Client-Server Lab Part 1 Processes

Outline

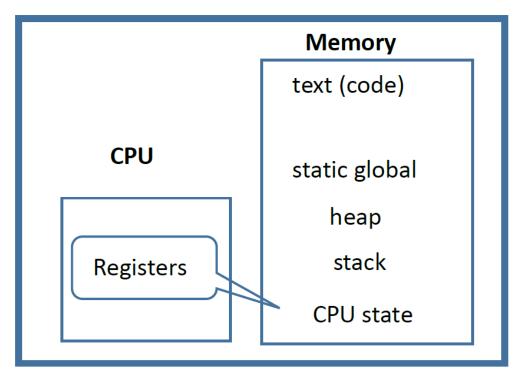
- What is a process
- Managing processes
- Launching programs from a process
- Communicating processes: pipes
- I/O redirection: dup2 ~

What is a Process

Process:

An instance of a program running in a computer





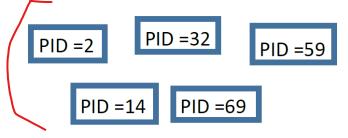
You can have multiple processes running in a computer

Time sharing: CPU running multiple programs Multi-core computers

General Purpose Computers run multiple processes

(e.g., Windows, Android, Apple OS, Linux)

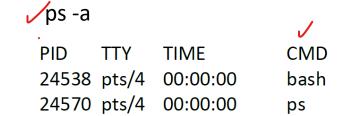
Each process has its own unique Process ID (PID)



Processes are created over time, successive processes getting higher PIDs

Processes run in "parallel" and independenty.

Processes are created/destroyed by the operating system



Operating System: Efficiently run processes.

Managing Processes

Create, terminate, wait for a process to terminate

Create

Parent

```
main() {

pid = fork();

if (pid < 0) { /* error */

   indicate an error occured
}

if (pid== 0) { /* child */

   processing for the child
}

else { /* parent */

   processing for the parent
}
}
```

fork():

- 1 Creates a child process, which is identical to the parent process
- 2 Returns
 - 1 To the parent, the PID of the child /
 - 2 To the child, the value 0

The returned value indicates if the program is a parent or child

See fork.c

Managing Processes: Create, wait, exit

Parent

```
main()
pid = fork()
if (pid < 0) {
 indicate an error occured
if (pid== 0) { /* child */
  processing for the child
processin or the parent
```

Child with PID=387 ✓

The parent and child are running "concurrently", either on different CPU cores or by time sharing or both

Exiting (terminating) a process:

```
vexit(EXIT_FAILURE); -- exiting due to failure
google: exit() Linux

wait and waitpid: blocking
  int status;

/ wait(&status); -- wait until a child terminates
/ waitpd(pid, &status, 0);
  -- wait until the child with pid terminates

See wait.c

for (i=0; i<4; i++) wait();</pre>
```

vexit(EXIT SUCCESS); -- exiting after success

Launching Processes: exec()

```
Programs of the Linux OS:
ps, ls, cat, mv
```

```
aloha.c:
void main()
printf("Aloha world!\n");
```

```
Linux Prompt: cat filename,
                                 e.g., more filename
cat and more are in directory 'bin'
```

Path to the program

The command to launch the program

Terminates the command line

execlp(|input 0| |input 1, input 2, input 3, ..., input 20,||(char *) NULL);

```
#include <stdio.h>
#include <stdlib.h>
 #include <unistd.h> /* Necessary for 'execlp' */
void main()
 printf("Start of program 'launch'\n");
hexeclp("./aloha","./aloha", (char *) NULL);
 printf("End of program 'launch'\n");
```

/bin/cat

execlp("\bin\ls", "ls", "-l", (char*) NULL);

The process is completely replaced by the "aloha" process

Start of program 'launch' Aloha world!

