Lecture 3: Processes and Threads

Instructor: Mitch Neilsen

Office: N219D

Quote of the Day

"Talent in cheaper than table salt.

What separates the talented individual from the successful one is a lot of hard work."

-- Stephen King

Project 0: Notes

- **`Add BXSHARE environment variable`**
  - export BXSHARE=/<home dir>/cis520/usr/local/share/bochs

- **`$ pintos run alarm-multiple`**
  - with Unity 3d – causes system freeze :-(
  - <ctrl>-alt F1, kill offending processes or reset

- Workarounds:
  - Select Unity 2d on login to lab machine
  - Run command-line version of output:

```
$ pintos -v -- run alarm-multiple
```

Project 0: Notes

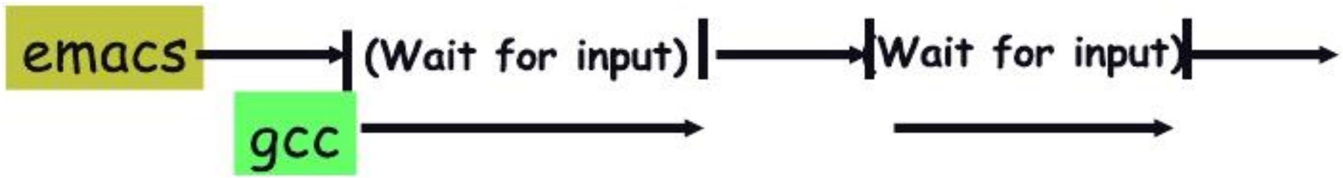
- Goals for this week:
  - Finish installing Pintos
  - Add new test program
    - Hint: first, use **grep -r alarm-multiple *** in the pintos/src/tests folder to see what changes are needed
    - Then, rebuild the operating system back in the pintos/src/threads folder; e.g., **make clean** and **make**

Processes

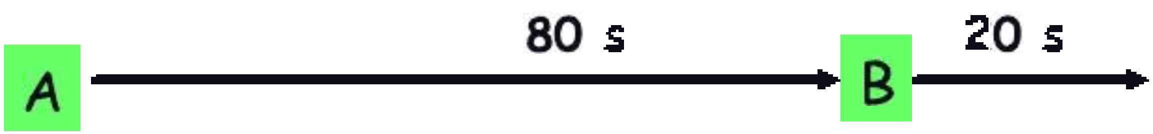
- **A *process* is an instance of a running program**
- **Modern OSes run multiple processes simultaneously**
- **Examples (all can run simultaneously):**
  - gcc fileA.c – compiler running on fileA
  - gcc fileB.c – compiler running on fileB
  - emacs – text editor
  - firefox – web browser
- **Non-examples (implemented as one process):**
  - Multiple firefox tabs or emacs frames (one process, multiple threads)
- **Why processes?**
  - Simplicity of programming
  - Higher throughput (better CPU utilization), lower latency

Speed

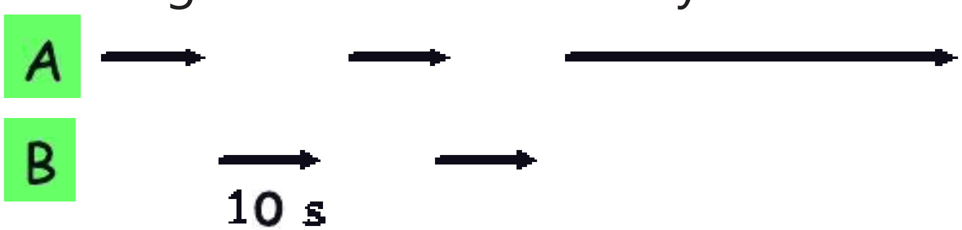
- **Multiple processes can increase CPU utilization**
  - Overlap one process's computation with another 's wait



- **Multiple processes can reduce latency**
  - Running *A* then *B* requires 100 sec for *B* to complete



