操作系统 Lab2 进程与线程

宁晨然 17307130178

一、实验目的

了解 xv-6 进程和线程的处理过程，理解 xv-6 代码中 allocproc 等函数的功能和如何实现。通过阅读代码加深对 xv-6 操作系统实现过程的理解。

二、实验过程

（一）Allocproc 函数

①作用：该函数用于每次创建进程的时候分配 process 所需要的槽、设置初始化状态、为其内核线程的运行做准备，并返回创建的进程 p。具体实现如下。

②实现过程：

```
static struct proc*
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;
```

A.根据“槽”分情况讨论：

首先创建 proc 结构体（即一个进程，结构体定义在 proc.h），若要继续下去，则需要先获得锁。

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

左图是 ptable，即进程表，其中有“锁”和进程数组。而要创建 proc 的时候必须确保能够获得锁。acquire()获得锁，如果不能获得锁则一直 spin，直到可以获得锁。

然后遍历 ptable 进程表：

如果有 UNUSED 未使用的进程槽，就将用该槽作为进程槽可以创建该进程，继续执行 found。如果没有成功找到空闲槽，则释放锁，并返回 0 值。

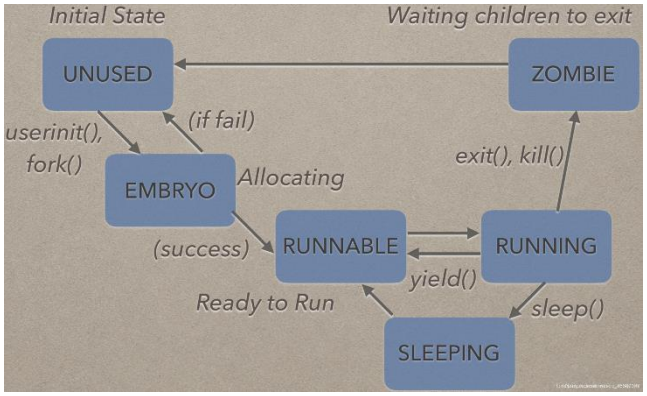
B.找到槽后，执行 Found：

```
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.
  sp -= 4;
  *(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;

  return p;
}
```

现在需要做的事情就是给刚创建的 proc 进程 p 初始化状态。设置 p->state 状态为 EMBRYO，此处可以根据 proc 结构体定义中的 state，发现共有以下状态。

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

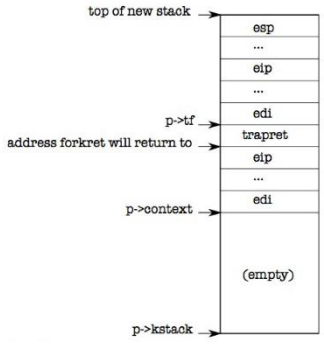
可以根据下图判断该状态的意思就是“正在分配相关信息的进程”，如果成功完成进程就变成就绪进程 RUNANBLE，如果失败就返回未使用进程 UNUSED。



注意到 p->pid=nextpid++，此处 nextpid 是一个全局变量，目的为了分配每一个进程不同的 pid，所以每次都会++保证唯一性。此时释放锁。

C.分配内核栈区：

使用 kalloc()分配内核栈区，该函数的作用是，分配一个 4096byte 的物理内存页，并返回内核可以使用的指针，如果内存无法分配则返回 0，才有 if 语句中判断是否为 0，如果没有分配成功栈，则将 proc 进程返回 UNUSED 状态标记失败。



sp 是一个 char*指针类型，赋值为 p->kstack 栈的顶部 top。

D.分配 trap frame 空间等:

首先 sp(stackpoint)减去 trapframe 所需要的空间大小，然后用该空间创建一个 p->tf，再减 4 用于分配 trapret，再分配一部分空间用于存放 p->context。

