第五章

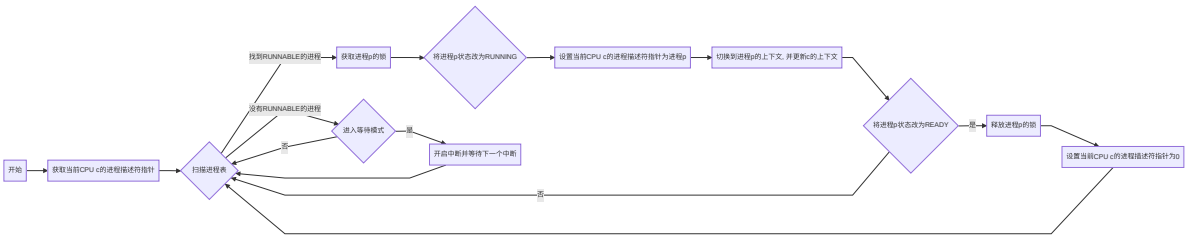
进程的数据结构

```
enum procstate {
    UNUSED,     // 未初始化
    USED,       // 基本初始化，未加载用户程序
    SLEEPING,   // 休眠状态(未使用，留待后续拓展)
    RUNNABLE,   // 可运行
    RUNNING,    // 当前正在运行
    ZOMBIE,     // 已经 exit；一个进程存在父进程且在父进程未结束时就结束，在等待父进程释放其资源
};

struct proc {
  struct spinlock lock;     // 进程锁

  // 使用这些变量需要持有 p->lock:
  enum procstate state;     // 进程状态
  struct proc *parent;      // 父进程
  void *chan;               // 如果不为零，则正在睡眠等待 chan
  int killed;               // 如果不为零，则已被杀死
  int xstate;               // 返回给父进程 wait 的退出状态码
  int pid;                  // 进程ID

  // 这些变量是私有的，因此不需要持有 p->lock。
  uint64 kstack;            // 内核栈的虚拟地址
  uint64 sz;                // 进程内存大小（字节）
  pagetable_t pagetable;    // 用户页表
  pagetable_t kpagetable;   // 内核页表
  struct trapframe *trapframe; // trampoline.S 的数据页
  struct context context;   // 在此处运行进程的 swtch()
  struct file *ofile[NOFILE];  // 打开的文件
  struct dirent *cwd;       // 当前目录
  char name[16];            // 进程名称（用于调试）
  int tmask;                // 跟踪掩码
};
```

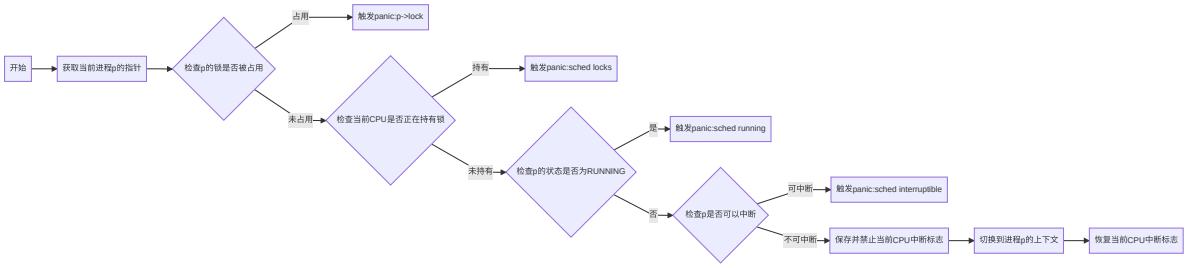
进程的基本管理

```
void scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  extern pagetable_t kernel_pagetable;

  c->proc = 0;
  for (;;) {
    // Avoid deadlock by ensuring that devices can interrupt.
    intr_on();

```

```
    int found = 0;
    for (p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if (p->state == RUNNABLE) {
    // Switch to chosen process.  It is the process's job
    // to release its lock and then reacquire it
    // before jumping back to us.
    // printf("[scheduler]found runnable proc with pid: %d\n", p->pid);
    p->state = RUNNING;
    c->proc = p;
    w_satp(MAKE_SATP(p->kpagetable));
    sfence_vma();
    swtch(&c->context, &p->context);
    w_satp(MAKE_SATP(kernel_pagetable));
    sfence_vma();
    // Process is done running for now.
    // It should have changed its p->state before coming back.
    c->proc = 0;

    found = 1;
      }
      release(&p->lock);
    }
    if (found == 0) {
      intr_on();
      asm volatile("wfi");
    }
  }
}
```