  - Running *A* and *B* concurrently allows *B* to finish faster



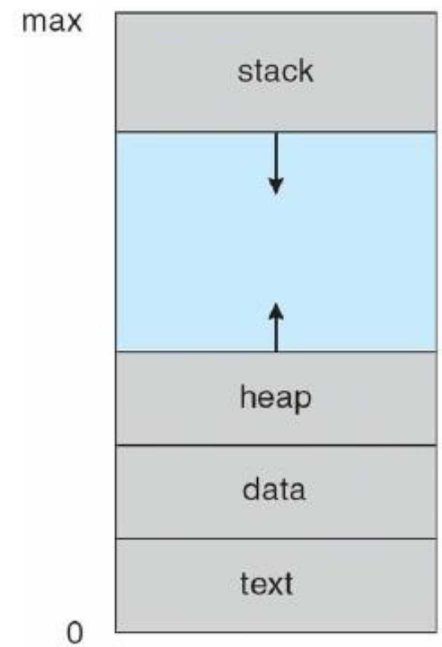
  - *A is* slightly slower, but less than 100 sec unless *A* and *B* are both completely CPU-bound

Processes in the real world

- **Processes, parallelism fact of life much longer than OSes have been around**
  - E.g., say takes 1 worker 10 months to make 1 widget
  - Company may hire 100 workers to make 10,000 widgets
  - Latency for first widget >> 1/10 month
  - Throughput may be < 10 widgets per month
    (if we can't perfectly parallelize tasks)
  - Or > 10 widgets per month if we get better utilization (e.g., 100 workers on 10,000 widgets never idly waiting for paint to dry)
- **You will see this with Pintos**
  - BUT, don't expect labs to take 1/3 time with three people ;-)

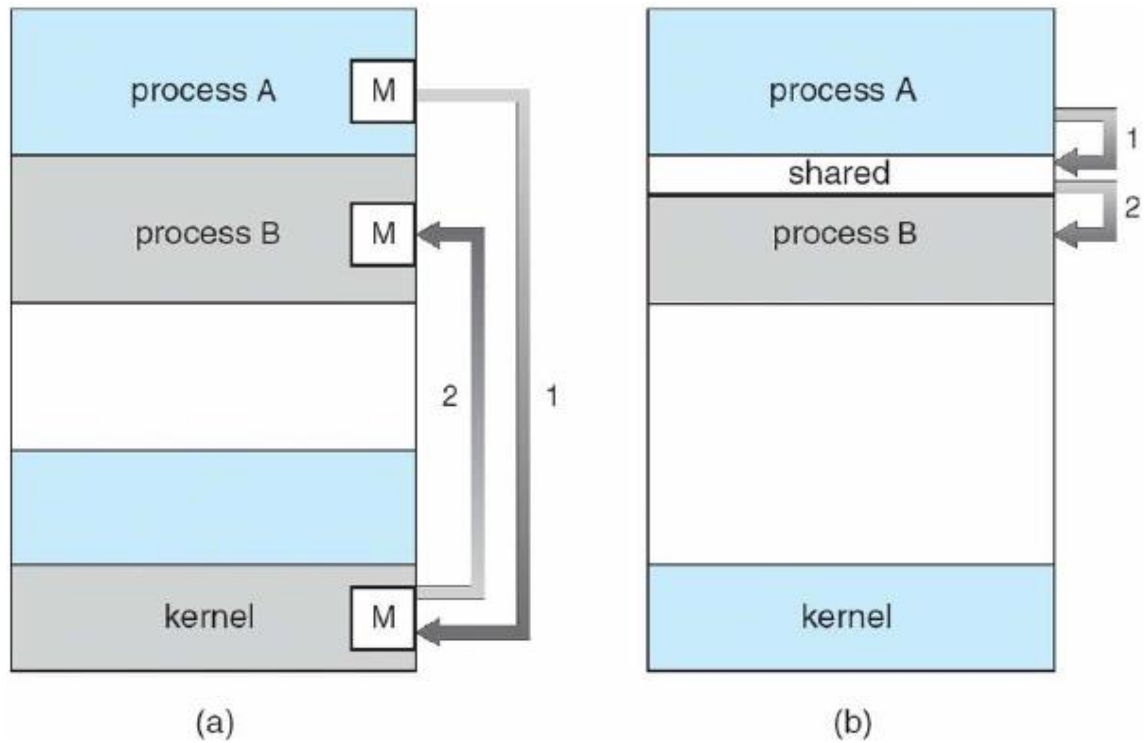
A process's view of the world

- **Each process has its own view of the machine**
  - Its own address space
  - Its own open files
  - Its own virtual CPU (through preemptive multitasking)



- *(char *)0xc000 **different in** P_1 **&** P_2
- **Greatly simplifies programming model**
  - gcc does not care that firefox is running
- **Sometimes want interaction between processes**
  - Simplest is through files: emacs edits file, gcc compiles it
  - More complicated: Shell/command, Window manager/app.

