The devil shell (dsh)

COMPSCI210 Recitation

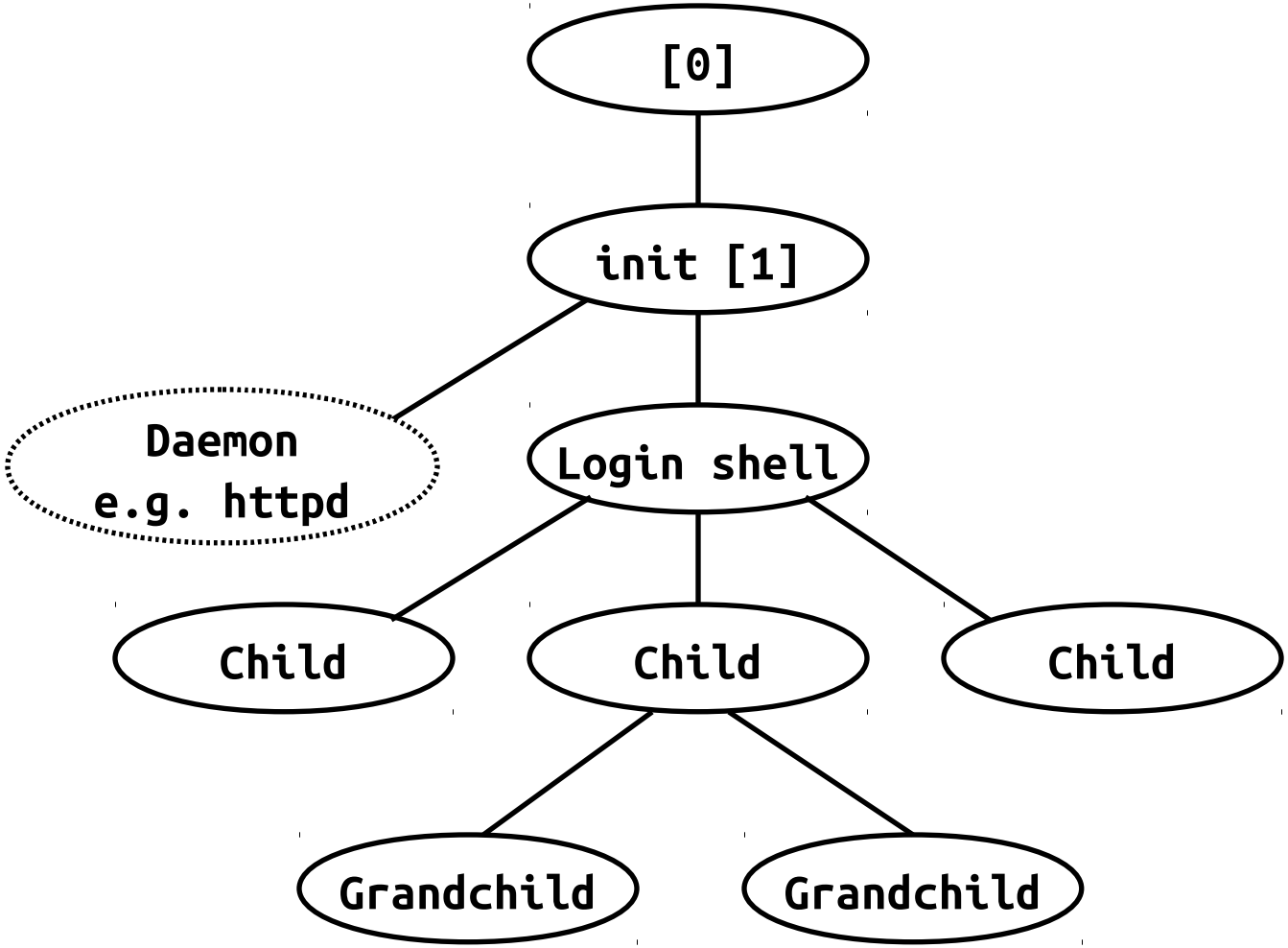
4th Feb 2013

Vamsi Thummala

Shell

- Interactive command interpreter
- A high level language (scripting)
- Interface to the OS
- Provides support for key OS ideas
  - Isolation
  - Concurrency
  - Communication
  - Synchronization

