Processes

- UNIX process creation

  image-file arg1 arg2 ...

- **Shell command line example**

  ls -l

  Equivalent to /bin/ls -l

  Why?

- **How do you find out where the image file is?**
- **Background processes**

  ls -l &

- **Execute a process by a system call**

```
int execlp ( const char *file, const char *arg, ... )
```

image-file name **Arguments as in main() and end with NULL**

Replace the current process image with the specified image file

Return only if it fails

Copyright ©1997

Processes (cont'd)

- Example

```
execlp( “ls”, “ls”, “-l”, NULL );
fprintf( stderr, “Couldn’t execute ‘ls’\n” ); exit(1);
```

- There are many other alternatives to `execlp`:

```
int execl( const char *path, const char *arg, ... );
int execv( const char *path, char * const argv[] );
int execle( const char *path,
        const char *arg, ..., char * const envp[] );
int execve( const char *path,
        char * const argv[], char * const envp[] );
int execvp( const char *file,
        char * const argv[] );
```

- What are the differences from `execlp`?
- See man page for details

Copyright ©1997

Fork

- **Sometimes you don't want the new process to replace the current one**
- **fork() splits the current process into two copies:**

```
pid = fork();
```

/* pid != 0 */	/* pid == 0 */
Parent	**Child**

- **Return -1 on failure (see man page for more details)**
- **A child process has a unique process id (pid), different from its parent's**
- **A child process inherits environment variables from its parents**
- **A child process share file descriptors with its parent**
- **Are there any problems with sharing file descriptors?**

Fork (cont'd)

- Duplicate a file descriptor and "pick" an unused descriptor for it

  `int dup( int fd );`

  Duplicate "fd"

  Use the "lowest" numbered, unallocated file descriptor

  Thus, one can close file descriptors to let the system pick the right one

- Connect the standard input (stdin) of a program to a file:

```
int fd;
fd = open( "foo", O_RDONLY, 0 );
close( 0 );
dup( fd );
close( fd );
```

- This is not so convenient, so there is an extension call:

  `int dup2( int fd1, int fd2 );`

  Use "fd2" to duplicate "fd1"
  If "fd2" is in use, perform "close(fd2)"

```
fd = open( "foo", O_RDONLY, 0 );
dup( fd, 0 );
close( fd );
```

Wait

- **A parent may want to wait until its child dies (terminates), and then do something**

  ```
  pid_t wait( int *status_location );
  ```

  Returned "pid" is the pid of the terminated child process

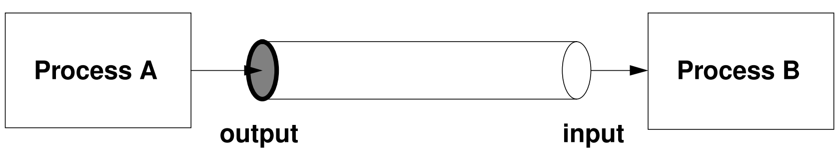
  Return -1 if no childen alive

- **The basic way to fork and wait is:**

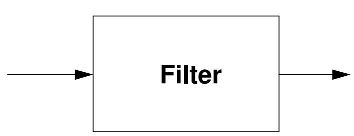
```
if ( fork() == 0 ) {
    execlp( "ls", "ls", -l", NULL );        /* child */
    ...
    fd = dup( fd1 );                        /* if needed */
    ...
}
pid = wait( &status );                      /* parent */
```


Pipes and Filters

- A pipe -- An interprocess communication channel



- A filter takes input from stdin and puts output to stdout



- Many Unix tools are filters and csh supports pipes

  ```
  ls -l | more
  ```

Pipe Creation

- System call to create a pipe

  `int pipe( int fd[ 2 ] );`

  Return 0 on success and -1 on failure

  fd[0] is opened for reading

  fd[1] is opened for writing

- **Two coordinated processes created by fork() can pass data using read() and write()**
- **Simple example: a parent writes to a pipe and a child reads from a pipe**

```
int pid, p[2];
...
pipe( p );
if ( ( pid = fork() ) == 0 ) {
    close( p[1] );
    ... read using p[0] as file descriptor ...
}
close( p[0] );
... write using p[1] as file descriptor ...
close( p[1] );           /* send EOF to the read port */
wait( &status );
```


Standard I/O Pipe

- **Connect a pipe to parent's stdout and child's stdin**

```
int pid, p[2];
...
pipe( p );
if ( ( pid = fork() ) == 0 ) {
    close( p[1] );
    dup2( p[0], 0 );         /* duplicate p[0] using stdin */
    close( p[0] );
    ... read from stdin ...
    exit( 0 );
}
close( p[0] );
dup2( p[1], 1 );             /* duplicate p[1] using stdout */
close( p[1] );
... write to stdout ...
wait( &status );
```

