Process management in xv6

Mythili Vutukuru

CSE, IIT Bombay

PCB in xv6: struct proc

```
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)
};

```

struct proc: page table

- Every instruction or data item in the memory image of process (code/data, stack, heap, etc.) has an address
  - Virtual addresses, starting from 0
  - Actual physical addresses in memory can be different (all processes cannot store their first instruction at address 0)
- Page table of a process maintains a mapping between the virtual addresses and physical addresses
- Page table used to translate virtual addresses to physical addresses

struct proc: kernel stack

- Stack to store CPU context when process jumps to kernel mode from user mode, or when process is context switched out
  - Why separate stack? OS does not trust user stack
  - Separate area of memory in the kernel, not accessible by regular user code
  - Linked from struct proc of a process

struct proc: list of open files

- Array of pointers to open files
  - When user opens a file, a new entry is created in this array, and the index of that entry is passed as a file descriptor to user
  - Subsequent read/write calls on a file use this file descriptor to refer to the file
  - First 3 files (array indices 0,1,2) open by default for every process: standard input, output and error
  - Subsequent files opened by a process will occupy later entries in the array

Process table (ptable) in xv6

- Ptable in xv6 is a fixed-size array of all processes
- Real kernels have dynamic-sized data structures

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

CPU scheduler in xv6

- The OS loops over all runnable processes in ptable, picks one, and sets it running on the CPU

```
// Loop over process table looking for process to run.
acquire(&ptable.lock);
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state != RUNNABLE)
    continue;

  // Switch to chosen process.  It is the process's job
  // to release ptable.lock and then reacquire it
  // before jumping back to us.
  c->proc = p;
  switchuvm(p);
  p->state = RUNNING;
```

xv6 system calls

- In xv6, as in other systems, system calls are made by user library functions
  - User code invokes library function only
- System calls available to user programs are defined in user library header “user.h”
  - Equivalent to C library headers (xv6 doesn’t use standard C library)

```c
struct stat;
struct rtcdate;

// 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 close(int);
int kill(int);
int exec(char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
```

What happens on a system call?

- The user library makes the actual system call to invoke OS code
- NOT a regular function call to OS code as it involves CPU privilege level change
- User library invokes special “trap” instruction called “int” in x86 (see usys.S) to make system call
- The trap (int) instruction causes a jump to kernel code that handles the system call
  - More on trap instruction later

```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

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

xv6: fork system call implementation

```c
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;
}
```

xv6: fork system call explanation

- Parent process invokes fork to create new child
  - Allocates new process in ptable, get new PID for child
  - Variable “np” is pointer to newly allocated struct proc of child
  - Variable “currproc” is pointer to struct proc of parent
  - Copies information (memory, files, size, …) from currproc to np
- Child process set to runnable, scheduler runs it at a later time
- Return value in parent is PID of child
- Return value in child is set to 0 (by changing child’s EAX register)

xv6: exit system call implementation

```
void
exit(void)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;

  if(curproc == initproc)
    panic("init exiting");

  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){
    if(curproc->ofile[fd]){
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(curproc->cwd);
  end_op();
  curproc->cwd = 0;

  acquire(&ptable.lock);
  // Parent might be sleeping in wait().
  wakeup1(curproc->parent);

  // Pass abandoned children to init.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == curproc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }

  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

xv6: exit system call explanation

- Exiting process cleans up some state (e.g., close files)
- Wakes up parent process that may be waiting to reap
- Passes abandoned children (orphans) to init
- Marks itself as zombie and invokes scheduler, never gets scheduled again

xv6: wait system call implementation

```c
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);
  }
}
```

xv6: wait system call explanation

- Search for dead children in process table
- If dead child found, clean up memory of zombie, return its PID
- If no children, return -1, no need to wait
- If children exist but haven’t terminated yet, wait until one dies

xv6: exec system implementation overview

- Copy new executable into memory from disk
- Create new stack, heap
- Copy command line arguments to new stack
- Switch process page table to use new memory image
- Process begins to run new code after system call ends
- Revert back to old memory image in case of any error