);
                goto out_balanced;
        }

        BUG_ON(busiest == env.dst_rq);

        schedstat_add(sd->lb_imbalance[idle], env.imbalance);

        env.src_cpu = busiest->cpu;
        env.src_rq = busiest;

        ld_moved = 0;
        if (busiest->nr_running > 1) {
                /*
                 * Attempt to move tasks. If find_busiest_group has found
                 * an imbalance but busiest->nr_running <= 1, the group is
                 * still unbalanced. ld_moved simply stays zero, so it is
                 * correctly treated as an imbalance.
                 */
                env.flags |= LBF_ALL_PINNED;
                env.loop_max  = min(sysctl_sched_nr_migrate, busiest->nr_running);
```

```
more_balance:
        rq_lock_irqsave(busiest, &rf);
        update_rq_clock(busiest);

        /*
         * cur_ld_moved - load moved in current iteration
         * ld_moved     - cumulative load moved across iterations
         */
        cur_ld_moved = detach_tasks(&env);

        /*
         * We've detached some tasks from busiest_rq. Every
         * task is masked "TASK_ON_RQ_MIGRATING", so we can safely
         * unlock busiest->lock, and we are able to be sure
         * that nobody can manipulate the tasks in parallel.
         * See task_rq_lock() family for the details.
         */

        rq_unlock(busiest, &rf);

        if (cur_ld_moved) {
                attach_tasks(&env);
                ld_moved += cur_ld_moved;
        }

        local_irq_restore(rf.flags);

        if (env.flags & LBF_NEED_BREAK) {
                env.flags &= ~LBF_NEED_BREAK;
                goto more_balance;
        }
```

```
        /*
         * Revisit (affine) tasks on src_cpu that couldn't be moved to
         * us and move them to an alternate dst_cpu in our sched_group
         * where they can run. The upper limit on how many times we
         * iterate on same src_cpu is dependent on number of cpus in our
         * sched_group.
         *
         * This changes load balance semantics a bit on who can move
         * load to a given_cpu. In addition to the given_cpu itself
         * (or a ilb_cpu acting on its behalf where given_cpu is
         * nohz-idle), we now have balance_cpu in a position to move
         * load to given_cpu. In rare situations, this may cause
         * conflicts (balance_cpu and given_cpu/ilb_cpu deciding
         * _independently_ and at _same_ time to move some load to
         * given_cpu) causing exceess load to be moved to given_cpu.
         * This however should not happen so much in practice and
         * moreover subsequent load balance cycles should correct the
         * excess load moved.
         */
        if ((env.flags & LBF_DST_PINNED) && env.imbalance > 0) {

                /* Prevent to re-select dst_cpu via env's cpus */
                cpumask_clear_cpu(env.dst_cpu, env.cpus);

                env.dst_rq       = cpu_rq(env.new_dst_cpu);
                env.dst_cpu      = env.new_dst_cpu;
                env.flags  &= ~LBF_DST_PINNED;
                env.loop   = 0;
                env.loop_break       = sched_nr_migrate_break;

                /*
                 * Go back to "more_balance" rather than "redo" since we
                 * need to continue with same src_cpu.
                 */
                goto more_balance;
        }
```

```
        /*
         * We failed to reach balance because of affinity.
         */
        if (sd_parent) {
                int *group_imbalance = &sd_parent->groups->sgc->imbalance;

                if ((env.flags & LBF_SOME_PINNED) && env.imbalance > 0)
                        *group_imbalance = 1;
        }

        /* All tasks on this runqueue were pinned by CPU affinity */
        if (unlikely(env.flags & LBF_ALL_PINNED)) {
                cpumask_clear_cpu(cpu_of(busiest), cpus);
                /*
                 * Attempting to continue load balancing at the current
                 * sched_domain level only makes sense if there are
                 * active CPUs remaining as possible busiest CPUs to
                 * pull load from which are not contained within the
                 * destination group that is receiving any migrated
                 * load.
                 */
                if (!cpumask_subset(cpus, env.dst_grpmask)) {
                        env.loop = 0;
                        env.loop_break = sched_nr_migrate_break;
                        goto redo;
                }
                goto out_all_pinned;
        }
}       //      对应于    if (busiest->nr_running > 1) {
```

