xv6©-rev10

The **fork** system call

Carmi Merimovich

Tel-Aviv Academic College

November 30, 2017

sys_fork

```
3760  int sys_fork(void) {
        return fork();
      }
```

`fork()`

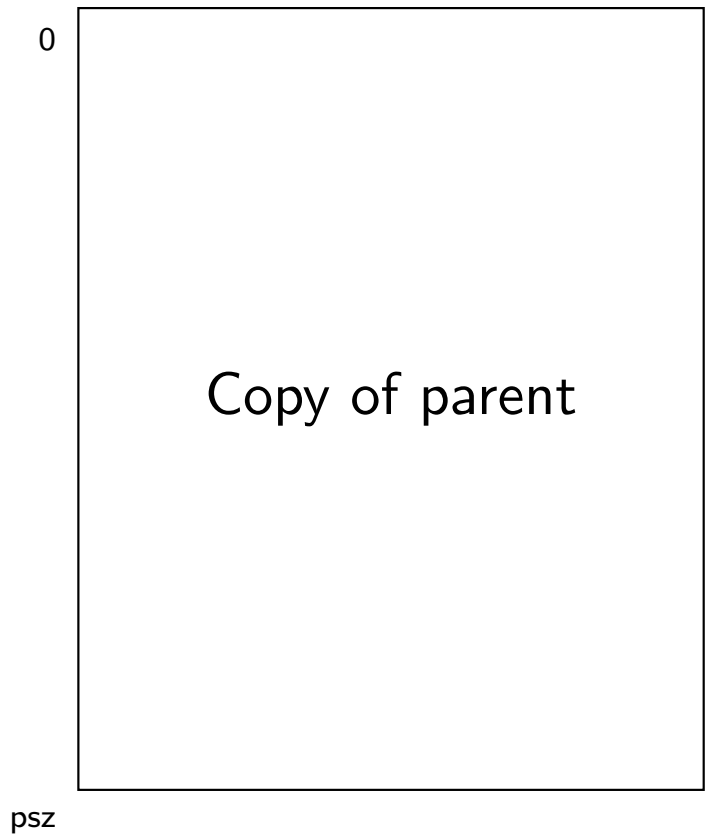
Recall:

- A child (of the invoker) process is created.
- The `pid` of the child process is returned to the invoker.
- The child process is (almost) identical to the parent process.
  - To the child, the return value of the system call is zero.
- So, how do we begin?

Child process state needed

eax	0
ebx	pebx
ecx	pecx
edx	pedx
ebp	pebp
esi	pesi
edi	pedi
esp	pesp
eip	peip

cs	pcs
ds	pds
ss	pss
es	pes
fs	pfs
gs	pgs



proc struct

How do we fill the fields of the new process?

```
uint sz; // @proc->sz@
pde_t* pgdir; // @Serious replication needed@
char *kstack; // @probably allocproc()@
enum procstate state; // @RUNNABLE@
volatile int pid; // @allocproc()@
struct proc *parent; // @proc@
struct trapframe *tf; // @allocproc()@
struct context *context; // @allocproc()@
void *chan; // @0@
int killed; // @0@
struct file *ofile[NOFILE]; // @filedup()@ (when st
struct inode *cwd; // @idup()@ (when studying fs)
char name[16]; // @proc->name@
};
```

Needed work

- It is not clear what to put in the new `trapframe`.
- Filling `pgdir` requires considerable replication code.

Replicating current process first page

- Allocating new block of memory:

```
dst = kalloc();
```

- Copying. One of the following is possible:

1. ```
   memmove(dst,0,4096);
   ```

2. ```
   p = walkpgdir(myproc()->pgdir,0,0);
   memmove(dst,p2v(PTE_ADDR(*p)),4096);
   ```

- This is of course useless as it is.

New address space and mapping

- Creating a new address space:

```
pgdir = setupkvm();
```

- Allocate and copy:

```
dst = kalloc();
p = walkpgdir(myproc()->pgdir,0,0);
memmove(dst,p2v(PTE_ADDR(*p)),4096);
```

- Adding translation rule:

```
mappages(pgdir, 0,4096,v2p(dst), (*p) & 4095);
```

Replcating means doing the copy and translation for each page.

