# Making Processes

- Topics
  - Process creation from the user level:
    - Fork, exec, wait, waitpid, pipe
- Learning Objectives:
  - Explain how processes are created
  - Create new processes, synchronize with them, and communicate exit codes back to the forking process.

# Where do Processes Come From?

- There are two models of process creation:
  - Copy an existing process (UNIX `fork/exec` model).
  - Single system call to create a new process (Windows model).
- The `pthread` interface is similar to the "create new process" model – it just creates threads instead of processes.
- In UNIX (and OS/161) we use the `fork/exec` model.

# Fork

- System call that copies the calling process, creating a second process that is identical (in all but one regard) to the process that called `fork`.
- We refer to the calling process as the parent and the new process as the child.
- On return from successful `fork`:
  - Parent: return value is the pid of the child process.
  - Child: return value is 0.
- If the `fork` fails:
  - No child process created.
  - Parent gets return value of -1 (and errno is set).

# Programming with `fork`

```
#include <unistd.h>
pid_t   ret_pid;

ret_pid = fork();
switch (ret_pid){
        case 0:
                /* I am the child. */
                break;
        case -1:
                /* Something bad happened. */
                break;
        default:
                /*
                 * I am the parent and my child's
                 * pid is ret_pid.
                 */
                break;
 }
```

# But what good are two identical processes?

- So fork let us create a new process, but it's identical to the original. That might not be very handy.
- Enter `exec` (and friends): The exec family of functions replaces the current process image with a new process image.
- `exec` is the original/traditional API
- `execve` is the modern day (more efficient) implementation.
- If execve returns, then it was unsuccessful and will return -1 and set errno.
- If successful, execve does not return, because it is off and running as a new process.
- Arguments:
  - Path: Name of a file to be executed
  - Args: Null-terminated argument vector
  - Envp: Null-terminated environment.

# Programming with Exec

```
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
pid_t    ret_pid;

ret_pid = fork();
switch (ret_pid){
        case 0:
                /* I am the child. */
                if (execve(path, argv, envp) == -1)
                        printf("Something bad happened: %s\n",
                            strerror(errno));
                break;
        case -1:
                /* Something bad happened. */
                break;
        default:
                /*
                 * I am the parent and my child's
                 * pid is ret_pid.
                 */
                break;
}
```

# Coordinating with your child

- Sometimes it is useful for a parent to wait until a specific child, all children, or any child exits.

```
pid_t wait (int *stat_loc)
pid_t waitpid(pid_t pid, int *stat_loc, int options)
```

- `wait`: suspends execution of the parent until some child of the parent terminates or the parent receives a signal.
  - Return value is the pid of the terminating process
  - `stat_loc` is filled in with a status indicating how/why the child terminated.
- `waitpid`: suspends until a particular child terminates.

# Programming with Fork, Exec, Wait

```
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
pid_t      ret_pid;

ret_pid = fork();
switch (ret_pid){
          case 0:
                    /* I am the child. */
                    if (execve(path, argv, envp) == -1)
                              printf("Something bad happened: %s\n",
                                  strerror(errno));
                    break;
          case -1:
                    /* Something bad happened. */
                    break;
          default:
                    /*
                     * I am the parent and my child's
                     * pid is ret_pid.
                     */
                    if (waitpid(ret_pid, &exit_status, WNOHANG) != ret_pid)
                              printf ("Something bad happened!\n");
                    break;
 }
```

# Communicating with child processes

- You've all used the | character to create pipes on the command line in the shell (I hope).
- What exactly does the pipe character do?
- The effect:
  - When you type:

    ```
    % foo | bar
    ```

  - The stdout stream of foo is connected to the stdin stream of bar.
- stdin/stdout (and stderr) are default file descriptors that are opened on behalf of every process.
  - By convention, stdin comes from the console
  - By convention, stdout goes to the display
- We want to connect foo's stdout to bar's stdin …

# The `pipe` system call

- `pipe(int filedes[2])` creates a pair of file descriptors, pointing to a pipe inode and places them in the array referenced by filedes.
  - filedes[0] is for reading
  - filedes[1] is for writing
- When a parent forks children, the parent and child share file descriptors.
- By combining, fork, exec, and pipe, parents can communicate with children and/or set up pipelines between children.