Inter-Process Communication



- **How can processes interact in real time?**
  (a) By passing messages through the kernel
  (b) By sharing a region of physical memory
  (c) Through asynchronous signals or alerts

Rest of lecture

- **User view of processes**
  - Crash course in basic Unix/Linux system call interface
  - How to create, kill, and communicate between processes
- **Kernel view of processes**
  - Implementing processes in the kernel
- **Threads**
- **How to implement threads**

UNIX files I/O

- **Applications "open" files (or devices) by name**
  - I/O happens through open files
- int open(char *path, int flags, /*mode*/...);
  - flags: O_RDONLY, O_WRONLY, O_RDWR
  - O_CREAT: create the file if non-existent
  - O_EXCL: (w/ O_CREAT) create even if file exists already
  - O_TRUNC: Truncate the file to length 0
  - O_APPEND: Start writing from end of file
  - mode: final argument with O_CREAT – set r,w,x permissions
- **Returns file descriptor—used for all I/O to file**

Error returns

- **What if** open **fails? Returns -1 (invalid fd)**
- **Most system calls return -1 on failure**
  - Specific kind of error stored in global int **errno**
- #include <sys/errno.h> **for possible values**
  - errno = 2 = ENOENT “No such file or directory”
  - errno = 13 = EACCES “Permission Denied”
- perror **function prints human-readable message**
  - perror ("initfile");

    → “initfile: No such file or directory”

Operations on file descriptors

- int read (int fd, void *buf, int nbytes);
  - Returns number of bytes read
  - Returns 0 bytes at end of file, or -1 on error
- int write (int fd, void *buf, int nbytes);
  - Returns number of bytes written, -1 on error
- off_t lseek (int fd, off t_pos, int whence);
  - whence: 0 – start, 1 – current, 2 – end
    - Returns previous file offset, or -1 on error
- int close (int fd);

File descriptor numbers

- **File descriptors are inherited by processes**
  - When one process spawns another, same fds by default
- **Descriptors 0, 1, and 2 have special meaning**
  - 0 – "standard input" (stdin in ANSI C)
  - 1 – "standard output" (stdout, printf in ANSI C)
  - 2 – "standard error " (stderr, perror in ANSI C)
  - Normally all three attached to terminal
- **Example:** type.c
  - Prints the contents of a file to stdout

type.c

```
void
typefile (char *filename)
{
    int fd, nread;
    char buf[1024];

    fd = open (filename, O_RDONLY);
    if (fd == -1) {
        perror (filename);
        return;
    }

    while ((nread = read (fd, buf, sizeof (buf))) > 0)
        write (1, buf, nread);

    close (fd);
}
```

Creating processes

- int fork (void);
  - Create new process that is exact copy of current one
  - Returns *process ID* of new process in “parent”
  - Returns 0 in “child”
- int waitpid (int pid, int *stat, int opt);
  - pid – process to wait for, or -1 for any
  - stat – will contain exit value, or signal
  - opt – usually 0 or WNOHANG
  - Returns process ID or -1 on error

Deleting processes

- void exit (int status); // modern Linux replace w/ return(..);
  - Current process ceases to exist
  - status shows up in waitpid (shifted)
  - By convention, status of 0 is success, non-zero error
- int kill (int pid, int sig);
  - Sends signal sig to process pid
  - SIGTERM most common value, kills process by default (but application can catch it for “cleanup”)
  - SIGKILL stronger, kills process always, and cannot be ignored