Unix Process Hierarchy



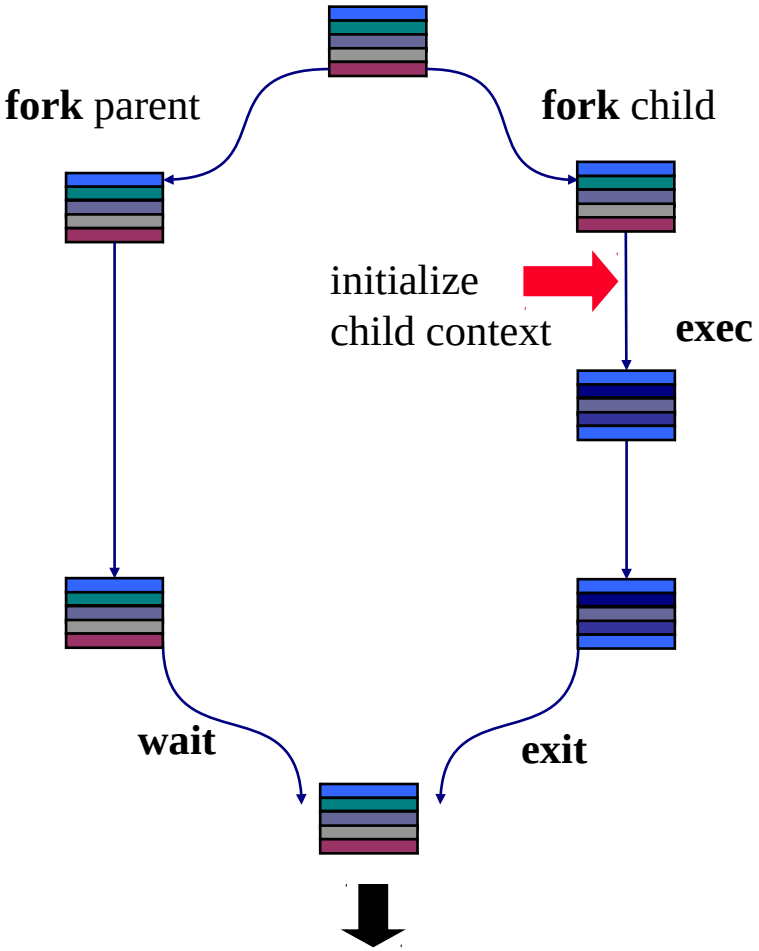
Shell Concepts

- Process creation
- Execution
- Input/Output redirection
- Pipelines
- Job control
  - Process groups
  - Foreground/background jobs
    - Given that many processes can be executed concurrently, which processes should have accesses to the keyboard/screen (I/O)?
  - Signals (limited for the lab!)
    - SIGCONT, SIGTTOU, SIGTTIN
    - Default actions are good enough, no special handling required

dsh: job, process, and cmdline

- Built-in commands
  - fg, bg, jobs, cd, ctrl-d (quit/exit)
- Process == command == an executable file
  - ls, ps, whoami, wc
- Job == at least one process; possibly pipeline of processes
  - ls | sort | wc
- Cmdline == at least one job; possibly sequence of jobs
  - ls | sort | wc; whoami

Unix fork/exec/exit/wait syscalls



int pid = fork();
Create a new process that is a clone of its parent.

exec*("program" [, argvp, envp]);
Overlay the calling process with a new program, and transfer control to it.

exit(status);
Exit with status, destroying the process. Note: this is not the only way for a process to exit!

int pid = wait*(&status);
Wait for exit (or other status change) of a child, and "reap" its exit status. Note: child may have exited before parent calls wait!

Process creation and execution

```
while (1) {
   printf("dsh$ ");
   new_job = readcmdline(args);
   switch (pid = fork()) {                    // new process; concurrency
     case -1:
         perror("Failed to fork\n");
     case 0:                                  // child when pid = 0
         exec (new_job->first_process, args, 0);    // run command
     default:                                 // parent pid > 0
         waitpid(pid, NULL, 0);               // wait until child is done
 }
```

Fork Example

- What is the output of the program?

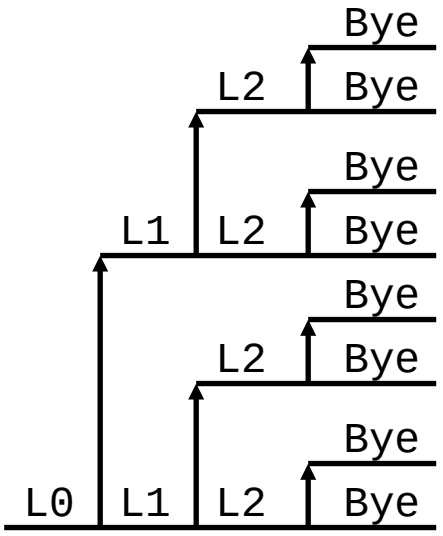
```
void fork_l3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```

- # of L0, L1, L2, Bye?

Fork Example

- What is the output of the program?

