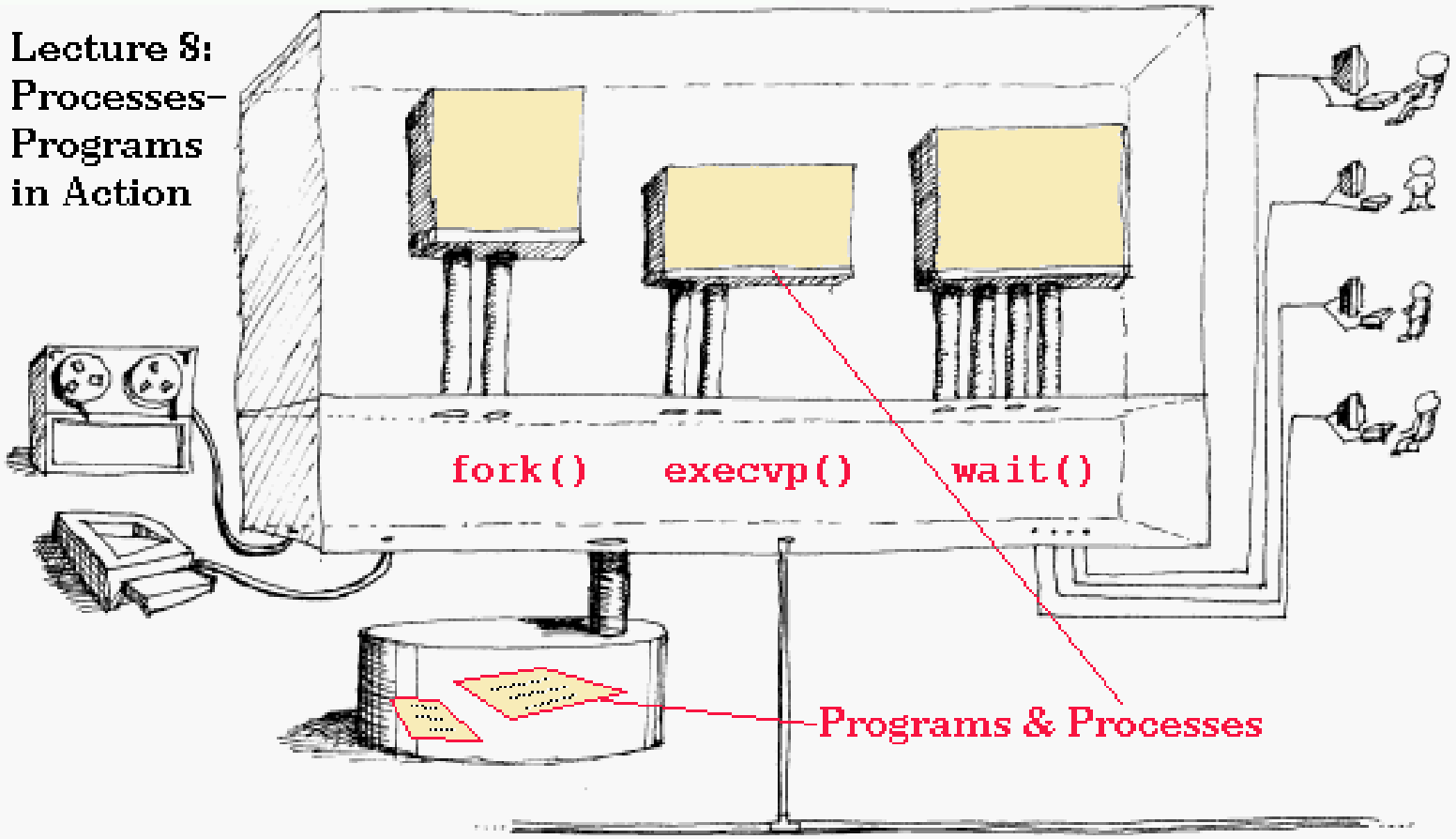
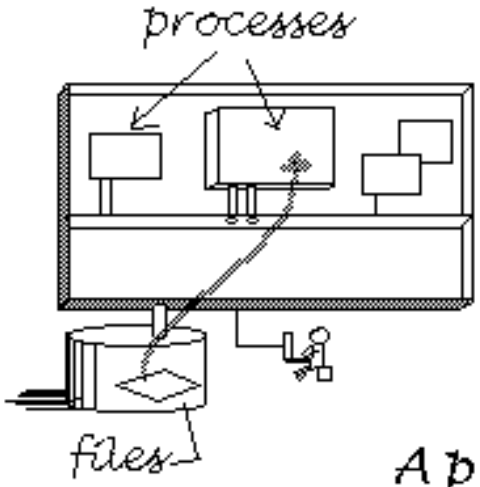
Lecture 8: Processes–Programs in Action