E.上述空间的用途:

allocproc 需要设置新进程的内核栈，此处用了一种方式使得创建第一个进程时能使用也可以在后续 fork 操作创建进程时被使用。函数设置好内核栈和一系列内核寄存器，使得进程第一次运行的时候会返回到用户空间。准备好的内核栈如上图，此时 allocproc 通过设置返回程序计数器的值，使得新进程的内核线程首先运行在 forkret 的代码中，然后返回到 trapret 中运行。

此时内核线程会从 p->context 中拷贝好的内容开始运行，所以将 p->context->eip 指向 forkret，让内核线程从 forkret 开始运行，接着返回到那个时刻栈底的地址，则栈指针指向了 p->context 的末尾。又因为 p->context 在内核栈上，上方有一个 trapret 的指针，所以 forkret 运行结束后会返回 trapret，接着 trapret 就从栈顶恢复用户寄存器，然后跳转到 proc 的代码。

③总结 allocproc

该函数分配并初始化了进程，可以供第一次创建进程（userinit）调用也可以供后面 fork 创建进程调用。实现过程主要由，在进程表中找到未使用的槽创建新进程，初始化进程状态，设置进程内核栈三个步骤组成，最后返回创建的进程（若未成功返回 0）。

（二）proc.c 中 fork(),wait(),exit()函数

1.fork()

fork()函数的功能：创建一个以 p 为父进程的新进程，像系统调用一样设置返回的栈，调用者必须将返回的进程设置为就绪。

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;
```

①通过 myproc()获取当前进程，使用 allocproc 创建新的进程，如果创建失败则返回-1。创建成功则将父进程的状态复制到该新进程：

使用 copyuvm()函数复制父进程的页表，该函数的返回值是父进程的页表，如果失败返回 0，并且释放子进程的栈空间，重设状态为 UNUSED，成功则继续。

②复制 size，即进程内存所占大小。并设置子进程的父节点为父进程，子进程的 trapframe 指向父进程的 trapframe，但需要设置子进程中的 eax（返回值）为 0。

```
  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

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

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

  pid = np->pid;

  acquire(&ptable.lock);

  np->state = RUNNABLE;

  release(&ptable.lock);

  return pid;
}
```

③设置进程访问的文件，使用 fileup 函数可以增加父进程所有访问的文件的被访问次数 f->ref++,idup 同

理。

④再复制进程的名字，pid 保存子进程的 pid 值（用于返回）。最后需要先请求锁，请求后设置子进程为就绪状态，再释放锁。返回值为子进程的 pid。

2.exit()

exit()函数的功能：退出当前进程但不返回，已退出进程保持 zombie 状态直到父进程发现该子进程已经退出。

```
void
exit(void)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;

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

  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){
    if(curproc->ofile[fd]){
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd] = 0;
    }
  }

  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");
}
```

同 fork()使用 myproc()函数获取当前进程 curproc()，如果当前进程是第一个初始进程则无法退出。

①关闭所有文件：遍历当前进程打开的文件，逐个关闭。使用了 fileclose()函数，该函数会更改文件被打开次数 f->ref。

②取消对 inode 节点的引用：由于 iput 是一个 FS 系统调用，所以调用前后需要使用 begin_op()和 end_op()函数。iput()取消了当前进程对 inode 的引用。

③唤醒父进程：请求锁之后，wakeup1()唤醒所有链上休眠状态的进程转换为就绪状态，此处唤醒父进程。

④转交子进程：遍历所有进程表中的进程，如果进程的父进程是当前需要退出的当前进程，则将子进程的父节点连接为 initproc 初始进程。并且如果子进程为 zombie 状态，则唤醒 initproc 进程。

⑤设置进程为状态 zombie，并使用 sched()调度程序。sched()的作用是，将当前进程保存的寄存器的值(context)存放进 cpu->scheduler 调度程序中，用于恢复寄存器之前的状态。

exit()仅退出了进程，但是没有返回。

3.wait()

wait()函数的功能：等待子进程退出，返回子进程的 pid，若无子进程则返回-1。

```
int
wait(void)
{
  struct proc *p;
  int havekids, pid;
  struct proc *curproc = myproc();

