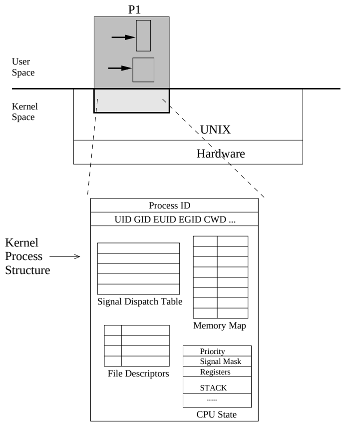
Labinstruction 1, Process Management

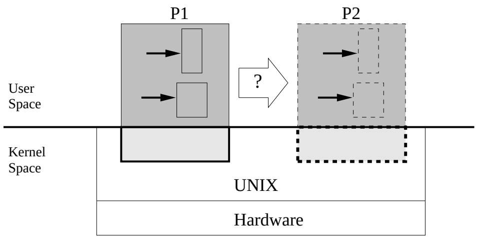
- processes in UNIX
- `fork(2)`
- `wait(2)`
- `exec(2)`
- init
- zombie processes
- I/O in C and UNIX
- fildescriptors
- streams
- `pipe(2)`
- labinstructions

Processes in UNIX



- Several processes can run at the same time and must all be handled by the operating system.
- The kernel needs to maintain data for each process it is running. The data tells the OS in what state the process is, what files are open, which user is running it, etc.
- The information is maintained in a *process structure*, (also called Process Control Block (PCB)), in the kernel.

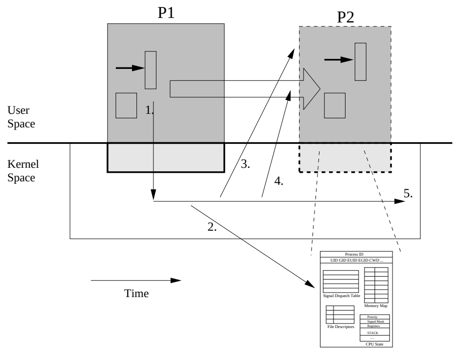
Create a new process



- How shall a new process be created?
- What program shall the new process be running?
- What values shall the new process have in its kernel process structure?
- All this is handled by the `fork()` system call.

`fork(2)`

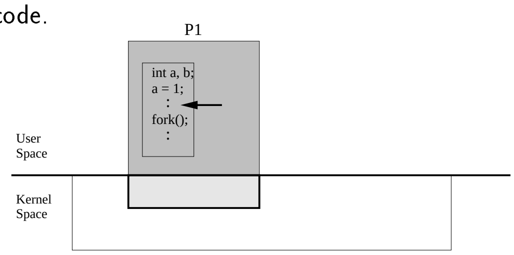
- UNIX system call.
- The only way to create an UNIX process.
- Very expensive operation!



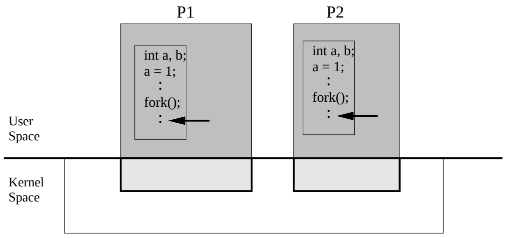
What happens:

1. process calls `fork`.
2. kernel creates kernel process structure for new process.
3. kernel allocates memory for new process.
4. kernel copies image of calling process to new process.
5. kernel schedules both processes.

When the kernel copies image of calling process to the new process, both processes will continue to run the "same" code.



Before the `fork()` call



After the `fork()` call

`fork(2)`

- `fork()` is called once, but returns *twice*
- the new process is a *child* of the calling process
- both parent and child get return value from `fork()`

Both processes are identical after `fork()`, except:

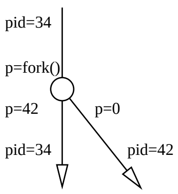
- each process has unique process identifier (pid)
- child process has no resource utilization
- `fork()` returns different value to parent and child

Note that:

- child gets own copies of all variables in parent
- child gets own copies of parent's open descriptors

Code layout

Since we have no real use for two completely identical processes, we use the return values to make one of the processes (usually the child) do something different. Your code should therefore look something like:



Or in C syntax:

```
/* create a new process */
p=fork();
switch (p) {
case -1:
  /* error: fork was unsuccessful */
case 0:
  /* this is the child process */
default:
 /* this is the parent process */
}
```

`exec(2)`

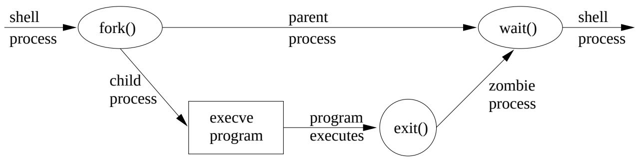
- the only way to run a program in UNIX.
- replaces an *existing* process with the contents of a (given) file.
- very expensive operation.
- shall *not* be used in the laboration.