```
if (!ld_moved) {
        schedstat_inc(sd->lb_failed[idle]);
        /*
         * Increment the failure counter only on periodic balance.
         * We do not want newidle balance, which can be very
         * frequent, pollute the failure counter causing
         * excessive cache_hot migrations and active balances.
         */
        if (idle != CPU_NEWLY_IDLE)
                sd->nr_balance_failed++;

        if (need_active_balance(&env)) {
                unsigned long flags;

                raw_spin_lock_irqsave(&busiest->lock, flags);

                /* don't kick the active_load_balance_cpu_stop,
                 * if the curr task on busiest cpu can't be
                 * moved to this_cpu
                 */
                if (!cpumask_test_cpu(this_cpu, &busiest->curr->cpus_allowed)) {
                        raw_spin_unlock_irqrestore(&busiest->lock,
                                                   flags);
                        env.flags |= LBF_ALL_PINNED;
                        goto out_one_pinned;
                }
```

```
                /*
                 * ->active_balance synchronizes accesses to
                 * ->active_balance_work.  Once set, it's cleared
                 * only after active load balance is finished.
                 */
                if (!busiest->active_balance) {
                        busiest->active_balance = 1;
                        busiest->push_cpu = this_cpu;
                        active_balance = 1;
                }
                raw_spin_unlock_irqrestore(&busiest->lock, flags);

                if (active_balance) {
                        stop_one_cpu_nowait(cpu_of(busiest),
                                active_load_balance_cpu_stop, busiest,
                                &busiest->active_balance_work);
                }

                /* We've kicked active balancing, force task migration. */
                sd->nr_balance_failed = sd->cache_nice_tries+1;
        }
} else
        sd->nr_balance_failed = 0;

if (likely(!active_balance)) {
        /* We were unbalanced, so reset the balancing interval */
        sd->balance_interval = sd->min_interval;
} else {
        /*
         * If we've begun active balancing, start to back off. This
         * case may not be covered by the all_pinned logic if there
         * is only 1 task on the busy runqueue (because we don't call
         * detach_tasks).
         */
        if (sd->balance_interval < sd->max_interval)
                sd->balance_interval *= 2;
}

goto out;
```

```
out_balanced:
        /*
         * We reach balance although we may have faced some affinity
         * constraints. Clear the imbalance flag if it was set.
         */
        if (sd_parent) {
                int *group_imbalance = &sd_parent->groups->sgc->imbalance;

                if (*group_imbalance)
                        *group_imbalance = 0;
        }

out_all_pinned:
        /*
         * We reach balance because all tasks are pinned at this level so
         * we can't migrate them. Let the imbalance flag set so parent level
         * can try to migrate them.
         */
        schedstat_inc(sd->lb_balanced[idle]);

        sd->nr_balance_failed = 0;

out_one_pinned:
        /* tune up the balancing interval */
        if (((env.flags & LBF_ALL_PINNED) &&
                        sd->balance_interval < MAX_PINNED_INTERVAL) ||
                        (sd->balance_interval < sd->max_interval))
                sd->balance_interval *= 2;

        ld_moved = 0;
out:
        return ld_moved;
}
```

2.为什么子进程和父进程一样都会返回main/init.c?

proc.c > ...