Replicating currnet process pages

NO ERROR CHECKING IN HERE!

```
pgdir = setupkvm();

for (va=0; va<myproc()->sz; va += PGSIZE) {
 kva = kalloc();
 memmov(kva, va, PGSIZE);
 pte = walkpgdir(myproc()->pgdir,va,0);
 mappages(pgdir,va,PGSIZE,v2p(kva),(*pte) & 4095);
}
```

- If there is allocation error, all previous allocations must be freed!
- We show freeing on the next slide.

Freeing address space

```
for (i=0; i < 512; i++) {
 if ((pgdir[i] & PTE_P) == 0)
  continue;
 pgtbl = p2v(pgdir[i] & -4096);
 for (j=0; j < 1024; j++) {
  if (pgtbl[j] & PTE_P) {
   kfree(p2v(pgtbl[j] & ~4095));
  }
 }
 kfree(pgtbl);
}
kfree(pgdir);
```

xv6 replicating and freeing address space

The following xv6 code replicates arbitrary address space.

- The code checks for allocation errors.
- It deallocates all previous allocation in case of failure.

copyuvm()

```
2035  pde_t* copyuvm(pde_t *pgdir, uint sz) {
       pde_t *d; pte_t *pte;
       uint pa, i;
       char *mem;
       if ((d = setupkvm()) == 0)  return 0;
       for (i = 0; i < sz; i += PGSIZE) {
        if ((pte = walkpgdir(pgdir, (void *) i, 0)) == 0) panic
        if (!(*pte & PTE_P)) panic("copyuvm: page not present")
        pa = PTE_ADDR(*pte);
        if ((mem = kalloc()) == 0)  goto bad;
        memmove(mem, (char*)p2v(pa), PGSIZE);
        if (mappages(d, (void*)i, PGSIZE, v2p(mem),
              PTE_FLAGS(*pte)) < 0) goto bad;
       }
       return d;
      bad:
       freevm(d);
       return 0;
      }
```

freevm()

```
2003 void freevm(pde_t *pgdir) {
  uint i;

  if (pgdir == 0)
    panic("freevm: no pgdir");
  deallocuvm(pgdir, KERNBASE, 0);
  for (i = 0; i < NPDENTRIES; i++) {
  if (pgdir[i] & PTE_P) {
    char * v = p2v(PTE_ADDR(pgdir[i]));
    kfree(v);
    }
  }
  kfree((char*)pgdir);
}
```

deallocuvm

```
1961 deallocuvm(pde_t *pgdir, uint oldsz, uint newsz) {
  pte_t *pte;
  uint a, pa;
  if (newsz >= oldsz)  return oldsz;
  a = PGROUNDUP(newsz);
  for (; a < oldsz; a += PGSIZE) {
   pte = walkpgdir(pgdir, (char*)a, 0);
   if (!pte) a += (NPTENTRIES - 1) * PGSIZE;
   else if ((*pte & PTE_P) != 0) {
    pa = PTE_ADDR(*pte);
    if (pa == 0) panic("kfree");
    char *v = p2v(pa);
    kfree(v);
    *pte = 0;
   }
  }
  return newsz;
```

fork

- `allocproc`:
  - An `EMBRYO` `proc` struct is constructed.
  - A kernel stack is allocated.
  - An uninitialized `trapframe` is allocated.
  - An artificial `context` is constructed.
- The user space memory of the caller is replicated.
- A new matching page table is constructed.
- The `trapframe` of the caller is copied to the uninitialized `trapframe`.
- The `eax` field of the new `trapframe` is cleared.
- File pointers are replicated.
- Rest of the caller **proc** struct fields are copied to the new **proc** struct.