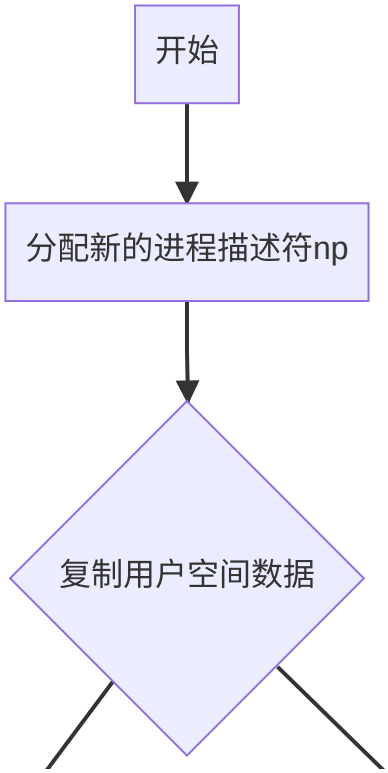


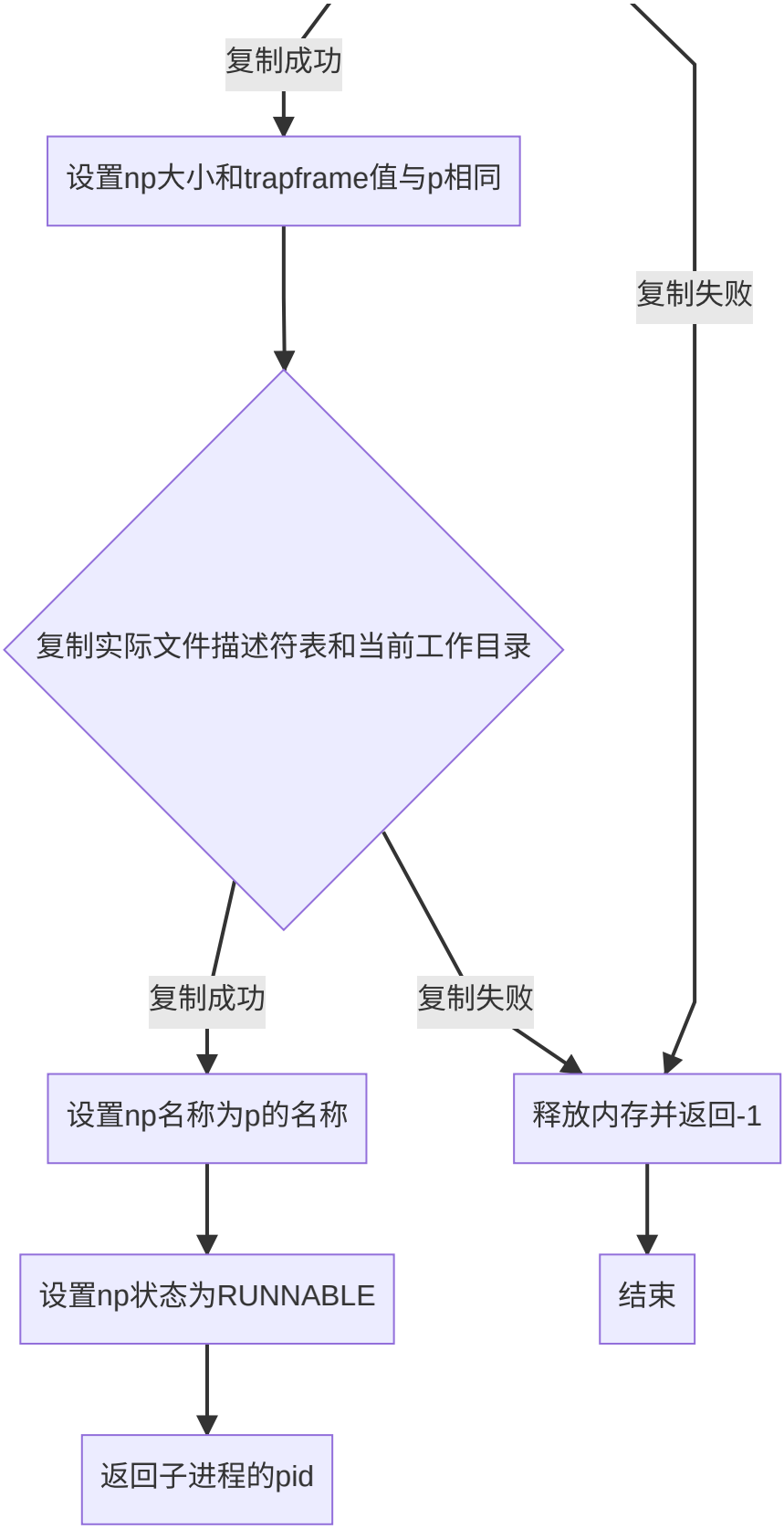

```
1  void
2  sched(void)
3  {
4    int intena;
5    struct proc *p = myproc();
6
7    if(!holding(&p->lock))
8      panic("sched p->lock");
9    if(mycpu()->noff != 1)
10     panic("sched locks");
11   if(p->state == RUNNING)
12     panic("sched running");
13   if(intr_get())
14     panic("sched interruptible");
15
16   intena = mycpu()->intena;
17   swtch(&p->context, &mycpu()->context);
18   mycpu()->intena = intena;
19 }
20
21
```