```
86    return 0;
87
88  found:
89    p->state = EMBRYO;
90    p->pid = nextpid++;
91
92    release(&ptable.lock);
93
94    // Allocate kernel stack.
95  > if((p->kstack = kalloc()) == 0){ ···
98    }
99    sp = p->kstack + KSTACKSIZE;
100
101   // Leave room for trap frame.
102   sp -= sizeof *p->tf;
103   p->tf = (struct trapframe*)sp;
104
105   // Set up new context to start executing at forkret,
106   // which returns to trapret.
107   sp -= 4;
108   *(uint*)sp = (uint)trapret;
109
110   sp -= sizeof *p->context;
111   p->context = (struct context*)sp;
112   memset(p->context, 0, sizeof *p->context);
113   p->context->eip = (uint)forkret;//eip is PC
114
115   return p;
116 }
```

allocproc()

swtch.S

```
1   # Context switch
2   #
3   #   void swtch(struct context **old, struct context *new);
4   #
5   # Save the current registers on the stack, creating
6   # a struct context, and save its address in *old.
7   # Switch stacks to new and pop previously-saved registers.
8
9   .globl swtch
10  swtch:
11    movl 4(%esp), %eax
12    movl 8(%esp), %edx
13
14    # Save old callee-saved registers
15    pushl %ebp
16    pushl %ebx
17    pushl %esi
18    pushl %edi
19
20    # Switch stacks
21    movl %esp, (%eax)
22    movl %edx, %esp
23
24    # Load new callee-saved registers
25    popl %edi
26    popl %esi
27    popl %ebx
28    popl %ebp
29    ret
30
```

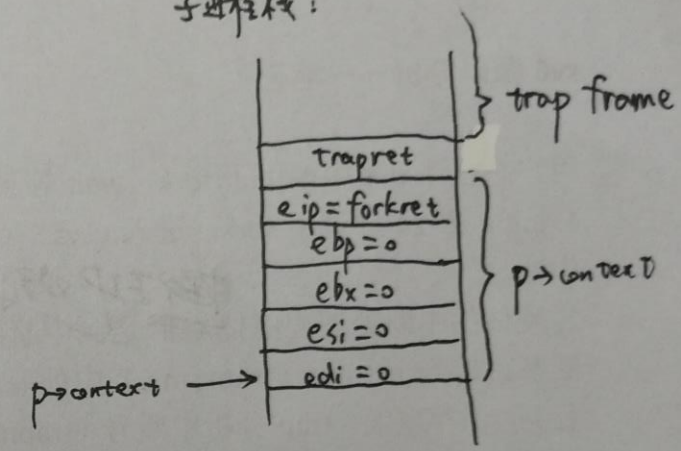
movl %esp, (%eax)把%esp赋值给P->context

如何进入子进程？

一、allocproc 造栈：

```
*(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;
```

子进程栈：



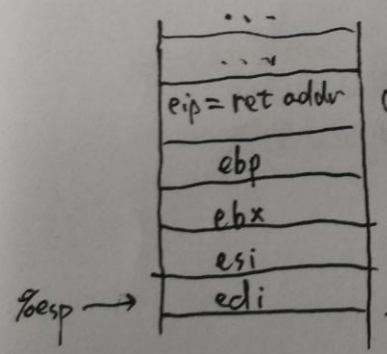
二、sched 切调度器：

4(%esp) → %eax　　8(%esp) → %edx

