LECTURE 10: PROCESS CREATION IN XV6

Process Creation- pg 23-26 of xv6Book

New process creation in xv6

- *init* process: first process created by xv6 after boot up
  - This *init* process forks shell process, which in turn forks other processes to run user commands
  - The *init* process is the ancestor of all processes in Unix-like systems
- After *init*, every other process is created by the fork system call, where a parent forks/spawns a child process
- The function "*allocproc*" called during both *init* process creation and in *fork* system call

(i) gives, structure, PID

(ii) sets up kernel stack.

  - Allocates new process structure, PID etc
  - Sets up the kernel stack of process so that it is ready to be context switched in by scheduler

allocproc

- Find unused entry in ptable, mark is as embryo
  - Marked as runnable after process creation completes
- New PID allocated
- New memory allocated for kernel stack
- Go to bottom of stack, leave space for trapframe
- Push return address of “trapret”
- Push context structure, with eip pointing to function “forkret”
- Why? When this new process is scheduled, it begins execution at forkret, then returns to trapret, then returns from trap to userspace
- Allocproc has created a hand-crafted kernel stack to make the process look like it had a trap and was context switched out in the past
  - Scheduler can switch this process in like any other

```
// Look in the process table for an UNUSED proc.
// If found, change state to EMBRYO and initialize
// state required to run in the kernel.
// Otherwise return 0.
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);

  // 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;
}
```

Init process creation

- Alloc proc has created new process
  - When scheduled, it runs function forkret, then trapret
- Trapframe of process set to make process return to first instruction of init code (initcode.S) in userspace
- The code “initcode.S” simply performs “exec” system call to run the init program

```
// Set up first user process.
void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  p = allocproc();

  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  // this assignment to p->state lets other cores
  // run this process. the acquire forces the above
  // writes to be visible, and the lock is also needed
  // because the assignment might not be atomic.
  acquire(&ptable.lock);

  p->state = RUNNABLE;

  release(&ptable.lock);
}
```

Init process

- Init program opens STDIN, STDOUT, STDERR files
  - Inherited by all subsequent processes as child inherits parent's files
- Forks a child, execs shell executable in the child, waits for child to die
- Reaps dead children (its own or other orphan descendants)

```c
// init: The initial user-level program

#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

char *argv[] = { "sh", 0 };

int
main(void)
{
  int pid, wpid;

  if(open("console", O_RDWR) < 0){
    mknod("console", 1, 1);
    open("console", O_RDWR);
  }
  dup(0);  // stdout
  dup(0);  // stderr

  for(;;){
    printf(1, "init: starting sh\n");
    pid = fork();
    if(pid < 0){
      printf(1, "init: fork failed\n");
      exit();
    }
    if(pid == 0){
      exec("sh", argv);
      printf(1, "init: exec sh failed\n");
      exit();
    }
    while((wpid=wait()) >= 0 && wpid != pid)
      printf(1, "zombie!\n");
  }
}
```

Forking new process

- Fork allocates new process via allocproc
- Parent memory and file descriptors copied (more later)
- Trapframe of child copied from that of parent
  - Result: child returns from trap to exact line of code as parent
  - Different physical memory but same virtual address (location in code)
  - Only return value in eax is changed, so parent and child have different return values from fork
- State of new child set to runnable, so scheduler thread will context switch to child process sometime in future
- Parent returns normally from trap/system call, child runs later when scheduled

```
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;
}
```

Summary of new process creation

- New process created by marking a new entry in ptable as RUNNABLE, after configuring the kernel stack, memory image etc of new process
- Neat hack: kernel stack of new process made tolook like that of a process that had been context switched out in the past, so that scheduler can context switch it in like any other process
  - No special treatment for newly forked process during “swtch”

Thank You