实验二 进程与线程

马逸君 17300180070

Question 1 阅读allocproc函数，分析这个函数的原理并解释这个函数的作用。

作用：限制同时在使用中的线程数量，如果未超出，则将新申请的进程准备好，供申请者使用。

原理概述：allocproc()函数遍历进程池（进程池是一个大小为常量的进程数组），寻找状态为UNUSED的进程，若未找到则返回0，若找到则将其状态设为EMBRYO，并对其状态进行必要的初始化工作（包括分配进程标识符、分配内核中的栈空间、为当前调用的“陷阱帧”(trap frame)预留空间、设置寄存器现场），以使得该进程可以投入运行。

详细原理：（见代码注释）

```
static struct proc*
allocproc(void) // 返回所申请到的新进程的指针，失败则返回0（空指针）
{
    struct proc *p; // 用于遍历进程池的循环变量
    char *sp;

    acquire(&ptable.lock); // 给进程表上锁，互斥访问

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    // 遍历进程池（一个大小为常量的进程数组）
        if(p->state == UNUSED) // 寻找状态为UNUSED(未用,可分配给新进程)的进程
            goto found; // 若找到，则进行必要初始化工作，分配给新进程（见后）

    // 未找到UNUSED进程
    release(&ptable.lock); // 互斥访问过程结束，给进程表解锁
    return 0; // 返回0（空指针），表示分配进程失败

found: // 找到可分配的UNUSED进程
    p->state = EMBRYO; // 将其状态设为EMBRYO(“胚胎”，即将被分配给新进程)
    p->pid = nextpid++; // 分配进程标识符

    release(&ptable.lock); // 访问进程表结束，给进程表解锁

    // 新进程的内核栈空间的分配
    if((p->kstack = kalloc()) == 0){ // 内核申请物理内存，作为新进程栈空间
    // kalloc()函数定义在kalloc.c中，分配一个4KB大小的物理内存的页面
    p->state = UNUSED;
    return 0;
    // 如果申请失败，则不能分配新进程。将新进程状态恢复为UNUSED，函数返回0
    }
    sp = p->kstack + KSTACKSIZE; // 预留KSTACKSIZE(=4K)大小的空间

    // 为新进程的陷阱帧(trap frame)预留空间
    // 陷阱帧：异常发生时寄存器组的值，异常/中断处理完毕后从中读取恢复寄存器组的值并继续执行原来过程
    sp -= sizeof *p->tf; // 新进程栈空间的最高处，预留sizeof(trapframe)大小的空间给陷阱帧
    p->tf = (struct trapframe*)sp; // 设定新进程陷阱帧的指针指向这块预留空间的首地址
```

```
37
38      // 设置新进程的上下文使其从forkret()开始执行，这个函数返回到trapret()中
39      // 读注释知，forkret()函数设计用于fork()新产生的子进程的首次调度，也适用于本函数产生的新进程。trapret()恢复段寄存器的值，跳转回用户空间。
40      sp -= 4;
41      *(uint*)sp = (uint)trapret; // 在剩余栈空间顶部压入trapret()函数的地址
42
43      sp -= sizeof *p->context; // 在剩余栈空间顶部预留sizeof(context)的空间给上下文
44      //上下文是一组被调用者保存寄存器,是调用者寄存器组的暂存,上下文切换和第一次运行时起作用
45      p->context = (struct context*)sp; // 设定新进程上下文的指针指向预留空间
46      memset(p->context, 0, sizeof *p->context); // 将新进程的上下文初始化为全0
47      p->context->eip = (uint)forkret; // 设定新进程的上下文eip指向forkret。这样一来，一旦新进程进入执行态，就会开始执行forkret()函数。
48
49      return p; // 返回新进程指针
50  }
```

Question 2 阅读proc.c中fork(),wait(),exit()函数的实现，分析这些函数是如何实现对应功能的。

fork()函数的功能是创建自身进程副本，该新进程将自身进程作为父进程。实现其功能的思路是，创建一个新进程，其继承当前进程的状态、打开文件、进程名信息，且其父进程指针指向当前进程，即这个新进程就是当前进程的子进程。

工作过程概述：首先申请一个新进程，然后将当前进程的状态（包括逐页复制进程的地址空间、复制进程的大小、陷阱帧信息）复制给新进程并设置新进程的父进程指针指向当前进程，使新进程打开和当前进程一样的文件和目录（且增加被打开文件和目录的计数器），将当前进程的进程名复制给新进程，将新进程置为RUNNABLE状态，最后返回新进程的PID。另外，鉴于fork()与典型的系统调用不同，它由父进程调用，但父进程和子进程两者都需要从其中返回，所以我们还需要为子进程的%eax（返回值）置0（而父进程得到的返回值即为函数的返回值，以便在调用fork()后的代码段区分父进程和子进程，让它们做出不同的行为）。如果上面申请新进程或是给新进程的状态表申请空间失败，则返回-1。

