The devil shell (dsh) – Continued
Shell and child: bg, fg, jobs

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).

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

- **R**: Running or runnable (on run queue)
- **D**: Uninterruptible sleep (waiting for some event)
- **S**: Interruptible sleep (waiting for some event or signal)
- **T**: Stopped, either by a job control signal or because it is being traced by a debugger
- **Z**: Zombie process, terminated but not yet reaped by its parent

- **s** This process is a session leader.
- **+** This process is part of a foreground process group.
Process States: Unix shell

- `ps j` or `ps -l` or `ps -jl`

vamsi@COMPSCI210$ ps j | cat | sort -n

<table>
<thead>
<tr>
<th>PPID</th>
<th>PID</th>
<th>PGID</th>
<th>SID</th>
<th>TTY</th>
<th>TPGID</th>
<th>STAT</th>
<th>UID</th>
<th>TIME</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>2021</td>
<td>2146</td>
<td>2146</td>
<td>2146</td>
<td>pts/0</td>
<td>24837</td>
<td>Ss</td>
<td>1000</td>
<td>0:01</td>
<td>bash</td>
</tr>
<tr>
<td>2146</td>
<td>24808</td>
<td>24808</td>
<td>2146</td>
<td>pts/0</td>
<td>24808</td>
<td>R+</td>
<td>1000</td>
<td>0:00</td>
<td>ps j</td>
</tr>
<tr>
<td>2146</td>
<td>24809</td>
<td>24808</td>
<td>2146</td>
<td>pts/0</td>
<td>24808</td>
<td>S+</td>
<td>1000</td>
<td>0:00</td>
<td>cat</td>
</tr>
<tr>
<td>2146</td>
<td>24810</td>
<td>24808</td>
<td>2146</td>
<td>pts/0</td>
<td>24808</td>
<td>S+</td>
<td>1000</td>
<td>0:00</td>
<td>sort</td>
</tr>
</tbody>
</table>

vamsi@COMPSCI210$ jobs

[1]+ Stopped                 vim
[2] Running                 sleep 50 &

vamsi@COMPSCI210$ ps -l | cat | sort -k3 -n

<table>
<thead>
<tr>
<th>F S</th>
<th>UID</th>
<th>PID</th>
<th>PPID</th>
<th>PRI</th>
<th>NI</th>
<th>ADDR</th>
<th>SZ</th>
<th>WCHAN</th>
<th>TTY</th>
<th>TIME</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 R</td>
<td>1000</td>
<td>25023</td>
<td>2146</td>
<td>0</td>
<td>80</td>
<td>0 -</td>
<td>1177</td>
<td>-</td>
<td>pts/0</td>
<td>00:00:00</td>
<td>ps</td>
</tr>
<tr>
<td>0 S</td>
<td>1000</td>
<td>2146</td>
<td>2021</td>
<td>0</td>
<td>80</td>
<td>0 -</td>
<td>2180</td>
<td>wait</td>
<td>pts/0</td>
<td>00:00:01</td>
<td>bash</td>
</tr>
<tr>
<td>0 S</td>
<td>1000</td>
<td>25021</td>
<td>2146</td>
<td>0</td>
<td>80</td>
<td>0 -</td>
<td>1051</td>
<td>hrtim</td>
<td>pts/0</td>
<td>00:00:00</td>
<td>sleep</td>
</tr>
<tr>
<td>0 S</td>
<td>1000</td>
<td>25024</td>
<td>2146</td>
<td>0</td>
<td>80</td>
<td>0 -</td>
<td>1057</td>
<td>pipe_w</td>
<td>pts/0</td>
<td>00:00:00</td>
<td>cat</td>
</tr>
<tr>
<td>0 S</td>
<td>1000</td>
<td>25025</td>
<td>2146</td>
<td>0</td>
<td>80</td>
<td>0 -</td>
<td>2154</td>
<td>pipe_w</td>
<td>pts/0</td>
<td>00:00:00</td>
<td>sort</td>
</tr>
<tr>
<td>0 T</td>
<td>1000</td>
<td>25012</td>
<td>2146</td>
<td>0</td>
<td>80</td>
<td>0 -</td>
<td>3163</td>
<td>signal</td>
<td>pts/0</td>
<td>00:00:00</td>
<td>vim</td>
</tr>
</tbody>
</table>
Pipeline: Chaining processes

• One-way communication channel
• Symbol: |

```c
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);
```
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.
Pipeline implementation

- **dsh**
  - pid=501
  - pgid=502

- **exec()**
  - e.g., `ls`
  - pid=502
  - pgid=502

- **exec()**
  - e.g., `wc`
  - pid=503
  - pgid=502

- **fork**

- **wait()**

Chaining:
- `dup2(fd[0], STDIN_FILENO)`
dsh additional requirements

- Auto compilation and execution of C programs
  - How to execute two processes sequentially?
- Error handling and logging
  - dup2(stderr, ...)
- Batch mode
  - $./dsh < batchFile
  - Batch mode is used for partial grading
  - It is important that you should test in batch mode before submission
IPC: Beyond pipes

• Named pipes
  
dsh$ mkfifo namedPipe
  
dsh$ cat < namedPipe > out &
  
dsh$ jobs
  
[1]+ Running cat < namedPipe > out &
  
dsh$ echo "Communicating to other process via name pipe" >
  
namedPipe
  
dsh$ cat out
  
Communicating to other process via name pipe

• Sockets
  
  − Named bidirectional pipe
  
  − To the kernel, an endpoint of communication
  
  − Can be used to communicate across a network
  
  − Underlying basis for all Internet applications
Client Server communication

1. Client sends request

Client process $\rightarrow$ Server process $\rightarrow$ Resource

2. Server handles request

3. Server sends response

4. Client handles response

- Create a socket with the `socket()` system call
- Bind the socket to an address using the `bind()` system call. For a server socket on the Internet, an address consists of a port number on the host machine.
- Listen for connections with the `listen()` system call
- Accept a connection with the `accept()` system call. This call typically blocks until a client connects with the server.
- Send and receive data using the `read()` and `write()` system calls
Client Server communication: A detailed example

Connection socket pair
(128.2.194.242:51213, 208.216.181.15:80)

Client socket address
128.2.194.242:51213

Server socket address
208.216.181.15:80

Client host address
128.2.194.242

Server host address
208.216.181.15

51213 is an ephemeral port allocated by the kernel

80 is a well-known port associated with Web servers

[Demo]
Socket interface

Client

open_clientfd

Socket

bind

listen

Connection request

accept

Connection request from next client

await connection

Client/Server Session

rio_writen

rio_readlineb

rio_readlineb

rio_writen

close

EOF

close

[CMU 15-213]
java.net

- Low level API
  - Addresses
  - Sockets
  - Interfaces

- High level API
  - URIs
  - URLs
  - Connections
Concept checkers for midterm

• Basic: address space, process, thread, event
• Kernel: syscall, context switch, and exceptions (trap, fault, interrupt)
• Protection: reference monitor, access control list, capability
• Execution: process vs. thread
• Concurrency: event-driven vs. threading
• Fragmentation: internal vs. external
• IPC: pipes vs. sockets
Fall 2012 midterm paper

• With solutions