Running programs

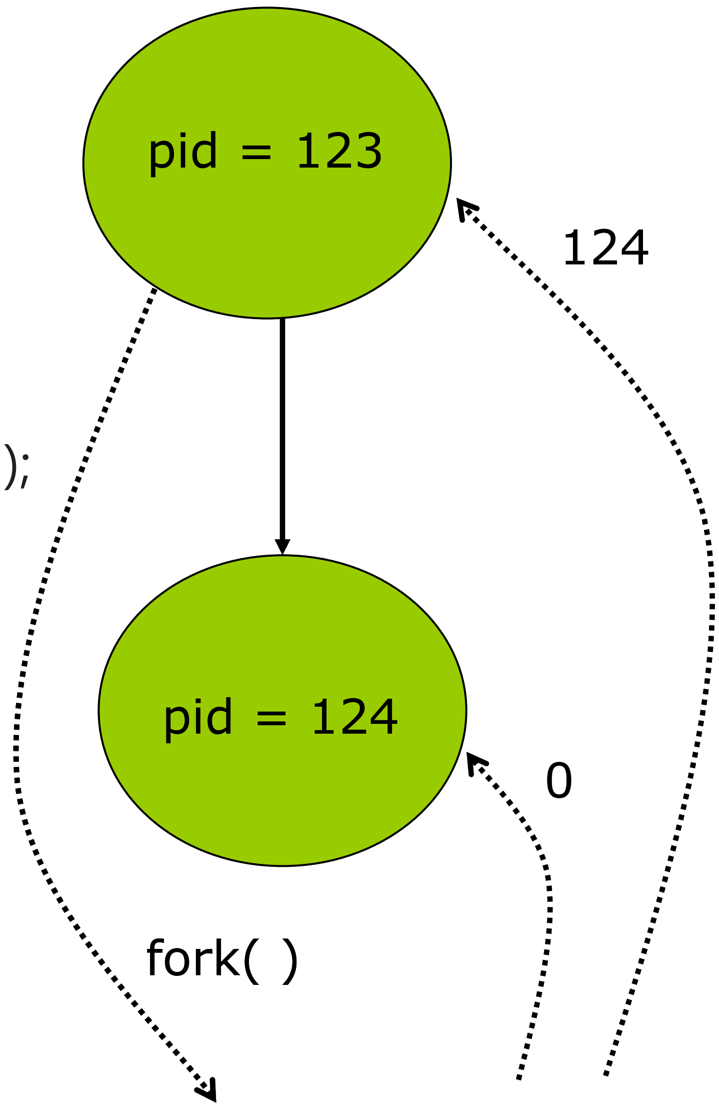
- int execve (char *prog, char **argv, char **envp);
  - prog – full pathname of program to run
  - argv – argument vector that gets passed to main
  - envp – environment variables, e.g., PATH, HOME
- **Generally called through a wrapper functions**
  - int execvp (char *prog, char **argv);
    Search PATH for prog, use current environment
  - int execlp (char *prog, char *arg, ...);
    List arguments one at a time, finish with NULL
- **Example:** minish.c
  - Loop that reads a command, then executes it
- **Warning: Pintos** exec **more like combined fork/exec**

minish.c **(simplified)**

```
pid_t pid; char **av;

void doexec () {
   execvp (av[0], av);
   perror (av[0]);
   exit (1);
}
   /* ... main loop: */
   for (;;) {
      parse_next_line_of_input (av, stdin);

      switch (pid = fork ()) {
      case -1:
         perror ("fork"); break;
      case 0:
         doexec (); break;
      default:
         waitpid (pid, NULL, 0); break;
      }
   }
```

minish.c **(simplified)**

```
pid_t pid; char **av;

void doexec () {
   execvp (av[0], av);
   perror (av[0]);
   exit (1);
}
   /* ... main loop: */
   for (;;) {
      parse_next_line_of_input (av, stdin);

      switch (pid = fork ()) {
      case -1:
         perror ("fork"); break;
      case 0:
         doexec (); break;
      default:
         waitpid (pid, NULL, 0); break;
      }
   }
```



Manipulating file descriptors

- int dup2 (int oldfd, int newfd);
  - Closes newfd, if it was a valid descriptor
  - Makes newfd an exact copy of oldfd
  - Two file descriptors will share same offset (lseek on one will affect both)
- int fcntl (int fd, F_SETFD, int val)
  - Sets *close on exec* flag if val = 1, clears if val = 0
  - Makes file descriptor non-inheritable by spawned programs
- **Example:** redirsh.c
  - Loop that reads a command and executes it
  - Recognizes: $ **command < input > output 2> errlog**

redirsh.c

```
void doexec (void) {
   int fd;

   /* infile non-NULL if user typed "command < infile" */
   if (infile) {
      if ((fd = open (infile, O_RDONLY)) < 0) {
         perror (infile);
         exit (1);
      }
      if (fd != 0) {
         dup2 (fd, 0);
         close (fd);
      }
   }

   /* ... Do same for outfile -> fd 1, errfile -> fd 2 ... */

   execvp (av[0], av);
   perror (av[0]);
   exit (1);
}
```