详细分析：

```
1   int
2   fork(void) // 向调用者返回衍生的子进程的pid，失败则返回-1。
3   {
4
5       int i, pid; // 临时变量，i用于循环，pid用于暂存子进程pid作为返回值
6       struct proc *np; // 新进程指针
7       struct proc *curproc = myproc(); // 当前进程指针
8
9       // 从进程池分配一个新进程为子进程
10      if((np = allocproc()) == 0){
11          return -1; // 如果失败，fork()函数在当前进程中返回-1
12      }
13
14      // 将当前进程的状态复制给子进程
15      if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){ // 复制页表
16          kfree(np->kstack);
17          np->kstack = 0;
18          np->state = UNUSED;
19          return -1;
20          // 若页表复制失败，则fork失败，将申请到的新进程恢复原始状态，返回-1
21      }
22      np->sz = curproc->sz; // 复制进程内存大小
```

```
    np->parent = curproc; // 设置新进程的父进程指针指向当前进程
    *np->tf = *curproc->tf; // 复制陷阱帧

    // 将子进程的%eax寄存器置0
    // 这样做的原因是，父进程(调用者)和子进程两者都需要从fork()中返回，我们通过为子进程%eax赋值的方式为其赋予返回值。子进程得到的返回值为0，而父进程得到的返回值（即为fork()函数直接return的返回值）等于子进程的pid（非0）。因为子进程也是从fork()函数的返回地址开始执行，这样一来，在调用fork()后的代码段就可以用if语句判断返回值来区分父进程和子进程，让它们做出不同的行为，如if (fork() == 0)后的语句就只有子进程会执行，父进程不会执行。
    np->tf->eax = 0;

    // 使新进程打开和当前进程一样的文件和目录
    for(i = 0; i < NOFILE; i++) // 遍历打开文件列表
        if(curproc->ofile[i]) // 如果父进程打开了一个文件
            np->ofile[i] = filedup(curproc->ofile[i]); //则让子进程也打开它
   // filedup(file*)函数：给其参数（文件）的引用计数器增加1，然后返回其参数的值
    np->cwd = idup(curproc->cwd); // 复制父进程当前处于的目录给子进程
    // idup(inode*)函数：给其参数（inode，未命名单文件）的引用计数器增加1，然后返回其参数的值

    // 将当前进程的进程名复制给新进程
    safestrcpy(np->name, curproc->name, sizeof(curproc->name));

    pid = np->pid; // 读取新进程的pid，准备返回

    acquire(&ptable.lock); // 开始访问进程表，进程表要求互斥访问，上锁

    np->state = RUNNABLE; // 在进程表中，将新进程置为RUNNABLE(就绪)状态

    release(&ptable.lock); // 访问完成，解锁

    return pid; // 返回新进程的PID
}
```

wait()函数的功能是等待子进程退出。实现其功能的思路是，在进程表中找出当前进程的处于ZOMBIE状态的子进程并完成它的终结工作，否则休眠并等待任一子进程exit()（或其他原因）将当前进程唤醒。（因为ZOMBIE状态是一个子进程已经退出、等待父进程完成其清理的状态）

工作过程概述：无限循环遍历进程表，如果发现当前进程有正处于ZOMBIE状态的子进程，则将其终止并初始化放回线程池中，返回其PID；如果当前进程已经没有子进程或是已经被杀死，则返回-1；如果当前进程有非ZOMBIE状态的子进程，则当前进程进入休眠状态等待（若有子进程exit()，则将唤醒当前进程），当前进程若被唤醒则将继续这个函数遍历进程表的操作。

详细分析：

```
int
wait(void)
{
    struct proc *p; // 用于遍历进程池，循环变量
    int havekids, pid; // havekids的含义是当前进程是否有子进程，1表示有，0表示无；pid用于暂存返回值（被终止的子进程的pid）
    struct proc *curproc = myproc(); //当前进程

    acquire(&ptable.lock); // 给进程表上锁，互斥访问
    for(;;){
    // (无限循环)遍历进程表，查找正处于ZOMBIE状态的子进程(ZOMBIE: 僵尸态，子进程已经退出、等待父进程完成其清理工作的状态)
```

```
        havekids = 0; // 初始化“有子进程”为0
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){ //遍历进程表
            if(p->parent != curproc) // 遍历到的进程不是当前进程的子进程
                continue; // 继续尝试进程表的下一个进程
            havekids = 1; // 找到子进程
            if(p->state == ZOMBIE){ // 找到ZOMBIE状态子进程
                pid = p->pid; // 记录其pid，作为返回值
                kfree(p->kstack); // 释放其内核栈空间
                p->kstack = 0; // 释放后，清除内核栈指针
                freevm(p->pgdir); // 关闭其打开的目录
                p->pid = 0; // 清除其pid变量的值
                p->parent = 0; // 清除父进程指针
                p->name[0] = 0; // 清除进程名
                p->killed = 0; // 清除“被终止”标志
                p->state = UNUSED; // 将状态置为UNUSED
                release(&ptable.lock); // 互斥访问过程结束，给进程表解锁
                return pid; // 返回被终止的子进程的PID
            }
            // 如果找到的子进程不是ZOMBIE状态，则什么都不做
        }

        if(!havekids || curproc->killed){ // 如果当前进程没有子进程，或已经被杀死
            release(&ptable.lock); // 互斥访问结束，给进程表解锁
            return -1; // 返回-1
        }

        // 当前进程有非ZOMBIE状态的子进程
        sleep(curproc, &ptable.lock); // 进入休眠状态，等待有子进程exit()。若有子进程exit()，则将调用wakeup1()，唤醒当前进程
    }
}
```