```
void fork_l3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



- # of L0, L1, L2, Bye?

Fork with Exec: Example

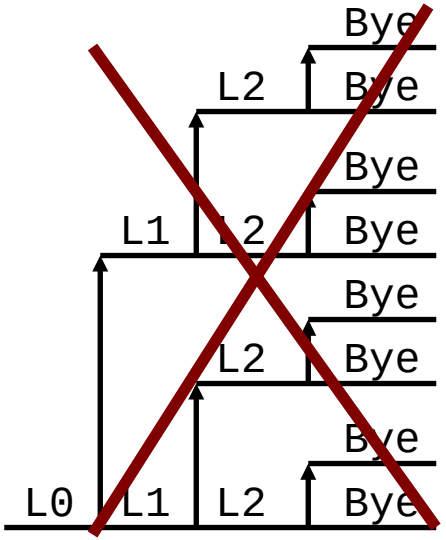
- What is the output of the program?

```
void fork_exec()
{
    printf("L0\n");
    fork();
    char *args[] = { "/bin/echo", NULL };
    if(execve("/bin/echo", args) < 0) {
        perror("execve");
        exit(EXIT_FAILURE);
    }
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```

Fork with Exec: Example

- What is the output of the program?

```
void fork_exec()
{
    printf("L0\n");
    fork();
    char *args[] = { "/bin/echo", NULL };
    if(execve("/bin/echo", args) < 0) {
        perror("execve");
        exit(EXIT_FAILURE);
    }
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