```
swtch(&p->context, mycpu()->scheduler);
```

旧进程栈：

注调用 swtch 时 call 指令将%eip压栈



③ movl %esp, (%eax)

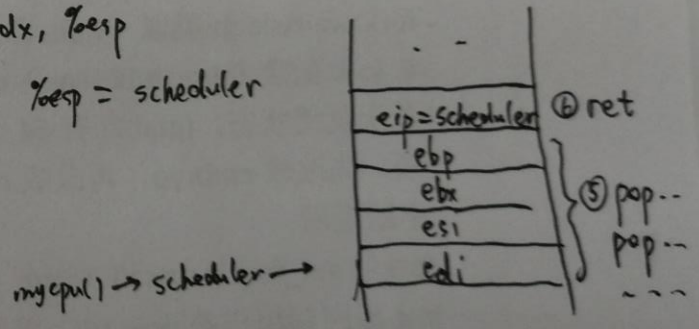
翻译成C语言：p->context = %esp;

②完成后的栈顶

④ movl %edx, %esp

翻译：%esp = scheduler

调度器栈：



第⑥步使得 eip 变为 scheduler此次调用 swtch 时的返回地址.

三. 调度器切新进程：

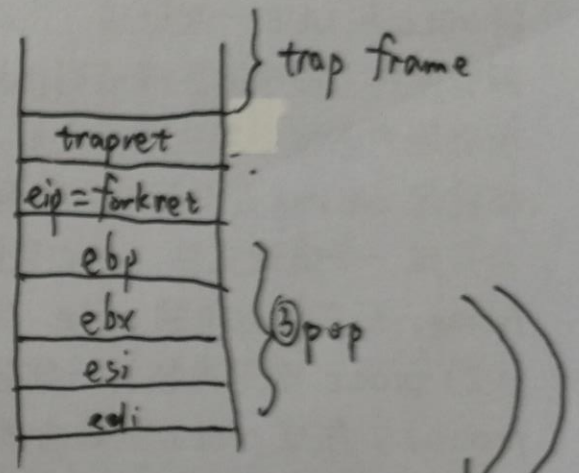
swtch(&(c→scheduler), p→context)

① 保存 调度器上下文，保存调度器栈顶
（同[二]中①～③）

② movl %edx, %esp

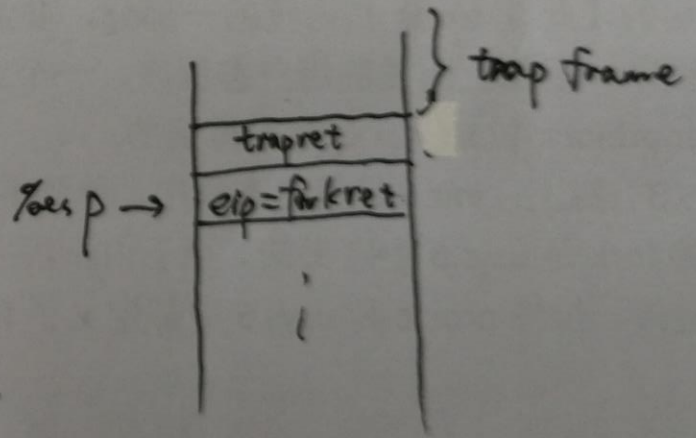
翻译：%esp = p→context

新进程（子进程栈）：



③ pop 四个寄存器

第③步完成后的栈：（新进程/子进程栈）



④ ret !!!

ret 指令将栈顶装入 %eip，所以执行完 ret 后，
%eip 被置为 forkret 的入口地址 !!!

四. forkret 继续返回，显然 ret 使得 %eip 被置为 trapret 的入口地址

```
192     // Copy process state from proc.
193 >   if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){ ···
198     }
199     np->sz = curproc->sz;
200     np->parent = curproc;
201     *np->tf = *curproc->tf;
202
```

```
// hardware and by trapasm.S; and passed to trap().
struct trapframe {
  // registers as pushed by pusha
  uint edi;
  uint esi;
  uint ebp;
  uint oesp;      // useless & ignored
  uint ebx;
  uint edx;
  uint ecx;
  uint eax;

  // rest of trap frame
  ushort gs;
  ushort padding1;
  ushort fs;
  ushort padding2;
  ushort es;
  ushort padding3;
  ushort ds;
  ushort padding4;
  uint trapno;

  // below here defined by x86 hardware
  uint err;
  uint eip;
  ushort cs;
  ushort padding5;
  uint eflags;

  // below here only when crossing rings, such as from user to
  uint esp;
  ushort ss;
  ushort padding6;
};
```

```
#include "mmu.h"

  # vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # Set up data segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es

  # 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

```

7. wait系统调用的功能？

proc.c > ...

```
269
270  // Wait for a child process to exit and return its pid.
271  // Return -1 if this process has no children.
272  int
273  wait(void)
274  {
275    struct proc *p;
276    int havekids, pid;
277    struct proc *curproc = myproc();
278
279    acquire(&ptable.lock);
280    for(;;){
281      // Scan through table looking for exited children.
282      havekids = 0;
283      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
284        if(p->parent != curproc)
285          continue;
286        havekids = 1;
287        if(p->state == ZOMBIE){
288          // Found one.
289          pid = p->pid;
290          kfree(p->kstack);
291          p->kstack = 0;
292          freevm(p->pgdir);
293          p->pid = 0;
294          p->parent = 0;
295          p->name[0] = 0;
296          p->killed = 0;
297          p->state = UNUSED;
298          release(&ptable.lock);
299          return pid;
300        }
301      }
```

proc.c > ...