Class 8: Processes = Programs in Action



I Overview: Data + Processing

a. data and programs are STORED in files
b. programs are RUN in processes

A program is RUN (executed) by

1. kernel copies the code into memory
2. CPU executes the instructions

Goals for this section of the course:

a. Understand the Unix model of a process

- the attributes of a process
- the life-cycle of a process
- the capabilities of a process

b. Learn how to program processes

- how to create processes
- how to use processes to run programs
- how to get processes to communicate

Method

experiment with the ps command

write a shell

II Exploring Darkest, Deep User Space using ps

ls just as the ls command lists the objects and attributes in the file system:

```
ls
ls -l
ls -a
...
```

ps the ps command lists the objects and their properties in the process table:

```
ps
ps -l
ps -a
```

lots of options, varies a LOT with Unix version

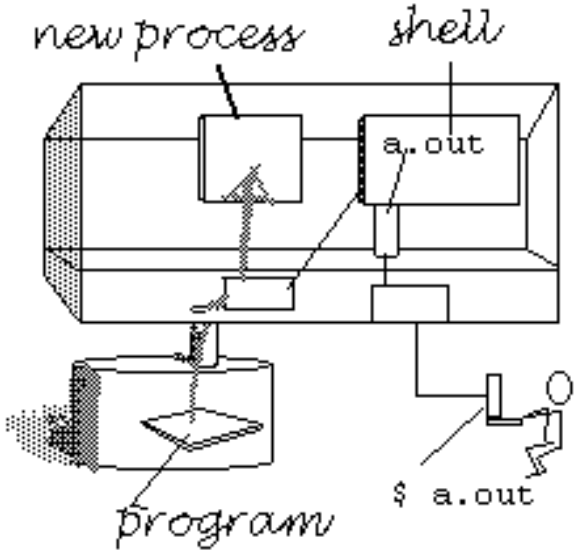
```
$ man ps
```

Attributes of a Process

- user
- tty
- addr
- status
- nice
- ...

III THE SHELL: A software tool for process and program control

A shell is a program designed to manage programs



let's write one!

Running Programs

a. user types a.out
b. shell asks kernel for a new process
c. shell asks kernel to execute the program a.out in that new process
d. the a.out program runs

Manage Input/Output

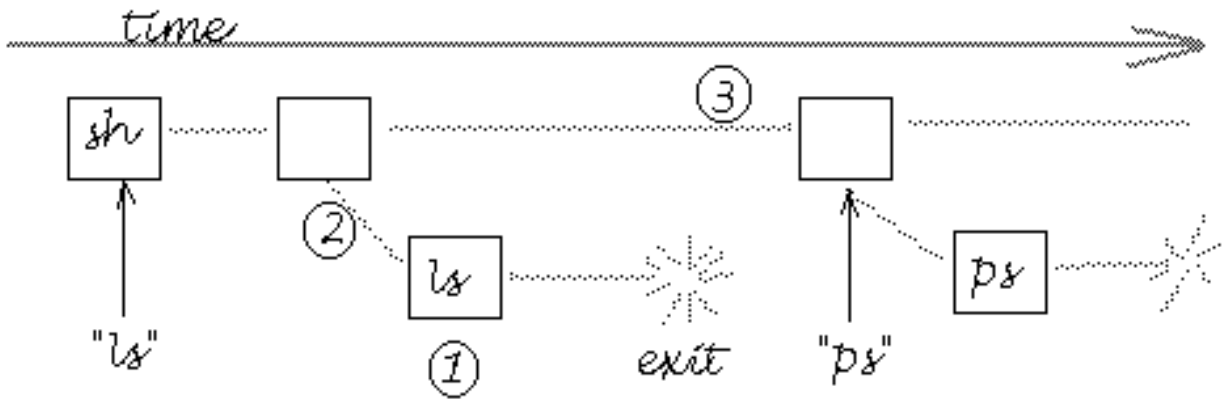
$ dulite > usagelist

Programming

variables
flow control(if, while..)