Pipes

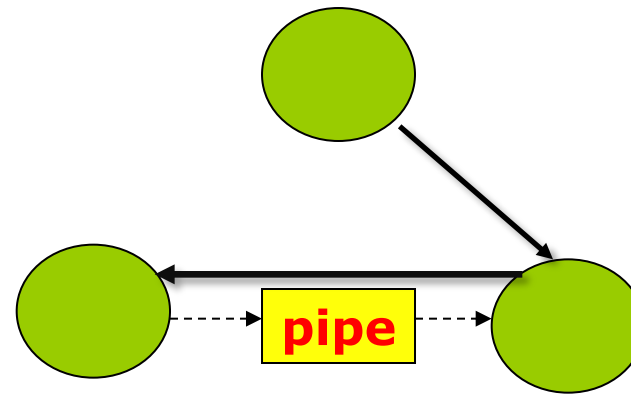
- int pipe (int fds[2]);
  - Returns two file descriptors in fds[0] and fds[1]
  - Writes to fds[1] will be read on fds[0]
  - When last copy of fds[1] closed, fds[0] will return EOF
  - Returns 0 on success, -1 on error
- **Operations on pipes**
  - read/write/close – as with files
  - When fds[1] closed, read(fds[0]) returns 0 bytes
  - When fds[0] closed, write(fds[1]):
    - Kills process with SIGPIPE, or if blocked
    - Fails with EPIPE
- **Example:** pipesh.c
  - Sets up pipeline: $ **command1 | command2 | command3 ...**

pipesh.c **(simplified)**

```
void doexec (void) {
   int pipefds[2];
   while (outcmd) {
      pipe (pipefds);
      switch (fork ()) {
      case -1:
         perror ("fork"); exit (1);
      case 0:
         dup2 (pipefds[1], 1);
         close (pipefds[0]); close (pipefds[1]);
         outcmd = NULL;
         break;
      default:
         dup2 (pipefds[0], 0);
         close (pipefds[0]); close (pipefds[1]);
         parse_command_line (&av, &outcmd, outcmd);
         break;
      }
   }
   /* ... */
}
```

pipesh.c **(simplified)**

```
void doexec (void) {
  int pipefds[2];
  while (outcmd) {
    pipe (pipefds);
    switch (fork ()) {
    case -1:
      perror ("fork"); exit (1);
    case 0:
      dup2 (pipefds[1], 1);
      close (pipefds[0]); close (pipefds[1]);
      outcmd = NULL;
      break;
    default:
      dup2 (pipefds[0], 0);
      close (pipefds[0]); close (pipefds[1]);
      parse_command_line (&av, &outcmd, outcmd);
      break;
    }
  }
} /* ... */
```



Why fork?

- **Most calls to** fork **followed by** execve
- **Could also combine into one *spawn* system call**
  - This is what Pintos exec does
- **Occasionally useful to fork one process**
  - Unix *dump* utility backs up file system to tape
  - If tape fills up, must restart at some logical point
  - Implemented by forking to revert to old state if tape ends
- **Real win is simplicity of interface**
  - Tons of things you might want to do to child:
    Manipulate file descriptors, environment, resource limits, etc.
  - Yet fork requires *no* arguments at all

