xv6©-rev10
(Copyright Frans Kaashoek, Robert Morris, and Russ Cox.)

The **fork** system call

Carmi Merimovich

Tel-Aviv Academic College

November 30, 2017

sys_fork

```
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

0

Copy of parent

psz

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

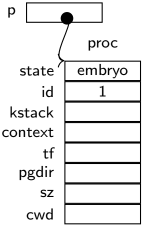
- proc struct and friends.
- Filling `pgdir` requires considerable code replication.
- Replicatin ofile and cwd requires help from the relevant modules.

proc struct and friends

allocproc(): (1) Finding unused `proc` structure

```c
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;

found:
 p->state = EMBRYO;
 p->pid = nextpid++;
 release(&ptable.lock);
```

allocproc(): (1) Operation



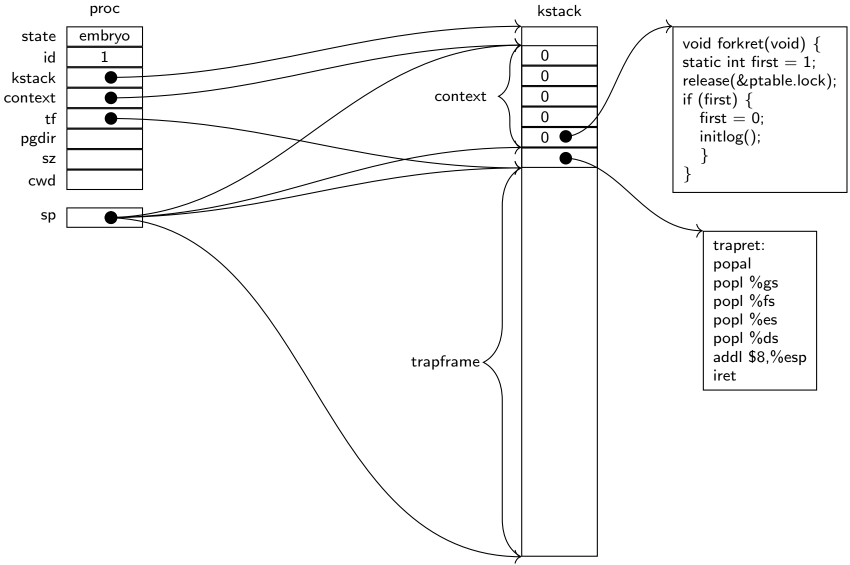

```
void forkret(void) {
static int first = 1;
release(&ptable.lock);
if (first) {
   first = 0;
   initlog();
   }
}
```

```
trapret:
popal
popl %gs
popl %fs
popl %es
popl %ds
addl $8,%esp
iret
```

allocproc: (2) Initialize process kernel stack

```c
  if ((p->kstack = kalloc()) == 0) {
   p->state = UNUSED;
   return 0;
  }
  sp     = p->kstack + KSTACKSIZE;
  sp    -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;
  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;
}
```

allocproc: (2) Operation



User space replication

(two methods)

Replcation page 0 of current process only

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.

Create new address space

Replcation page 0 of current process only

Add mapping rules in the new address space to the replication

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.

Replicate and Map ALL user space pages

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);
}
for (i = 512; i < 1023; i++) {
 if ((pgdir[i] & PTE_P) == 0)
  continue;
 pgtbl = p2v(pgdir[i] & -4096);
 kfree(pgtbl);
}
kfree(pgdir);
```

xv6 code for user space replication

(More general than needed)

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()

```
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()

```c
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

```
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)

```
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 (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;
}
```

How and when the child runs?!

Recall the scheduler

scheduler

```
2758 void scheduler(void) {
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;
  for(;;) { sti();
   acquire(&ptable.lock);
   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if (p->state != RUNNABLE) continue;