Shells

- An interface between a user and the operating system.
- Let us give commands to the system.
- A program that starts other programs.
- Expands commands and handles wildcards, for example `*.*`.
- Uses `fork()` and `exec()`.
- Examples are: C-shell, TC-shell, Korn-shell etc.

Shells (cont.)

How a shell works:



Or in (simplified) C syntax:

```
while(1) {                          /* repeat forever */
  type_prompt();                    /* display prompt on the screen */
  read_command(command, params);    /* read input from terminal */

  p = fork();                       /* fork off child process */
  switch (p) {
    case -1:                        /* error: fork was unsuccessful */
      print_error_message();

    case 0:                         /* this is the child process */
      execve(command, params, 0);   /* replace the child process code
                                       with file specified by command */

    default:                        /* this is the parent process */
      waitpid(p, &status, 0);       /* wait for child process to finish *
  }
}
```

init

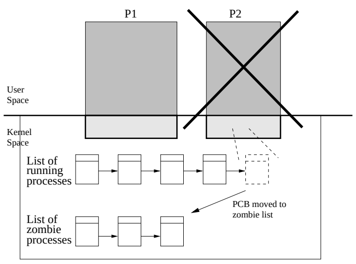
- We differ between "parent" and "child" processes → we get a hierarchy among the processes.
- Existential question: is there a process which is the ancestor of all processes?
- Answer: YES, the *init* process.
- when system is starting, init is created "manually" from boot image
- init has always pid=1

What happens:

- init started manually
- init forks, results in two copies of init
- child process execs, resulting in two different processes
- the cycle repeats...
- init is ancestor of *all* processes

Zombies

What happens when a process is finished?



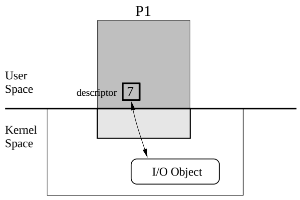
- information about resource usage is kept in pcb for each process
- after process exits, exit status is added to pcb
- pcb moved to *zombie list*
- pcb is not reused until someone reads this information
- `ps(1)` shows zombie processes as `defunct`
- zombies cannot be killed, because *there is no running process to kill*
- only one way to remove zombie: read pcb with `wait(2)`.
- only parent process can do this!

I/O in C and UNIX

- There is two ways of doing I/O using C and UNIX, namely using **descriptors** (non-buffered I/O) or **streams** (buffered I/O).

Descriptors (non-buffered I/O)

- A descriptor is an integer that identifies a particular I/O object, and is assigned by the kernel when we open an object for use.



- All I/O made via descriptors is made through system calls. Two examples are:
  - `read(2)` - read from a descriptor
  - `write(2)` - write to a descriptor
- I/O with descriptors is *stream* oriented, i.e. a stream of bytes without structure.

Descriptors (cont.)

- One of the UNIX philosophies is that all types of I/O should have the same, consistent interface, regardless of underlying object.
- Examples of things the standard I/O system calls allows you to manipulate are:
  - regular files.
  - special files, ie. peripherals such as terminals, printers and disk drives.
  - interprocess communication facilities, such as pipes and sockets.
- Example, file manipulation:

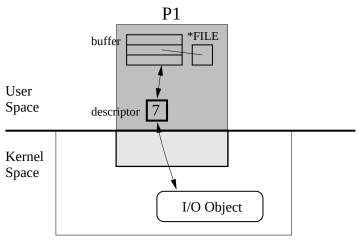
```
int fd;                /* File descriptor */
fd=open(fileName, ...);
  /* Open a file, return file descriptor */
...
read(fd, ...);         /* Read from file  */
...
write(fd, ...);        /* Write to file   */
...
close(fd);             /* Close the file,
                          freeing file descriptor */
```

Descriptors (cont.)

- A single file may be opened several times and thus have several file descriptors associated with it. See manual pages `dup(2)` and `open(2)`.
- When a process is started it will normally have three open descriptors:
  - 0 - standard input file (usually the keyboard)
  - 1 - standard output file (usually the shell)
  - 2 - standard error file

Streams (buffered I/O)

- Each access to a I/O object using a descriptor will result in a (costly) system call.
- A file with associated buffering is called a stream and is declared as a pointer to a defined type `FILE`.
- A single open file can be accessed both through streams and through file descriptors.



- When a process is started it will normally have three open streams associated with it (see `stdio(3S)`).
  - `stdin` - standard input, (as `cin` in C++).
  - `stdout` - standard output, (as `cout` in C++).
    - file - buffered
    - terminal - line buffered
  - `stderr` - standard error, (as `cerr` in C++). Is not buffered.

Streams (cont.)