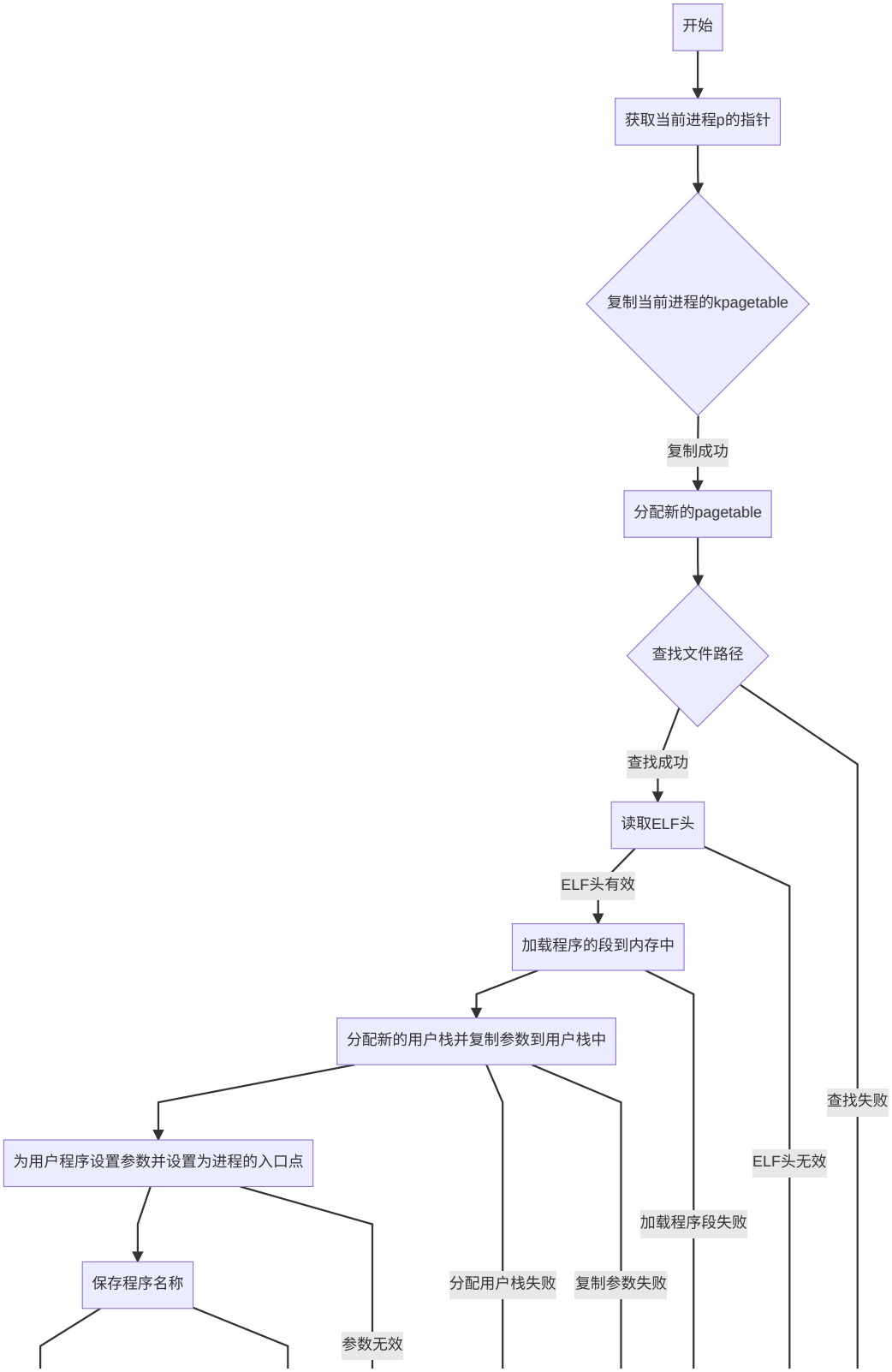
进程的重要系统调用

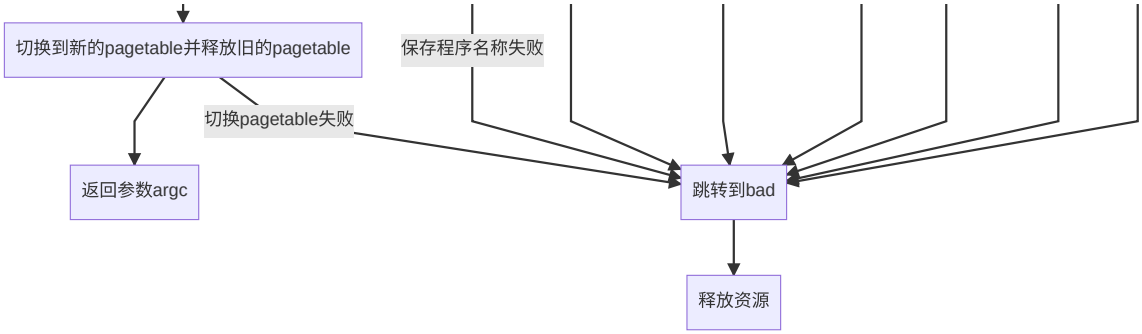
fork系统调用



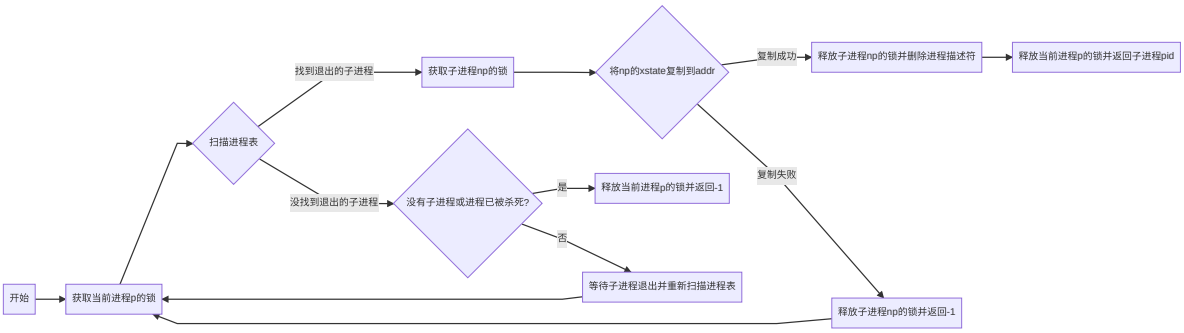

复制成功
设置np大小和trapframe值与p相同
复制失败
复制实际文件描述符表和当前工作目录
复制成功
复制失败
设置np名称为p的名称
释放内存并返回-1
设置np状态为RUNNABLE
结束
返回子进程的pid

exec系统调用


开始
获取当前进程p的指针
复制当前进程的kpagetable
复制成功
分配新的pagetable
查找文件路径
查找成功
读取ELF头
ELF头有效
加载程序的段到内存中
分配新的用户栈并复制参数到用户栈中
为用户程序设置参数并设置为进程的入口点
保存程序名称
参数无效
分配用户栈失败
复制参数失败
加载程序段失败
ELF头无效
查找失败


切换到新的pagetable并释放旧的pagetable
保存程序名称失败
切换pagetable失败
返回参数argc
跳转到bad
释放资源

wait系统调用


开始
获取当前进程p的锁
扫描进程表
找到退出的子进程
获取子进程np的锁
将np的xstate复制到addr
复制成功
释放子进程np的锁并删除进程描述符
释放当前进程p的锁并返回子进程pid
复制失败
释放子进程np的锁并返回-1
没找到退出的子进程
没有子进程或进程已被杀死?
是
释放当前进程p的锁并返回-1
否
等待子进程退出并重新扫描进程表