```
272  int
273  wait(void)
274  {
275    struct proc *p;
276    int havekids, pid;
277    struct proc *curproc = myproc();
278
279    acquire(&ptable.lock);
280    for(;;){
281      // Scan through table looking for exited children.
282      havekids = 0;
283      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
284        if(p->parent != curproc)
285          continue;
286        havekids = 1;
287 >      if(p->state == ZOMBIE){ ···
300        }
301      }
302
303      // No point waiting if we don't have any children.
304      if(!havekids || curproc->killed){
305        release(&ptable.lock);
306        return -1;
307      }
308
309      // Wait for children to exit.  (See wakeup1 call in proc_exit.)
310      sleep(curproc, &ptable.lock);  //DOC: wait-sleep
311    }
312  }
```

sleep等待的chan

```
38  struct proc {
39    uint sz;                     // Size of process memory (bytes)
40    pde_t* pgdir;                // Page table
41    char *kstack;                // Bottom of kernel stack for this process
42    enum procstate state;        // Process state
43    int pid;                     // Process ID
44    struct proc *parent;         // Parent process
45    struct trapframe *tf;        // Trap frame for current syscall
46    struct context *context;     // swtch() here to run process
47    void *chan;                  // If non-zero, sleeping on chan
48    int killed;                  // If non-zero, have been killed
49    struct file *ofile[NOFILE];  // Open files
50    struct inode *cwd;           // Current directory
51    char name[16];               // Process name (debugging)
52  };
```

```
415  // Atomically release lock and sleep on chan.
416  // Reacquires lock when awakened.
417  void
418  sleep(void *chan, struct spinlock *lk)
419  {
420    struct proc *p = myproc();
421
422    if(p == 0)
423      panic("sleep");
424
425    if(lk == 0)
426      panic("sleep without lk");
427
428    // Must acquire ptable.lock in order to
429    // change p->state and then call sched.
430    // Once we hold ptable.lock, we can be
431    // guaranteed that we won't miss any wakeup
432    // (wakeup runs with ptable.lock locked),
433    // so it's okay to release lk.
434    if(lk != &ptable.lock){  //DOC: sleeplock0
435      acquire(&ptable.lock);  //DOC: sleeplock1
436      release(lk);
437    }
438    // Go to sleep.
439    p->chan = chan;
440    p->state = SLEEPING; //
```

```
226  // until its parent calls wait() to find out it exited.
227  void
228  exit(void)
229  {
230    struct proc *curproc = myproc();
231    struct proc *p;
232    int fd;
233
234    if(curproc == initproc)
235      panic("init exiting");
236
237    // Close all open files.
238    for(fd = 0; fd < NOFILE; fd++){
239      if(curproc->ofile[fd]){
240        fileclose(curproc->ofile[fd]);
241        curproc->ofile[fd] = 0;
242      }
243    }
244
245    begin_op();
246    iput(curproc->cwd);
247    end_op();
248    curproc->cwd = 0;
249
250    acquire(&ptable.lock);
251
252    // Parent might be sleeping in wait().
253    wakeup1(curproc->parent);
254
255    // Pass abandoned children to init.
256    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
257      if(p->parent == curproc){
258        p->parent = initproc;
259        if(p->state == ZOMBIE)
260          wakeup1(initproc);
261      }
262    }
```

```
  // The ptable lock must be held.
  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;
  }
```

Thank you for your listenning!