# Creating a Pipeline (foo | bar)

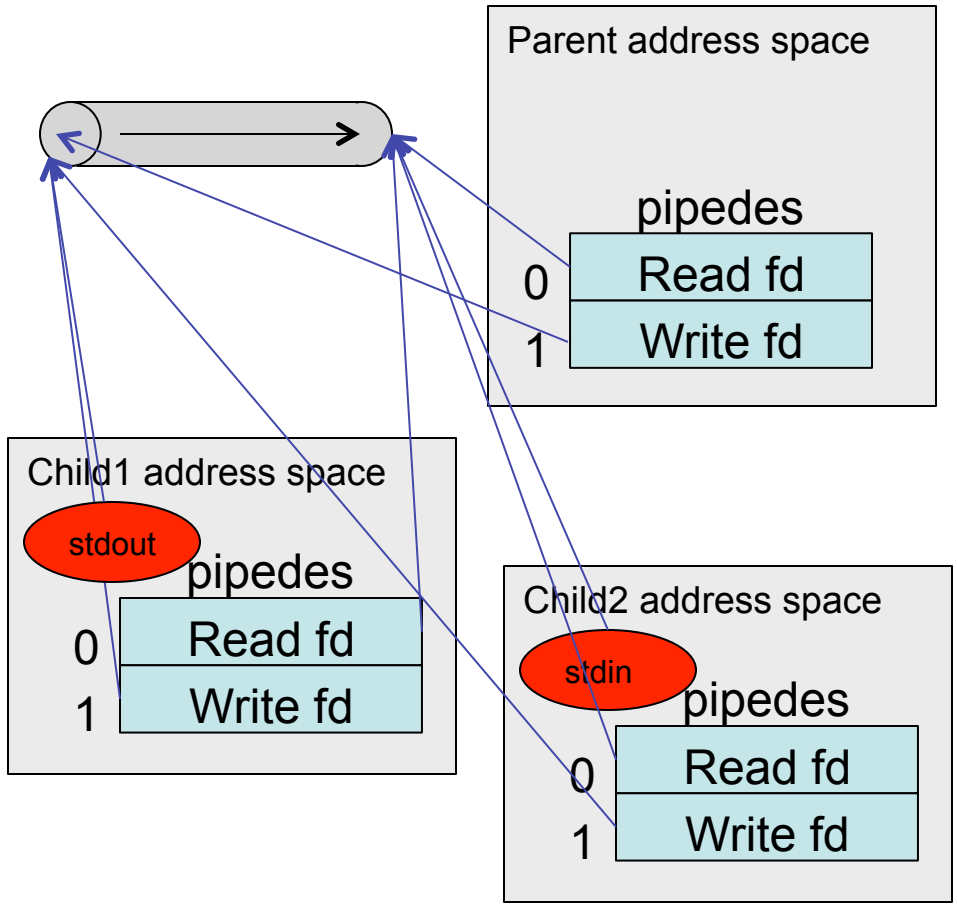
Note: Terrible error handling to save space!

```
pid_t child1, child2;
int pipedes[2], status;

assert (pipe(pipedes) == 0);                /* Create the pipe. */
child1 = fork();
if (child1 == 0) {
          /* child */
          close(pipedes[0]);                /* Close read end */
          dup2(pipedes[1], STDOUT_FILENO); /* Make stdout the same as the pipe write fd */
          execve("foo", argv, envp);        /* Assume argp, envp are set */
}
/* only parent gets here */
child2 = fork();
if (child2 == 0) {
          /* child */
          close (pipedes[1]);               /* Close writing end */
          dup2(pipedes[0], STDIN_FILENO);  /* Make stdin the same as the pipe read fd */
          execve("bar", argv, envp);
}
/* Parent once again */
close (pipedes[0]);                         /* Close pipe fDs in parent. */
close (pipedes[1]);
waitpid(child2, &status, 0);                /* Wait for second process to complete. */
```

# Creating a Pipeline (foo | bar)

Note: Terrible error handling to save space!

```
pid_t child1, child2;
int pipedes[2], status;

assert (pipe(pipedes) == 0);
child1 = fork();
if (child1 == 0) {
            /* child */
            close(pipedes[0]);
            dup2(pipedes[1], STDOUT_FILENO);
            execve("foo", argv, envp);
}
/* only parent gets here */
child2 = fork();
if (child2 == 0) {
            /* child */
            close (pipedes[1]);
            dup2(pipedes[0], STDIN_FILENO);
            execve("bar", argv, envp);
}
/* Parent once again */
close (pipedes[0]);
close (pipedes[1]);
waitpid(child2, &status, 0);
```



# The OS's job during process creation

- Fork:
  - Ensure that the parent is not running (i.e., if it is multi-threaded, **none** of its threads is running).
  - Create a new process.
  - Copy the parent process to the child process.
    - The details of this will depend on your implementation of processes.
  - Fix up the return of the two processes so the parent and child get different return values.
- Exec:
  - Replace old process contents with new process.
- Wait:
  - Allow parents to wait on children and properly collect exit status.
- Pipes and FDs:
  - Be sure to understand what happens to FDs across a fork, exec and make sure you do the same!