- Each stream is connected to a corresponding file descriptor.
- The functions for manipulating streams are declared in the `stdio(3S)` package. Interesting functions are:
  - `fopen(3S)` - opens file and associates stream with it.
  - `fdopen(3S)` - associates a stream with a file descr.
  - `getc(3S)` - reads the next character from a stream.
  - gets(3S) - reads a string from a stream.
  - `putc(3S)` - write a character to a stream.
  - `fprintf(3S)` - write output to a stream.
  - fscanf(3S) - reads from a stream.
  - `flose(3S)` - close a stream.
- The command `setbuf(3S)` lets us set the amount of buffering we will have to a filedescriptor
- For more information see manual pages `stdio(3S)` and `fopen(3S)`.

printf(3C)

- `printf` and other I/O functions are declared in the `stdio.h` library.
- `printf(const char *format, ...)` is the standard C library output function which allows you to get a nice printout on your screen.
- The first argument to `printf(3S)` should always be a *format-string*.
- The format-string includes *text* and *transforming specifications* on how the values that are printed shall be interpreted. All transforming specifications begins with a % sign. The rest of the parameters to `printf(3S)` is the values the shall be printed within the text.
- Three ways of printing the same thing:
  - `printf("Name: %s and age: %d \n", name, i);`
  - `fprintf(stdout, "Name: %s and age: %d \n", name, i);`
  - `cout << "Name: " << name << " and age: " << i << newline;`
- For more information see manual page `printf(3S)`.

Pipes

- Pipes are a communication mechanism for sending information between processes.
- Can be used to fasten up dataprocessing between processes in a pipe-line like fashion.
- They are commonly used within shells to connect the standard output of one utility to the standard input of another.
- For example, the sequence:

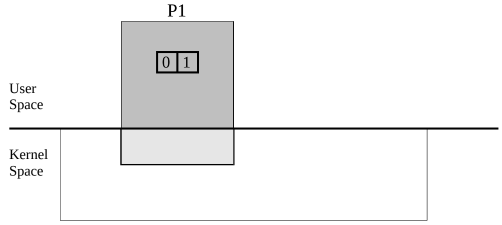
  `$ command1 | command2`

  causes the standard output of `command1` to "flow through" to the standard input of `command2`.

Pipes, cont.

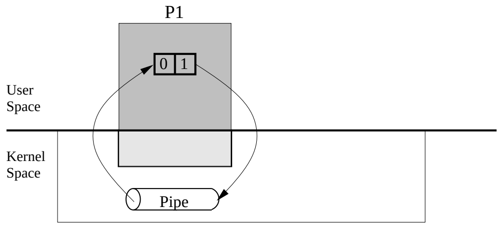
A pipe is a pair of connected descriptors, normally declared as an array. We use the system call `pipe(2)` to create one:

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



Before the `pipe(fds)` call.

Pipes, cont.



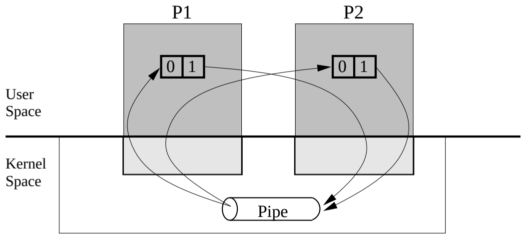
After the `pipe(fds)` call.

We can write to one of the descriptors, (`fds[1]`), and read from the other, (`fds[0]`) like:

- `write(fds[1], ...)`
- `read(fds[0], ...)`

Pipes, cont.

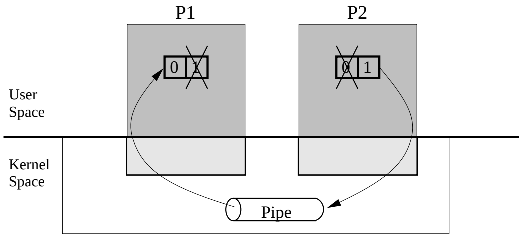
After a process forks, both parent and child have copies of descriptors, which refer to same object:



The pipe is located in the kernel and will not be copied between the processes.

Pipes, cont.

Each process closes one of the descriptors, leaving a one-way communication channel:



- P1 and P2 can communicate through the pipe.
- You can say that P2 is a producer and P1 a consumer.

Limitations

- only “related” processes can use pipes
- stream of bytes has no structure

think about...

- what happens when a process writes to pipe, and no process reads from the other end?
- what happens when a process reads from pipe before any data has been written?
- what happens when a process reads from pipe, and no process can write to the other end?
- see the manual pages for `pipe(2)`, `read(2)` and `write(2)` on how to use the commands and for further clues of the above questions.

Lab 1: Process Management

- Using the `ps` command.
- Process management. How to create processes and how to avoid defunct processes.
- Process communication. How to communicate between processes using a pipe and non-buffered I/O.
  - Main process: Creates pipe, consumer- and producer process.
  - Producer process: Reads possible passwords from a file, (`words1` or `words2`), and sends them, one at a time, through the pipe to the consumer.
  - Consumer Process: Receives possible passwords from the producer and tests them against user names using the `guess(account, password)` function.

  The `guess()` function, the number of user names (accounts) to test, `numaccts`, and an array, `accounts[]`, holding all the user names, are all located in the files `guess.h` and `guess.c`.

  The paths to the guess files, like the needed compilation information, are given in the laboration.