exit()函数的功能是结束当前进程。实现其功能的思路是，解除将被终止的进程与磁盘文件和目录的关联，设置其状态为ZOMBIE并通知父进程，妥善处理子进程的“后事”，然后将控制权交给scheduler，这样就完成了当前进程的退出工作。

工作过程概述：关闭当前进程打开的所有文件、解除对当前目录的引用、唤醒父进程、将当前进程退出后遗留的所有子进程的父指针修改到initproc（这其中如果有ZOMBIE状态的子进程，则将新父进程initproc唤醒）、设置当前进程的状态为ZOMBIE、将控制权交给scheduler（sched()永不返回）。如果发现sched()返回到当前行，或是当前调用的exit()函数企图终止initproc这个进程，则调用panic()，中止整个xv6并报错。

详细分析：

```
void
exit(void) // 结束当前进程。该函数不会返回。
{
    struct proc *curproc = myproc(); // 当前进程
    struct proc *p; // 用于遍历进程池，循环变量
    int fd; //

    if(curproc == initproc) // 如果发现initproc这个进程试图exit
        panic("init exiting"); // 调用panic()，中止整个xv6并报错

    // 关闭当前进程打开的所有文件
    for(fd = 0; fd < NOFILE; fd++){ // 遍历所有打开的文件
        if(curproc->ofile[fd]){ // 如果当前进程打开了这个文件
```

```
            fileclose(curproc->ofile[fd]); // 当前进程关闭这个文件
            curproc->ofile[fd] = 0; // 清除当前进程打开这个文件的标记
        }
    }

    begin_op(); // 该函数在每次涉及文件系统(FS)的系统调用(syscall)前都要调用，功能是统计正在执行的FS syscall个数增加1
    iput(curproc->cwd); // 解除对当前目录的引用
    end_op(); // 该函数在每次涉及文件系统的系统调用后都要调用，功能是给正在执行的FS syscall个数减去1，若个数变为0则提交日志更改，记入日志
    curproc->cwd = 0; // 清除当前进程的“当前目录”指针

    acquire(&ptable.lock); // 给进程表上锁，互斥访问

    wakeup1(curproc->parent); // 唤醒可能正在wait()中休眠的父进程

    // 将当前进程所有子进程的父指针修改到initproc（防止它们的父进程指针无效，成为“孤儿”进程）
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){ // 遍历进程表
        if(p->parent == curproc){ // 如果找到当前进程的子进程
            p->parent = initproc; // 修改其父指针为initproc
            if(p->state == ZOMBIE) // 如果该子进程在ZOMBIE状态
                wakeup1(initproc); //唤醒新父进程initproc（可能正在wait()中休眠）
        }
    }

    curproc->state = ZOMBIE; // 设置当前进程的状态为ZOMBIE
    sched(); // 将控制权交给scheduler，永不返回
    panic("zombie exit"); // 如果sched()返回了，调用panic()中止整个xv6并报错
}
```

选做 分析syscall的过程，修改syscall获取父进程ID。

syscall过程概述：用户代码将所需调用的系统调用的编号存入%eax，将该系统调用所需的参数按顺序压栈，然后发出int 64指令来完成需要的系统调用。（形式上是通过调用这些系统调用函数对应的用户态函数来完成，这些用户态函数的定义在user.h中，实现在usys.S中。用户程序调用用户态函数时系统首先将参数按顺序压栈，然后去调用汇编语言写成的用户态函数体，函数体中完成将编号存入%eax和发出int 64指令的操作。）

代码分析：

首先是syscall.c，该文件包含syscall()函数体。

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

  num = curproc->tf->eax; // 所需调用的系统调用的编号存放在%eax中
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num](); // 调用系统调用对应的内核函数
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
```