See exec.c

Launching Processes: program launching a program

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
void main()
if (fork()) {
    execlp("./aloha", "./aloha", (char *) NULL);
    exit(EXIT FAILURE);
< Code for main() >
```

Now we have two processes





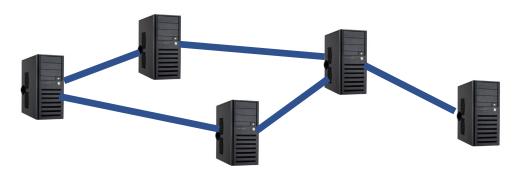
Issue:

✓ Processes can run independently like virtual computers running programs; isolated from each other

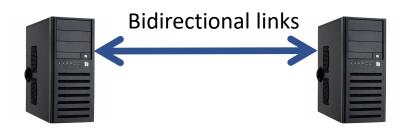


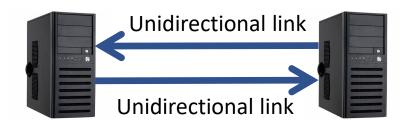
How can we get them to work together?

We can set up communication links



Communication Links





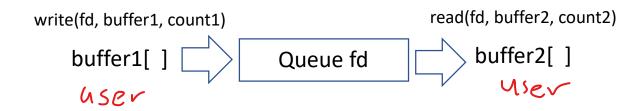
Pipes Shared memory Internet sockets ID of data to queue transfer # bytes to transfer ssize_t write(fd, buffer1, count); Nonblocking

Returns # bytes actually transferred

send

ID of data to queue transfer # bytes to transfer ssize_t read(fd, buffer2, count); Nonblocking Returns # bytes actually transferred

recv



Links: Pipes

write(fd[1], buffer1, count1) read(fd[0], buffer2, count2)
buffer1[] Queue fd buffer2[]
close(fd[1]); close(fd[0]);

You send data from process 1 to process 2:

Create a pipe(fd)

Then create the processes

Process 1 writes to the pipe

Process 2 reads from the pipe

Redirecting I/O: dup2

You send data from process 1 to process 2:

Create a pipe(fd)

Stuom

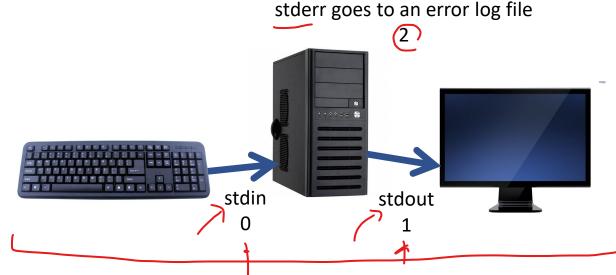
ps -a

Can't insert a write(fd, ,);

Then create the processes, which are launced programs, e.g., ps –a, hexdump

Programs such as ps-a or hexdump will read and write data through stdin and stdout

Queue fd

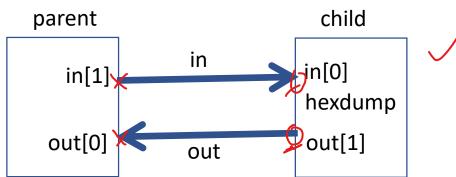


hexdump

Can't insert a read(fd, ,);

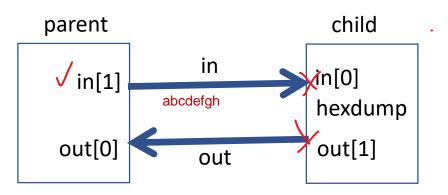
cat filename

Example



```
int in[2], out[2], n, pid;
  char buf[255];
v if (pipe(in) < 0) error("pipe in"); /* Create pipe 'in' */
  if (pipe(out) < 0) error("pipe out"); /* Create pipe 'out' */
  if ((pid=fork()) == 0) { /* Child */
                                                                        Child
    close(0); /* Close stdin, stdout, stderr */
    dup2(in[0],0); /* Redirect stdin, stdout, and stderr to/from pipes */
    dup2 (out[1],1);
    dup2 (out[1],2);
  /close(in[1]); /* Close unused ends of pipes */
  ✓close(out[0]); /* Then when the other end is closed the processes get EOF */
    execl("/usr/bin/hexdump", "hexdump", "-C", (char *) NULL);
    error("Could not exec hexdump");
                                                   hexdump -C <input>
```

Example



```
/* Parent process */
printf("Spawned 'hexdump -C' as a child process at pid %d\n", pid);
close(in[0]); /* Close unused end of pipes */
close(out[1]);
printf("String sent to child: %s\n\n", data);
write(in[1], data, strlen(data)); /* Parent sends string to hexdump */
close(in[1]); /* This will for an EOF to be sent to child */
n = read(out[0], buf, 250); /* Read back any output from hexdump */
buf[n] = 0;
printf("This was received by the child: %s",buf);
exit(0);
```