    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;
    swtch(&c->scheduler, p->context);
    switchkvm();
    c->proc = 0;
   }
   release(&ptable.lock);
  }
}
```

switchuvm

If switching to user mode is expected:

- The `tr` register should contain the index of a TSS descriptor.
- The `TSS` descriptor should point to a `taskstate` structure.
- The `ss0` and `esp0` fields should point to a valid kernel stack top.
- The above is ESSENTIAL for proper interrupt service in user mode.

switchuvm

```
void switchuvm(struct proc *p) {
  pushcli();
  mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A,
                          &mycpu()->ts,
                          sizeof(mycpu()->ts)-1, 0);
  mycpu()->gdt[SEG_TSS].s = 0;
  mycpu()->ts.ss0 = SEG_KDATA<<3;
  mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
  mycpu()->ts.iomb = (ushort) 0xFFFF;
  ltr(SEG_TSS << 3);
  if (p->pgdir == 0)
    panic("switchuvm: no pgdir");
  lcr3(v2p(p->pgdir)); // switch to new address space
  popcli();
}
```

taskstate (hardware structure)

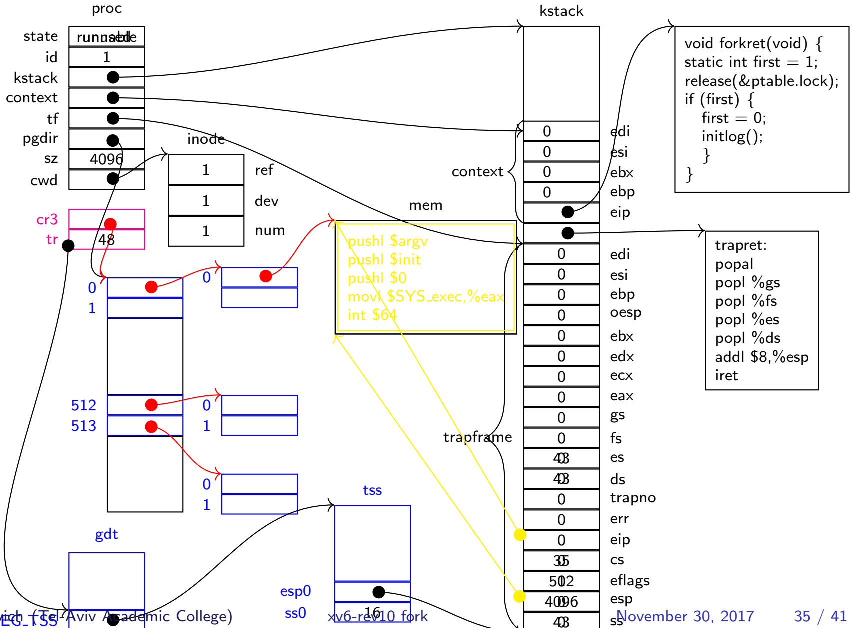
link	
esp0	
	ss0
esp1	
	ss1
esp2	
	ss2
cr3	
eip	
eflags	
eax	
ecx	
edx	
ebx	
esp	
ebp	
esi	
edi	

	es
	cs
	ss
	ds
	fs
	gs
	ldt
	t
iomb	

taskstate in C

