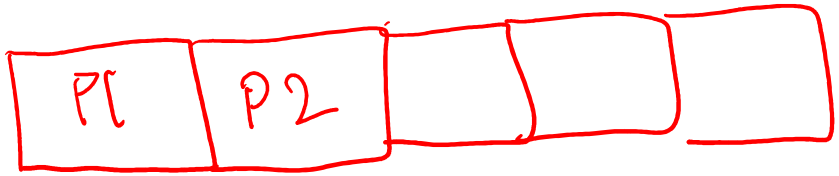
Lecture 22: Processes in xv6

Mythili Vutukuru
IIT Bombay
https://www.cse.iitb.ac.in/~mythili/os/

The process abstraction

- The OS is responsible for concurrently running multiple processes (on one or more CPU cores/processors)
  - Create, run, terminate a process
  - Context switch from one process to another
  - Handle any events (e.g., system calls from process)
- OS maintains all information about an active process in a process control block (PCB)
  - Set of PCBs of all active processes is a critical kernel data structure
  - Maintained as part of kernel memory (part of RAM that stores kernel code and data, more on this later)
- PCB is known by different names in different OS
  - struct proc in xv6
  - task_struct in Linux



PCB in xv6: struct proc

- Page 23, process structure and process states

```
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)
};
```

Why does it points to the bottom of kernel stack.

struct proc: kernel stack

```
char *kstack;                // Bottom of kernel stack for this process
```

- Recall: register state (CPU context) saved on user stack during function calls, to restore/resume later
- Likewise, CPU context stored on kernel stack when process jumps into OS to run kernel code
  - Why separate stack? OS does not trust user stack
  - Separate area of memory per process within the kernel, not accessible by regular user code
  - Linked from struct proc of a process

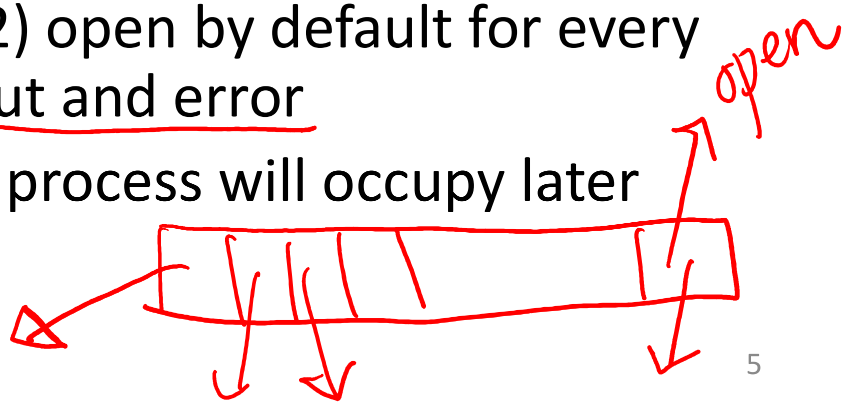
PCB

Kstack

struct proc: list of open files

```
struct file *ofile[NOFILE];  // Open files
```

- Array of pointers to open files (struct file has information about the open file, more on this later)
  - 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



struct proc: page table

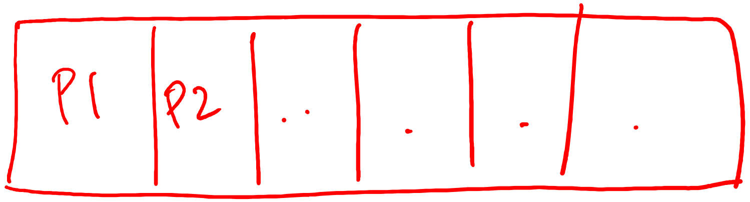
```
2339    pde_t* pgdir;                    // Page table
```

- 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 (more on this later)

Process table (ptable) in xv6

lock

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



- ptable: Fixed-size array of all processes
  - Real kernels have dynamic-sized data structures
- CPU scheduler in the OS loops over all runnable processes, 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;
```

look important.

In xv6, switchuvm(p) is a function that switches the CPU to the user virtual memory of the process p. It is used to set up the memory mapping so that when the process runs, it accesses its own virtual address space instead of the kernel's memory.

Process state transition examples

- A process that needs to sleep (e.g., for disk I/O) will set its state to SLEEPING and invoke scheduler
- A process that has run for its fair share will set itself to RUNNABLE (from RUNNING) and invoke scheduler

  Timer interrupt
- Scheduler will once again find another RUNNABLE process and set it to RUNNING

When timer interrupt occurs.

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
```

This will be invoked in case of input/output type of interruption occurs.

```
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();

  if(p == 0)
    panic("sleep");

  if(lk == 0)
    panic("sleep without lk");

  // Must acquire ptable.lock in order to
  // change p->state and then call sched.
  // Once we hold ptable.lock, we can be
  // guaranteed that we won't miss any wakeup
  // (wakeup runs with ptable.lock locked),
  // so it's okay to release lk.
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  }
  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING;

  sched();
```

call the scheduler.

Summary of xv6 processes

- We have seen basics of PCB structure (struct proc), list of processes (ptable), scheduler code, state transitions switching.
- We will keep revisiting this xv6 code multiple times to understand it better
  - – Each concept will deepen understanding further