IV. Writing a Shell: The Basic Loop

get command
execute command
wait for exit



We need to learn how to:

1. Run a program
2. Create a process
3. Wait for exit()

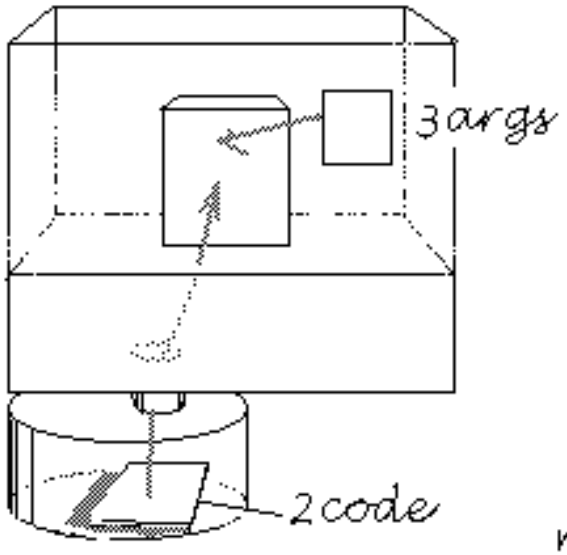
now, read on . . .

Q1: How Can a Program Run a Program ?

Ans: The program calls execvp()

Usage: execvp(progname, arglist)

char * char *[]



1. Program calls execvp
2. Kernel loads program
3. Kernel puts arglist into program
4. Kernel calls main(ac,av)

note: execvp() returns -1 on error. See man page.

ex1: execdemo.c — how to run a program

purpose: run the command ls -l /usr/bin

code:

```
main()
{
  char *args[5] = { "ls", "ls", "-l", "/usr/bin", NULL };
  printf("Before exec\n");
  execvp( "ls", args );
  printf("after exec\n");
}
```

note: The second message does not appear

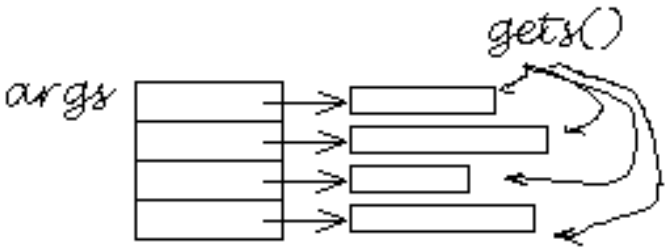
why? exec() loads the program into the current process. The new program REPLACES the current program. It is like a brain transplant. (see prev. picture)

also always runs the same command. Better to have some user input to specify program to run

ex2: psh1.c a 'prompting' shell

purpose: prompt user for a command and its args
then run that program, passing the args

outline:



1) Build arglist
one string at a
time.
(Add NULL to end)

2) Pass args[0] and
list to execvp()

problem: the execvp() works but takes the
process with it

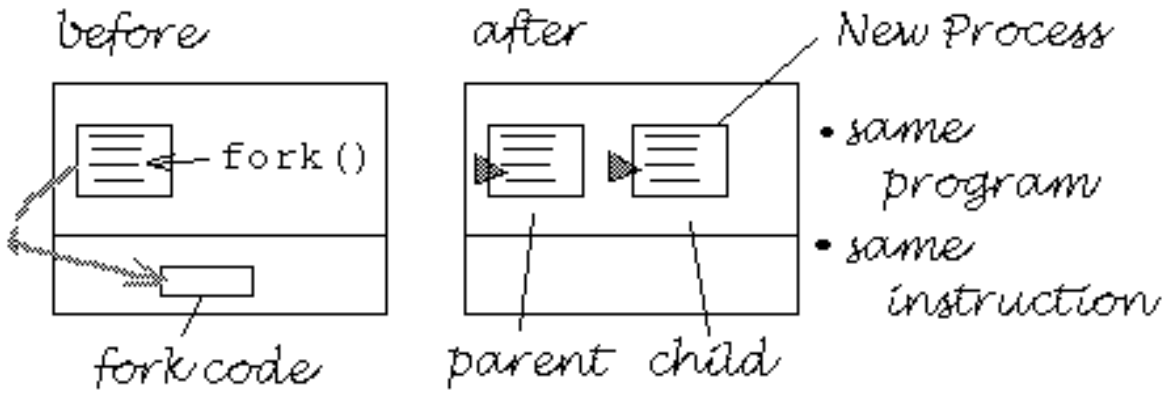
solution: create a new process and have that
one exec the new program

Q2: How Do We Get a New Process to Run the Requested Program?

ans: A process calls fork() to clone itself

usage: fork() // takes no arguments

picture:



returns:

