OS_Assignment3

Assign	
Property	
tag	homework
姓名	周鹏宇
学号	2019K8009929039

描述

以下代码为 Linux/Unix 创建进程的例子，其中

- fork 克隆出一个进程
- exec覆盖掉当前进程

```
if ((pid = fork()) == 0) {
    /* child process */
    exec("foo");    /* does not return */
} else {
    /* parent */
    wait(pid);      /* wait for child to die */
}
```

作业要求

请大家阅读 xv6 源码，撰写一份分析报告。报告需要给出以下信息：

1. 在 xv6 中，PCB 信息是如何表示的？是用什么数据结构？存放在哪个文件中？
2. 上述代码中，修改了哪些 PCB 中的信息？

1. 在xv-6中，PCB由位于`proc.h`中定义的结构体`proc`刻画

```
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)
};
```

- sz：进程所占内存的大小，单位为字节
- pgdir：进程的页表，其还记录了保存进程内存的物理页地址
- kstack：进程的内核栈栈底。进程由用户栈和内核栈，当进程陷入内核时，会在内核栈中执行
- state：记录进程的状态，可以注意到共有`UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE`六个状态，除了已经提及的运行（`RUNNING`），就绪（`RUNNABLE`）和阻塞态外，还有诸如僵尸态（`ZOMBIE` 进程可以处于已退出但尚未清理的最终(final)状态），初始态（`EMBRYO` 进程在创建时处于的状态）等等
- pid：进程ID，用于标记不同进程
- parent：当前进程的父进程
- tf：当前系统调用的陷阱帧
  - 其中`struct trapframe` 为：

```
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 kernel
  uint esp;
  ushort ss;
  ushort padding6;
};
```

    - 当内核获取控制权时，用户寄存器被以`trapframe`的形式保存在内核栈上

- context:切换进程时保存的寄存器，包含`edi`、`esi`、`ebx`、`ebp`、`eip`五个寄存器
- chan：若非0，则进程睡眠且保存至其指向的地址
- killed：若非0，则进程已被杀死
- ofile[NOFILE]：记录打开的文件
    - 其中`struct file`为：

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off;
};
```

- cwd：进程当前的工作目录
- name：进程名（便于调试）

2. 代码解读：

   - 功能：从主进程克隆出一个子进程，该子进程执行文件foo，同时主进程等待子进程运行结束

   本段代码用到的三个函数：

```
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;

  // 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;
}
```

   fork()系统调用是操作系统提供的**创建新进程**的方法。新创建的进程几乎与调用进程完全一样，对操作系统来说，这时看起来有**两个完全一样**的程序在运行，并都

从 fork()系统调用中返回，新创建的进程称为**子进程**(child)， 原来的进程称为**父进程**(parent)。子进程并**不是完全拷贝**了父进程。具体来说，虽然它拥有自己的地址空间(即拥有自己的私有内存)、寄存器、程序计数器等，但是它从 fork()返回的值是不同的。父进程获得的返回值是**新创建子进程的 PID**，而子进程获得的返回值是 **0**。它的输出**不确定的(deterministic)**

其具体行为为：在为子进程创建 PCB (即 structproc) 后，fork 函数从父进程的页表中复制内容给子进程的页表 pgdir，设置子进程与父进程相同的sz，指定子进程的 parent 域为父进程，复制父进程的陷阱帧给子进程，并将子进程 `tf` 中的 `eax` 域置零，设置子进程的 `ofile`， `cwd`， `name` 域与父进程相同，指定子进程状态 (state) 为 RUNNABLE。

```
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
```

```
  }
}
```

父进程调用 wait()，延迟自己的执行，直到子进程执行完毕。当子进程结束时，wait()才返回父进程。如果父进程碰巧先运行，它会马上调用 wait()，该系统调用会当子进程运行结束后才返回。

其具体行为为：wait 函数会不断循环在进程表中查找一个当前进程的子进程且其状态为 ZOMBIE，查找成功后清空该子进程的内核栈与页表，将进程名 name 置为空串， 将 pid，parent，killed 均置为 0，指定其状态为 UNUSED。

```
int
exec(char *path, char **argv)
{
  char *s, *last;
  int i, off;
  uint argc, sz, sp, ustack[3+MAXARG+1];
  struct elfhdr elf;
  struct inode *ip;
  struct proghdr ph;
  pde_t *pgdir, *oldpgdir;
  struct proc *curproc = myproc();

  begin_op();

  if((ip = namei(path)) == 0){
    end_op();
    cprintf("exec: fail\n");
    return -1;
  }
  ilock(ip);
  pgdir = 0;

  // Check ELF header
  if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
    goto bad;
  if(elf.magic != ELF_MAGIC)
    goto bad;

  if((pgdir = setupkvm()) == 0)
    goto bad;

  // Load program into memory.
  sz = 0;
  for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
      goto bad;
    if(ph.type != ELF_PROG_LOAD)
      continue;
    if(ph.memsz < ph.filesz)
      goto bad;
    if(ph.vaddr + ph.memsz < ph.vaddr)
      goto bad;
```

```
    if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
      goto bad;
    if(ph.vaddr % PGSIZE != 0)
      goto bad;
    if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
      goto bad;
  }
  iunlockput(ip);
  end_op();
  ip = 0;

  // Allocate two pages at the next page boundary.
  // Make the first inaccessible.  Use the second as the user stack.
  sz = PGROUNDUP(sz);
  if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
    goto bad;
  clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
  sp = sz;

  // Push argument strings, prepare rest of stack in ustack.
  for(argc = 0; argv[argc]; argc++) {
    if(argc >= MAXARG)
      goto bad;
    sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
    if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
      goto bad;
    ustack[3+argc] = sp;
  }
  ustack[3+argc] = 0;

  ustack[0] = 0xffffffff;  // fake return PC
  ustack[1] = argc;
  ustack[2] = sp - (argc+1)*4;  // argv pointer

  sp -= (3+argc+1) * 4;
  if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
    goto bad;

  // Save program name for debugging.
  for(last=s=path; *s; s++)
    if(*s == '/')
      last = s+1;
  safestrcpy(curproc->name, last, sizeof(curproc->name));

  // Commit to the user image.
  oldpgdir = curproc->pgdir;
  curproc->pgdir = pgdir;
  curproc->sz = sz;
  curproc->tf->eip = elf.entry;  // main
  curproc->tf->esp = sp;
  switchuvm(curproc);
  freevm(oldpgdir);
  return 0;

 bad:
  if(pgdir)
    freevm(pgdir);
  if(ip){
```

```
    iunlockput(ip);
    end_op();
  }
  return -1;
}
```

exec()系统调用用可以让子进程执行与父进程我同的程序。给定可执行程序的名称及需要的参数后，exec()会从可执行程序中加载代码和静态数据，并用它覆写自己的代码段(以及静态数据)，堆、栈及其他内存空间也会被重新初始化。然后操作系统就执行该程序，将参数通过 argv 传递给该进程。因此，它并没有创建新进程，而是直接将当前运行的程序替换为相同的运行程序。子进程执行 exec()之后，几乎就像原进程从未运行过一样。对 exec()的成功调用永远不会返回。

参考文献：

1. The xv6 Operating System”Russ Cox, Frans Kaashoek, Robert Morris, Nickolai Zeldovich.
2. Operating Systems Three Easy Pieces