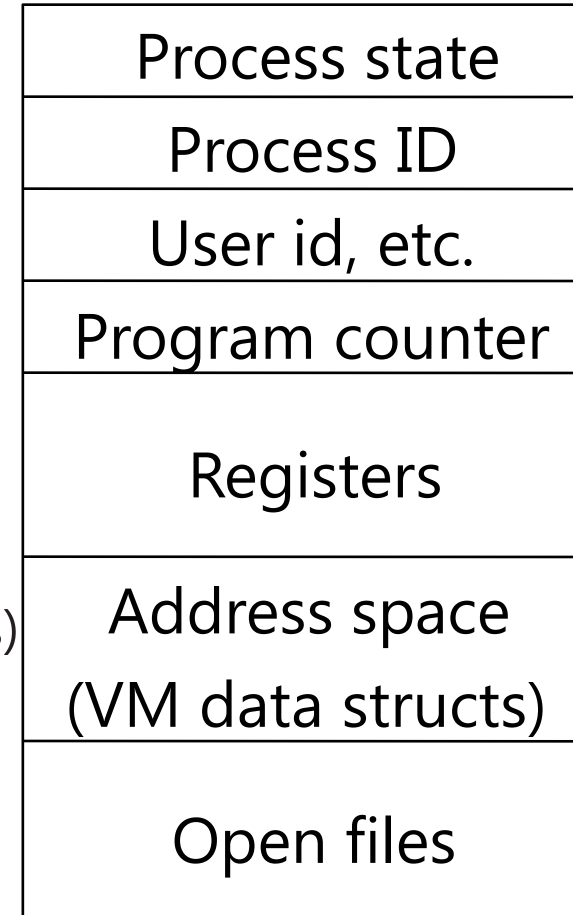
Spawning process w/o fork

- **Without fork, require tons of different options**
- **Example: Windows** CreateProcess **system call**

```
BOOL WINAPI CreateProcess(
__in_opt LPCTSTR lpApplicationName,
__inout_opt LPTSTR lpCommandLine,
__in_opt LPSECURITY_ATTRIBUTES lpProcessAttributes,
__in_opt LPSECURITY_ATTRIBUTES lpThreadAttributes,
__in BOOL bInheritHandles,
__in DWORD dwCreationFlags,
__in_opt LPVOID lpEnvironment,
__in_opt LPCTSTR lpCurrentDirectory,
__in LPSTARTUPINFO lpStartupInfo,
__out LPPROCESS_INFORMATION lpProcessInformation );
```

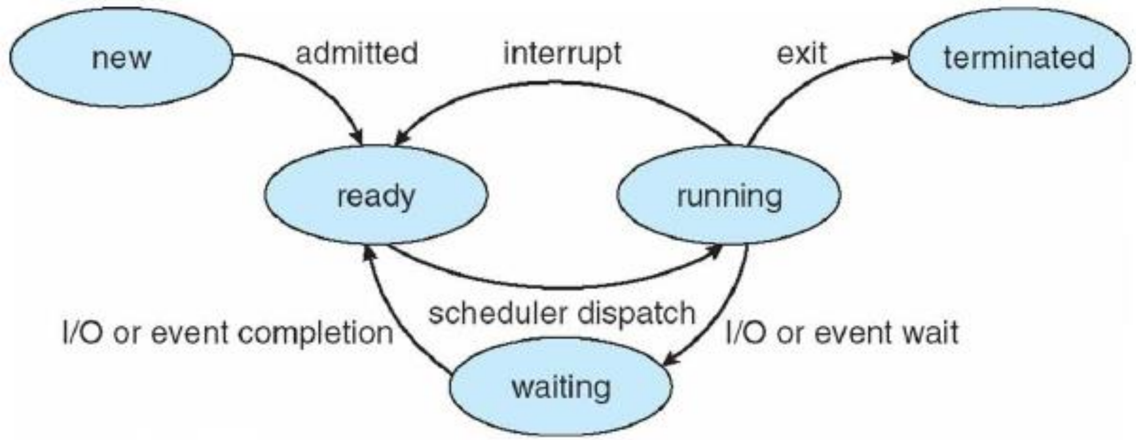
Implementing processes

- **OS keeps data structure for each proc**
  - Process Control Block (PCB)
  - Called proc in Unix, task_struct in Linux
- **Tracks *state* of the process**
  - Running, runnable, blocked, etc.
- **Includes information necessary to run**
  - Registers, virtual memory mappings, etc.
  - Open files (including memory mapped files)
- **Various other data about the process**
  - Credentials (user/group ID), signal mask, controlling terminal, priority, accounting statistics, whether being debugged, which system call binary emulation in use, . . .