-1 => error
0 in child
pid in parent

kernel allocates a new chunk of memory, copies code and other resources into the new space

fork() examples

1) forkdemo1.c

```
printf( "my pid = %d\n" , getpid() );
n = fork() ;
printf( "my pid = %d, n = %d\n" , getpid() , n);
```

shows two independent processes

2) forkdemo2.c

```
print pid
fork
fork
fork
printpid
```

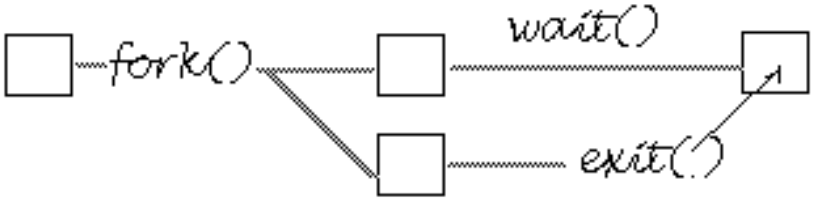
predict outcome of this code

Conclusion: by using fork(), we can create a new process. That new process can call execvp() to run the program.

Q3: What Does the Parent Do While the Child Is exec()-ing the Program?

ans: wait() causes the process to pause until a child process exit()s
and..transfers the arg to exit() from the child to the parent

usage: `p = wait( &from_child );`



returns: -1 if no children
else pauses then returns pid of child

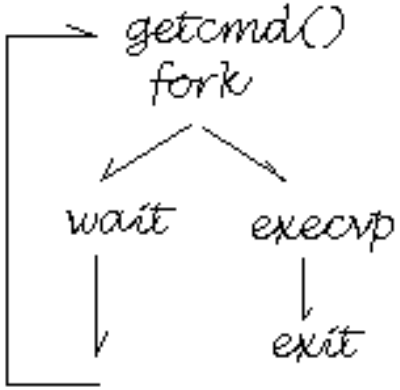
examples of wait()

waitdemo.c

shows how wait works

psh2.c

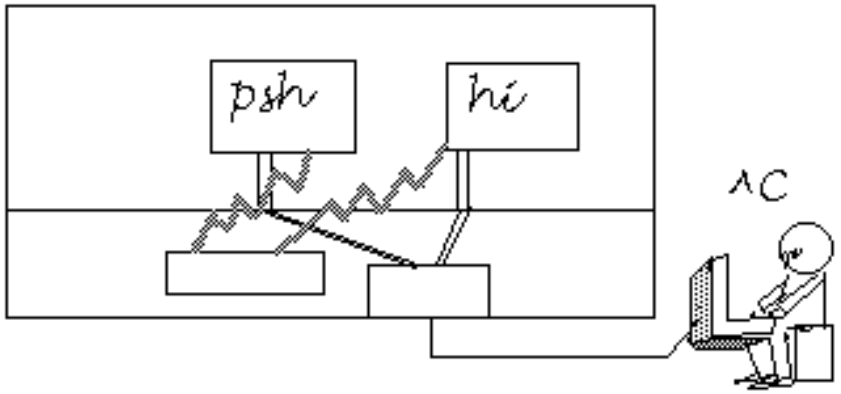
use fork() to create a new process,
in child, use execvp() to run a program
in parent, use wait() to suspend until child exit()s



Cool...but.. what happens if you press ^C in child ?

Q4: What happens when ^C generates a SIGINT ?

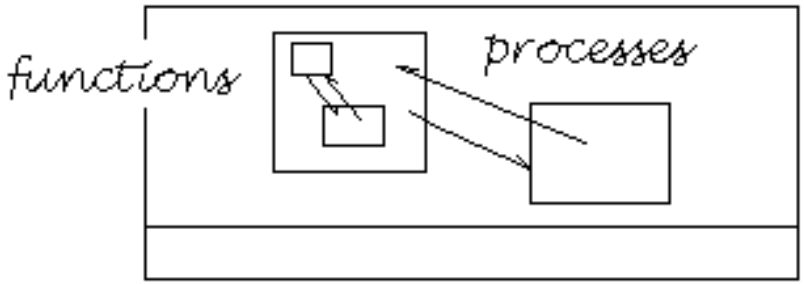
ans: the signal is sent to ALL processes attaches to that tty



question: what can the shell do to avoid getting ^C while it wait()s for the child to exit?

Reflection: execvp and exit are like call and return

call/return A function in a C program calls a function, passes it args. The function does stuff and returns a value.



exec/exit A C program can execvp a program and pass it args.

The program does stuff and can return a value via exit(n). The caller receives the value by calling wait(&n) The return value is stored in bits 8-15 of n.

Another means of communication:

C functions can also pass values via global variables

C programs can pass values via the 'environment'

for details,
tune in next week