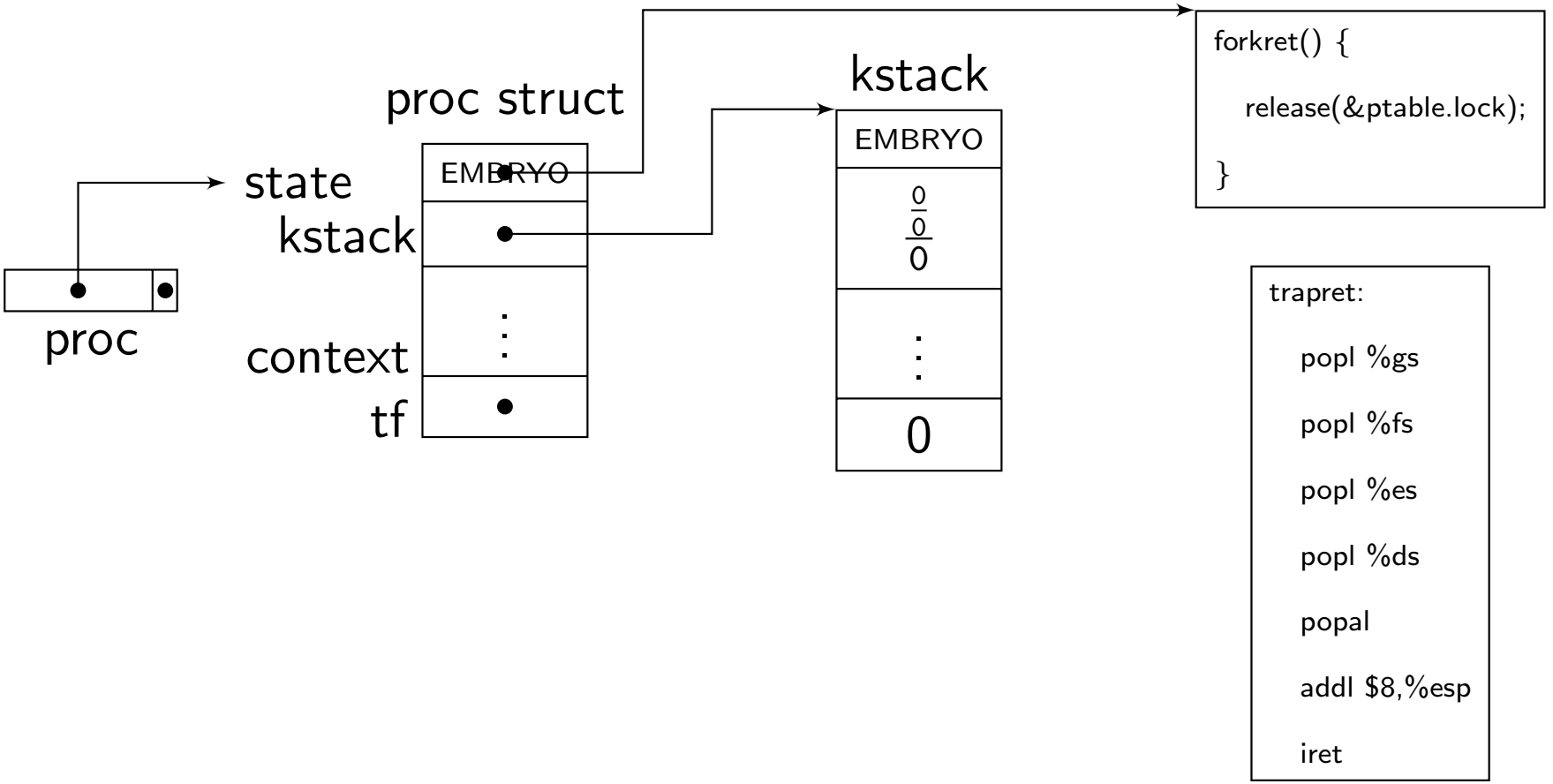
fork (1)

```
2580  int fork(void) {
       int i, pid;
       struct proc *np;

       if ((np = allocproc()) == 0)
        return -1;

       if ((np->pgdir=copyuvm(myproc()->pgdir,myproc()->sz
                        == 0) {
        kfree(np->kstack);
        np->kstack = 0;
        np->state = UNUSED;
        return -1;
       }
```

fork (1) operation



fork (2)

```
  np->sz = myproc()->sz;
  np->parent = myproc();
  *np->tf = *myproc()->tf;

  np->tf->eax = 0;

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

  safestrcpy(np->name, myproc()->name, sizeof(myproc()
  pid = np->pid;
  np->state = RUNNABLE;
  return pid;
}
```