PCB

Process states



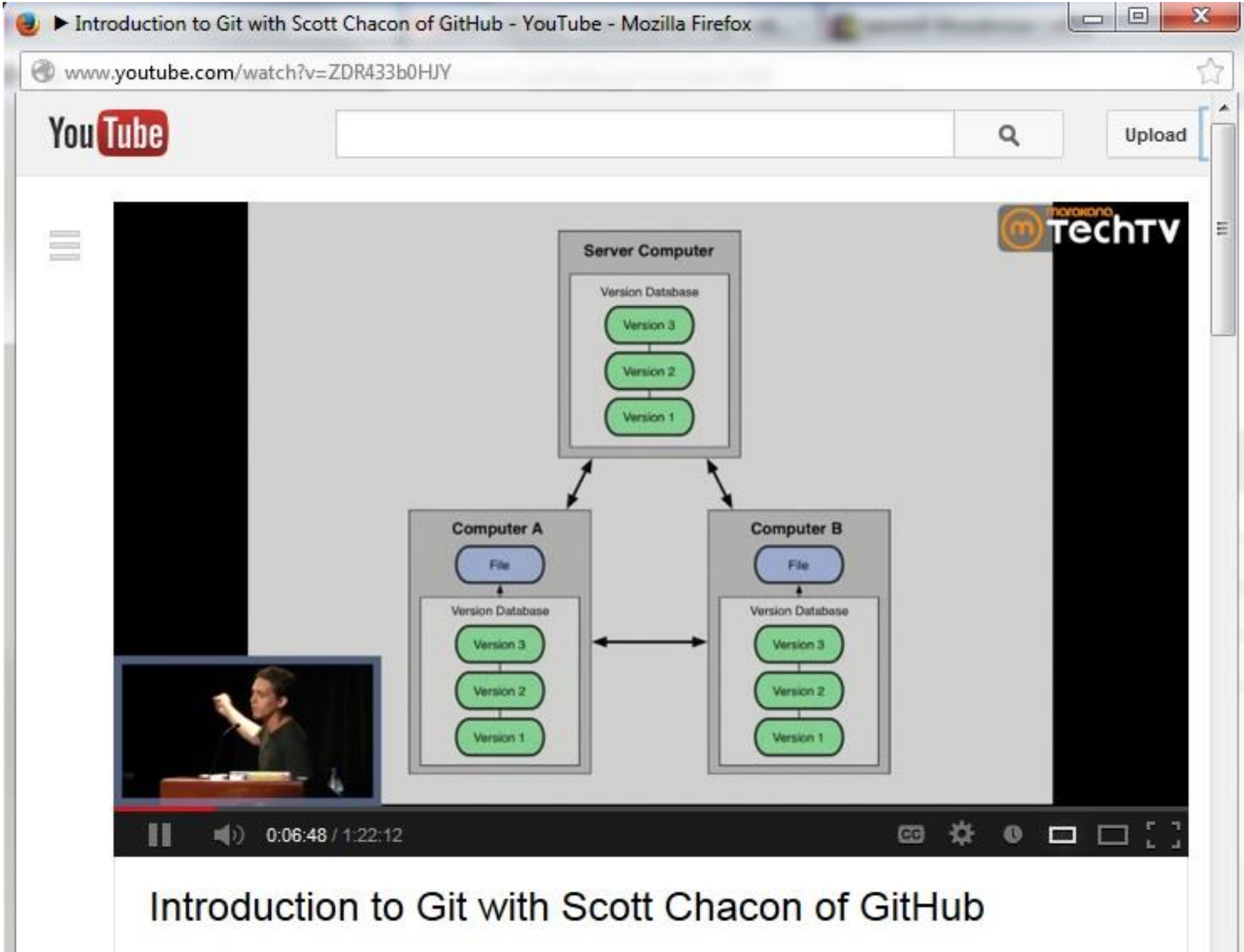
- **Process can be in one of several states**
  - *new* & *terminated* at beginning & end of life
  - *running* – currently executing (or will execute on kernel return)
  - *ready* – can run, but kernel has chosen different process to run
  - *waiting* – needs async event (e.g., disk operation) to proceed
- **Which process should kernel run?**
  - if 0 runnable, run idle loop, if 1 runnable, run it
  - if >1 runnable, must make scheduling decision

Processes

- **A *process* is an instance of a running program**
- **Modern OSes run multiple processes simultaneously**
- **Why processes?**
  - Simplicity of programming
  - Higher throughput (better CPU utilization), lower latency
  - But, relatively expensive to create a new proccess, next time we will turn our attention to scheduling and threads

Summary

- Course web page via K-State OnLine has all lecture notes, assignments, handouts, etc.
- Read Ch. 1-4: Processes and Threads
- Friday: Finish Project 0 and set up base revision of Pintos on version control system; e.g., using subversion, git, etc.
- Watch YouTube Video on Git with Scott Chacon: http://www.youtube.com/watch?v=ZDR433b0HJY



Introduction to Git with Scott Chacon of GitHub - YouTube - Mozilla Firefox
www.youtube.com/watch?v=ZDR433b0HJY
You Tube
Upload
TechTV
Server Computer
Version Database
Version 3
Version 2
Version 1
Computer A
File
Version Database
Version 3
Version 2
Version 1
Computer B
File
Version Database
Version 3
Version 2
Version 1
0:06:48 / 1:22:12
Introduction to Git with Scott Chacon of GitHub