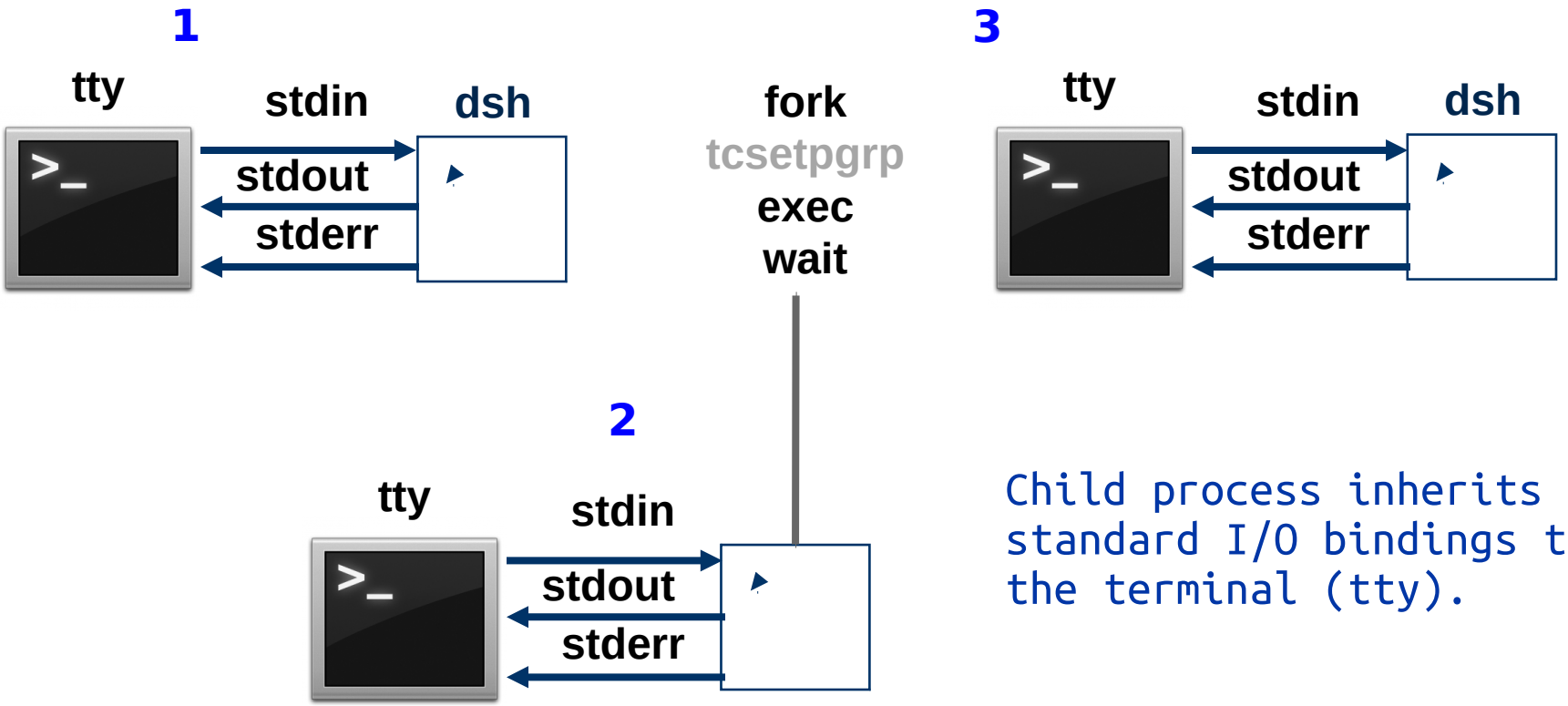
Process groups

- A process group is a collection of (related) processes. Each group has a process group ID.
- Process groups are useful for signal handling
- There is at most one foreground process group which controls the tty
- Each group has a group leader who pid = pgid
  - To get the group ID of a process:

    pid_t getpgrp(void)
  - A process may join an existing group, create a new group.

    int setpgid(pid_t, pid, pid_t, pgid)
  - A signal can be sent to the whole group of processes.

Shell and child: bg, fg, jobs



Child process inherits standard I/O bindings to the terminal (tty).

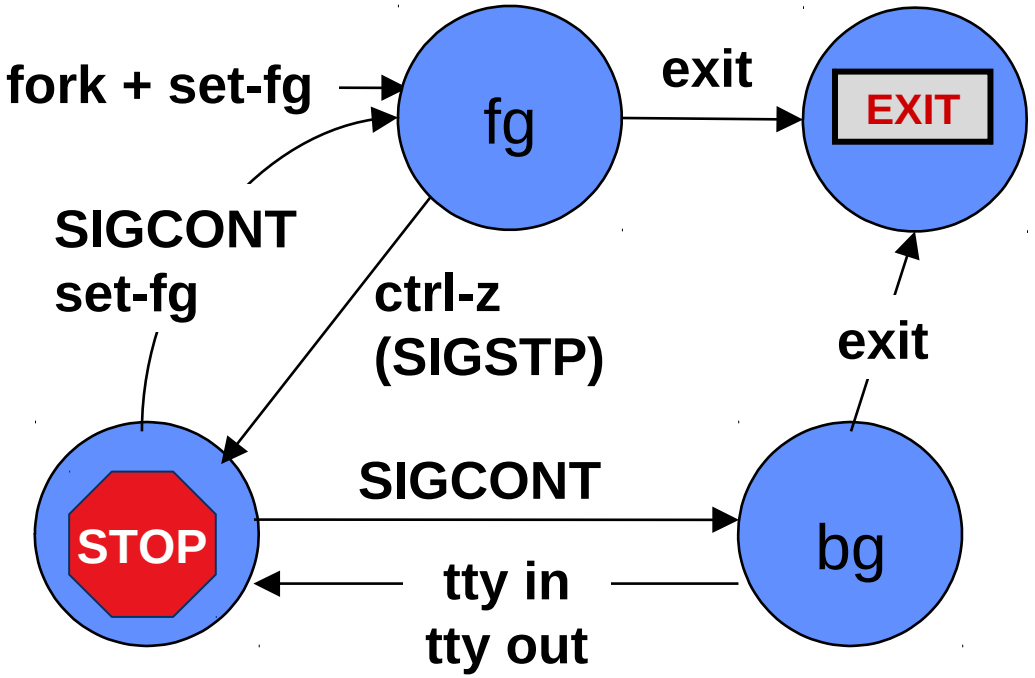
If child is to run in the **foreground**:
Child takes control of the terminal (tty) input (tcsetpgrp).
The foreground process receives all tty input until it stops or exits.
At most one process can control the tty input (others may write to tty).

```
void spawn_job(job_t *j, bool fg) {

      pid_t pid;
      process_t *p;

      for(p = j->first_process; p; p=p->next) { // Loop through the process
           switch (pid = fork()) {
                case -1: /* fork failure */
                          perror("fork"); exit(1);
                case 0: /* child */
                         /* establish a new process group
                          * Q: what if setpgid fails?
                          */
                          if (j->pgid < 0)
                              j->pgid = getpid();

                          if (setpgid(0,j->pgid) == 0 && fg) // If success and fg is set
                              tcsetpgrp(STDIN_FILENO, j->pgid); // assign the terminal

                         /* Exec code here */
                default: /* parent */
                          /* establish child process group here too. */
                          if (j->pgid < 0)
                               j->pgid = pid;
                          setpgid(pid, j->pgid);
           }
           waitpid(WAIT_ANY, &status, WNOHANG | WUNTRACED) /* wait_() for jobs to complete */
           tcsetpgrp(STDIN_FILENO, getpid()); /* grab control of the terminal */
      }
}
```

Job states and transitions

User can send a STOP signal to a foreground process/job by typing **ctrl-z** on tty.