```
  }
}
```

syscalls数组中存放的是指向对应内核函数的函数指针，数组中各元素的下标对应其系统调用编号。

```
static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
...
};
```

这些内核函数的函数体定义在不同的源码文件中。如sysproc.c中的sys_getpid()：

```
int
sys_getpid(void)
{
  return myproc()->pid;
}
```

用户代码通过调用内核函数对应的用户态函数来进行系统调用，如usertests.c：

```
void
mem(void)
{
  void *m1, *m2;
  int pid, ppid;

  printf(1, "mem test\n");
  ppid = getpid(); // getpid()函数是getpid系统调用对应的用户态函数

  ...
}
```

这些用户态函数定义在user.h中：

```
// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
...
int uptime(void);
```

用户态函数的实现是在usys.S中：

```
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
# 宏展开后，每行扩展为一个汇编过程，它的行为是：将对应的系统调用编号送%eax，然后执行int 64这条指令。

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
...
```

为了直观地展示用户态函数的实现，我们可以在.asm文件中查看宏展开的结果，如usertests.asm：

```
00003922 <getpid>:
SYSCALL(getpid)
    3922:   b8 0b 00 00 00          mov    $0xb,%eax
    3927:   cd 40                   int    $0x40
    3929:   c3                      ret
```

以上是无参数系统调用相关的代码段。为了说明有参数系统调用发生时用户代码传参的方式，此处引用usertests.asm中对write()这个系统调用的调用语句：

```
  int cc = write(fd, buf, 512);
3442:   83 ec 04                sub    $0x4,%esp
3445:   68 00 02 00 00          push   $0x200
344a:   68 e0 85 00 00          push   $0x85e0
344f:   57                      push   %edi
3450:   e8 6d 04 00 00          call   38c2 <write>
```

作为参考，write()的定义如下（user.h）：

```
int write(int, const void*, int);
```

可以看到，在调用write这个汇编过程之前，系统首先将write后面的三个函数依次压栈，然后再去call write。

以上就是对syscall相关代码的详细分析。

给syscall增加获取父进程ID功能：

首先读取另一个功能类似的syscall——sys_getpid的定义，作为仿照样本：

```
mayijun@ubuntu:~/xv6$ make clean

mayijun@ubuntu:~/xv6$ ls * | xargs grep -i -n 'getpid'
syscall.c:92:extern int sys_getpid(void);
syscall.c:119:[SYS_getpid] sys_getpid,
syscall.h:12:#define SYS_getpid 11
sysproc.c:40:sys_getpid(void)
user.h:22:int getpid(void);
usertests.c:434: ppid = getpid();
usertests.c:1498:  ppid = getpid();
usys.S:28:SYSCALL(getpid)
```

（值得一提的是最后一行的usys.S，一开始我只搜索了.c和.h文件（即ls *.c *.h），漏掉了汇编语言源文件.S，结果实现不完全，后来才发现问题并改正。特此批注，以示强调。）

故首先在sysproc.c中添加新函数的函数体（注：语法上来说不一定要写在sysproc.c中，但是因为新函数实现的功能也是进程相关，故将其实现在sysproc.c中）：

```
int
sys_getfpid(void)
{
    return myproc()->parent->pid;
}
```

然后在syscall.c中把新函数的函数指针添加到函数指针数组syscall中（需先外部引用一下新函数的定义）：

```
extern int sys_getfpid(void);
[SYS_getfpid] sys_getfpid
```

在syscall.h中为新函数分配编号：

```
#define SYS_getfpid 22
```

在user.h中添加新函数的用户态定义：

```
int getfpid(void);
```

在usys.S中添加新函数的用户态实现：

```
SYSCALL(getfpid)
```

测试：在xv6目录下新建源码文件getfpid.c如下

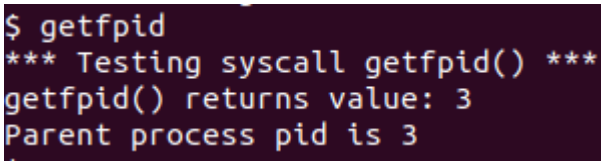
```c
#include "types.h"
#include "user.h"

int main()
{
    printf(1, "*** Testing syscall getfpid() ***\n");
    if (fork() == 0) printf(1, "getfpid() returns value: %d\n", getfpid());
    else {wait(); printf(1, "Parent process pid is %d\n", getpid());}
    exit();
}
```

并修改Makefile，在UPROGS字段下添加getfpid

```
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
    _getfpid\
```

最后在qemu中实验如下



参考资料：

[1] https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev8.pdf

xv6用户文档 Russ Cox, Frans Kaashoek, Robert Morris 2014.9.3

[2] https://stackoverflow.com/questions/47851969/what-is-trap-frame-and-what-is-difference-between-trap-frame-and-task-struct

StackOverflow - [What is trap frame? And what is difference between trap frame and task_struct?] Tarak Patel 2017.12.17