  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != curproc)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        // Found one.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        release(&ptable.lock);
        return pid;
      }
    }

    // No point waiting if we don't have any children.
    if(!havekids || curproc->killed){
      release(&ptable.lock);
      return -1;
    }

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

①同理前获取当前进程 myproc()

②遍历进程表查找所有子进程：havekids 即是否有子进程。当查找到有一个子进程时，判断是否处于 zombie 状态（这个状态表示子进程已经退出，等待返回），如果查找到一个后：

设置返回值 pid 为该子进程的 pid，释放子进程的栈空间和页表空并且设置该子进程的 pid/parent/name/killed 均为 0，并将状态设置未使用 UNUSED，释放锁后返回 PID。

③若当前进程被杀死，或者没有子进程，则返回-1。

④若子进程没有处于 zombie，则等待子进程退出 exit()，该进程处于休眠状态。

（三）system call

syscall()：对于系统调用，trap 会调用 syscall()，syscall 从中断帧中读出系统调用号，即保存的%eax，%eax 保存的就是 sys_exec，syscall()就会调用系统调用函数表的第 sys_exec 个表项，并执行该 syscalls[num]()函数，将返回值存进%eax 中。

```
void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();

  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

如果%eax 存值不合法，则无对应系统调用，则%eax=-1，并且输出报错信息。

此处系统调用函数被封装起来了，通过系统调用实际函数表，eax 作为索引直接查表执行系统调用功能，实现了对用户封装的系统调用接口，只允许用户得到系统调用的结果，而系统调用的接口内部则是内核中一系列已经具备一定功能的多内核函数，调用请求处理传递给内核，调用后由内核函数处理后，将处理结果返回给调用者（用户/应用程序）。

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

左图就是系统调用封装接口表的查表索引图，用户通过进程中 eax 的值获取系统调用的接口函数，得到返回值也传递给 eax。

修改 syscall()获取父进程 PID 即如下图。

添加一个封装好的 syscall 函数：

①syscall.h

```
#define SYS_get_parentpid  22
```

syscall 里面需要先定义函数宏

②syscall.c

```
extern int sys_get_parentpid(void);
[SYS_get_parentpid]   sys_get_parentpid,
```

在系统调用中声明该函数，并且在 syscalls 数组中添加该函数名字

③sysproc.c

```
int sys_get_parentpid(void)
{
  if(myproc()->parent)
    return myproc()->parent->pid;
  else return -1;
}
```

在 sysproc 处添加实现函数的细节，这个函数是供系统调用使用，所以封装好了。如果没有父进程，则返回-1.

④user.h

```
int get_parentpid(void);
```

这是给用户调用的函数，需要声明

⑤usys.S

```
SYSCALL(get_parentpid)
```

用户调用的函数实现细节，调用实际就是调用 sys_get_parentpid

⑤usertest.c

```
void
forktest(void)
{
  int  pid;

  printf(1, "fork test\n");

pid = fork();
  if(pid<0) printf(1,"error in fork!\n");
  else if(pid==0)
  {
        int temp1 = getpid();
        int temp2 = get_parentpid();
    printf(1, "current process pid : %d\n", temp1);
    printf(1, "parent process pid : %d\n", temp2);
  }
  else
  {
        int temp = wait();
        printf(1,"parent process pid : %d\n",temp);
  }
```

由于 main 函数会执行 forktest，所以在测试中添加 pid 的测试，如图会输出当前进程的 pid 和父进程 pid。

执行结果：

```
usertests starting
fork test
current process pid : 5
parent process pid : 4
parent process pid : 5
```

三、实验感受

本次实验的重点在于阅读代码，所以并没有直接在 ubuntu 里面看，在 vscode 里面方便

查看函数的定义来源，各代码的链接关系。阅读代码结合了 xv-6 手册，并且查阅了一些资料才朦朦胧胧看懂，感觉实验难度略大。

理解了进程调用的过程、创建、状态修改等基本操作需要更改、注意的地方，所以更加加深了对这方面知识的应用和实际操作能力。