Continue a stopped process by sending it a SIGCONT signal with "**kill***" syscall.



Kernel (tty driver) sends signal to process **P** if **P** attempts to read from tty and p is in background, and (optionally) if **P** attempts to write to tty. Default action of these signals is **STOP**.

Resuming a job

```
/* Sends SIGCONT signal to wake up the blocked job */
void continue_job(job_t *j)
{
     if(kill(-(j->pgid), SIGCONT) < 0)
          perror("kill(SIGCONT)");
}
```

Another oddity of Unix: kill + negative sign

Interpretation: SIGCONT signal to a process group

Input/Output (I/O)

- I/O through file descriptors
  - File descriptor may be for a file, terminal, …
- Example calls
  - `read(fd, buf, sizeof(buf));`
  - `write(fd, buf, sizeof(buf));`
  - `write(STDOUT_FILENO, buf, sizeof(buf)); // writing to stdout`
    - Avoid printf()
- Convention:
  - 0: input
  - 1: output
  - 2: error
- Child inherits open file descriptors from parents
  - Files, pipes, and sockets are external to process and can be shared

I/O redirection (< >)

- Example: “ls > tmpFile”
- Modify *dsh* to insert before exec:

```
close(1);                              // release fd 1
fd = creat(“tmpFile”, 0644);           // fd will be 1
          // or
fd = open(“tmpFile”, O_WRONLY | O_CREAT | O_APPEND,
(S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) // 0644
```

- No modifications to “ls”!
- “ls” could be writing to file, terminal, etc., but programmer of “ls” doesn’t need to know

Pipeline: Chaining processes

- One-way communication channel
- Symbol: |

```
int fdarray[2]; char buffer[100];
pipe(fdarray);
write(fdarray[1], “hello world”, 11);
read(fdarray[0], buffer, sizeof(buffer));
printf(“Received string: %s\n”, buffer);
```

Pipe between parent/child

```
int fdarray[2];
char buffer[100];
pipe(fdarray);
    switch (pid = fork()) {
          case -1: perror("fork failed"); exit(1);
          case 0:  write(fdarray[1], "hello world", 5);
          default: n = read(fdarray[0], buffer,  sizeof(buffer));
                   //block until data is available
    }
```

How does the pipes work in shell, i.e, "ls | wc"?

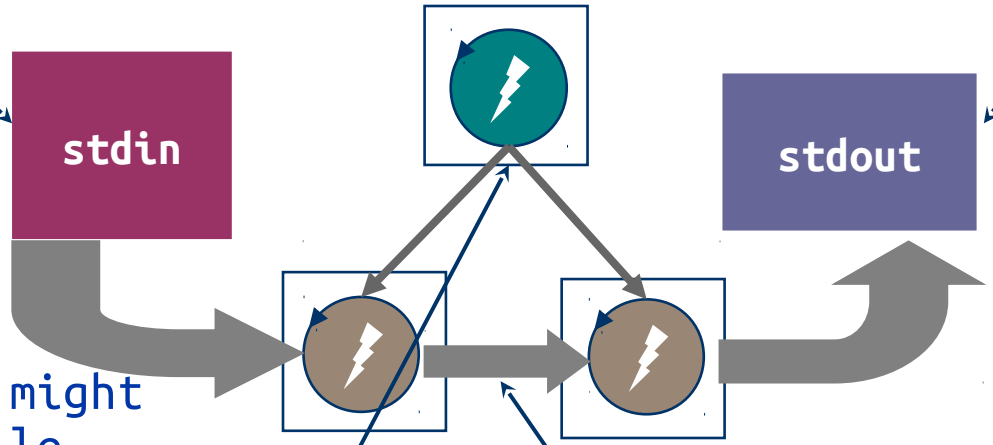
Need to duplicate the child descriptors to stdin/stdout

```
dup2(oldfd, newfd); // duplicates fd; closes and copies at one shot
```

Pipes are core to Unix programming environment

Standard unix programs read a byte stream from standard input (fd==0).

They write their output to standard output (fd==1).



Stdin or stdout might be bound to a file, pipe, device, or network socket.

If the parent sets it up, the program doesn't even have to know.

That style makes it easy to combine simple programs using pipes or files.

dsh additional requirements

- Auto compilation and execution of C programs
- Error handling and Logging
- Batch mode