Signals

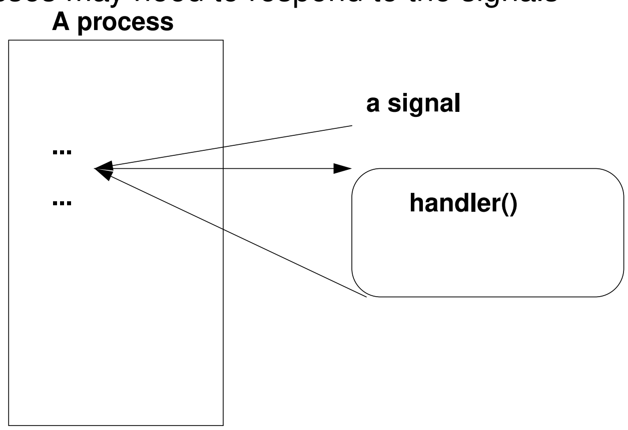
- Outside world signals are interrupts

    CTRL-C is typed

    phone is hung up

    . . .

- Processes may need to respond to the signals



UNIX Signal Facility

- **UNIX provides a mechanism for catching a signal (like an exception)**

```
void (* signal( int sig, void (* handler)( int ) ) ( int );
```

The function “handler()” will be invoked on a signal “sig”

```
handler( sig );
```

Return the old signal handler on success and -1 on an error

- **Examples**

```
#include <signal.h>
...
signal( SIGINT, SIG_IGN );         /* ignore interrupts */
...
signal( SIGINT, SIG_DFL );         /* restore the default handler */
```

- **There are more than 30 predefined signals in <signal.h>**

Signal Handling Example

- Remove “tempfile” on an interrupt:

```
#include <sys/signal.h>
char *tempfile = “temp.xxx”;

void cleanup( int sig )
{
    unlink( tempfile );
    exit( 1 );
};

void main( void )
{
    int fd;

    signal( SIGINT, cleanup );    /* set up cleanup() for interrupt */

    fd = open( tempfile, O_CREAT, 0666 );
        ...                       /* processing using tempfile */
    close( fd );
}
```

- **What happens to background processes (started with “&” for example)?**

Signal Handling Safely

```
#include <sys/signal.h>
char *tempfile = “temp.xxx”;

void cleanup( int sig )
{
    unlink( tempfile );
    exit( 1 );
};

void main( void )
{
    int fd;
                              /* protect background processes */
    if ( signal( SIGINT, SIG_IGN ) != SIG_IGN )
        signal( SIGINT, cleanup );
    fd = open( tempfile, O_CREAT, 0666 );
        ...                   /* processing using tempfile */
    close( fd );
}
```

Interpret Signals

- Interrupt a long printout and go to main processing loop

```
#include <sys/signal.h>
#include <setjmp.h>
jmp_buf jmpbuf;

void handler( int sig )
{
    signal( SIGINT, handler );            /* reset handler */
    fprintf( stderr, "Interrupted\n" );
    longjmp( jmpbuf, 0 );                 /* go to main loop */
}
void main( void )
{
    if ( signal( SIGINT, SIG_IGN ) != SIG_IGN )
        signal( SIGINT, handler);
    setjmp( jmpbuf );
    for ( ; ; ) {                         /* main processing loop */
        ...
    }
}
```


Signals with Fork

- Signals are sent to all your processes
- You often want to disable signals and enable them later
- fork() may cause two processes to read your terminal (/dev/tty) at the same time
- An example:

```
#include <signal.h>

...
if ( fork() == 0 )
    execlp( ... );
h = signal( SIGINT, SIG_IGN );     /* parent ignores interrupts */
wait( &status );                   /* until child is done */
signal( SIGINT, h );               /* restore interrupts */
```

Alarm

- Create a child process and kill it in 20 seconds

```
#include <signal.h>
int pid;

void OnAlarm( int sig )
{
    kill( pid, SIGKILL );
}
...
if ( ( pid = fork() ) == 0 ) {
    execlp( . . . );
}
signal( SIGALARM, OnAlarm );       /* setup the handler */
alarm( 20 );                       /* fire off the alarm */
...
```

- **What if you want to have an alarm in less than a second?**

```
unsigned ualarm( unsigned value, unsigned interval )
int setitimer( int which, struct itimerval *value,
                 struct itimerval *ovalue )
```


More Alarm Examples

- Implement time out in communication protocols

  Retransmission if no acknowledgment

  Decide “NFS is not responding.”

- Implement imestamps for performance measurements

  The typical way to get a timestamp is:

```
#include <sys/time.h>
...
gettimeofday( struct timeval *tp; struct timezone *tzp );
```

  But, this call takes x0 to x00 microseconds

  So, use ualarm to implement something less expensive.

- Other examples?