```c
struct taskstate {
  uint link;
  uint esp0;
  ushort ss0;
  ushort padding1;
  uint *esp1;
  ushort ss1;
  ushort padding2;
  uint *esp2;
  ushort ss2;
  ushort padding3;
  void *cr3;
  uint *eip;
  uint eflags;
  uint eax;
  uint ecx;
  uint edx;
  uint ebx;
  uint *esp;
  uint *ebp;
  uint esi;
  uint edi;
  ushort es;
  ushort padding4;
  ushort cs;
  ushort padding5;
  ushort ss;
  ushort padding6;
  ushort ds;
  ushort padding7;
  ushort fs;
  ushort padding8;
  ushort gs;
  ushort padding9;
  ushort ldt;
  ushort padding10;
  ushort t;
  ushort iomb;
};
```

switchuvm



proc
state
id
1
kstack
context
tf
pgdir
sz
4096
cwd
cr3
tr
48
inode
1
ref
1
dev
1
num
kstack
context
0
edi
0
esi
0
ebx
0
ebp
eip
void forkret(void) {
static int first = 1;
release(&ptable.lock);
if (first) {
first = 0;
initlog();
}
}
trapret:
popal
popl %gs
popl %fs
popl %es
popl %ds
addl $8,%esp
iret
mem
pushl $argv
pushl $init
pushl $0
movl $SYS_exec,%eax
int $64
0
edi
0
esi
0
ebp
0
oesp
0
ebx
0
edx
0
ecx
0
eax
0
gs
0
fs
es
ds
0
trapno
0
err
0
eip
cs
eflags
esp
ss
trapframe
0
1
512
513
0
1
tss
esp0
ss0
16
gdt
SEG_TSS

swtch

co-routines

- The `scheduler` switches to a process by using:

```
swtch(&c->scheduler, p->context);
```

- A process leaves the cpu by returning to the `scheduler` using:

```
swtch(&p->context, mycpu()->scheduler);
```

- We have here co-routines.

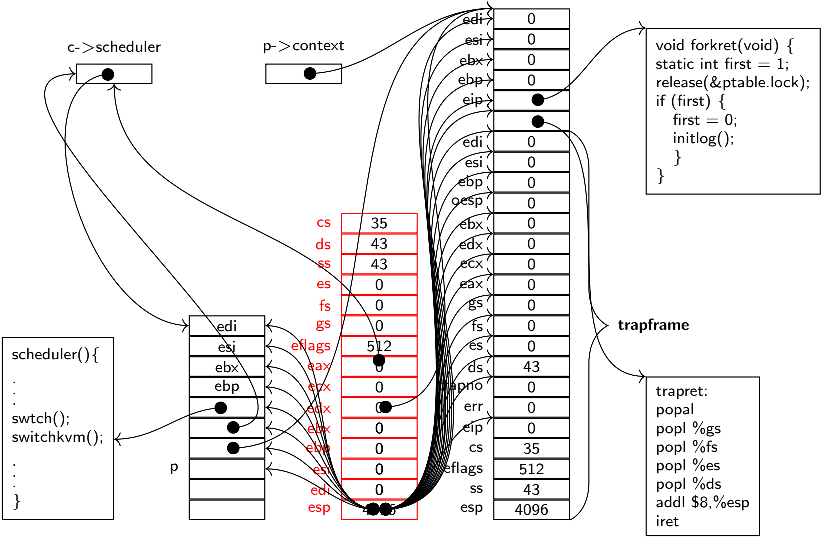
swtch()

```
.globl swtch
swtch:
 movl 4(%esp), %eax
 movl 8(%esp), %edx

 pushl %ebp
 pushl %ebx
 pushl %esi
 pushl %edi

 movl %esp, (%eax)
 movl %edx, %esp
```

```
 popl %edi
 popl %esi
 popl %ebx
 popl %ebp
 ret
```

The `eip` field of `context` is generated by the instruction calling `swtch`.

swtch() operation



Context switch

- Calling **swtch()**:
  - Creates a **context** structure on the current stack.
  - Stores the **cotext** structure address created in the first argument.
  - Load the **context** structure pointed to by the second argument.
- We are switching KERNEL contexts.
- User mode context of a process is loaded by the kernel side of the process.

Kernel context seems too small

- Where are **eax**, **ecx**, **edx**????
  - The **gcc** calling conventions deals with them!
- Where are **cs**, **ds**, and **ss**????
  - The base and limit fields are identical across all kernel sides.
  - So, they need to be loaded only on aech kernel entering.
- Where is **gdtr**????
  - The address and size fields are different across processors.
  - The base and limit MUST NOT change between kernel sides on the same CPU.
  - Since **gdtr** is privileged, it needs to be loaded ONLY on kernel initialization.