

# DPST1092 24T3 — Processes

<https://www.cse.unsw.edu.au/~dp1092/24T3/>

## Housekeeping

Catch UP / ADMIN

Rest of the term

- Week 9 Processes
- Week 10 THREADS
- Week 11 VIRTUAL MEMORY (Not in final exam)
- Week 12 Revision Important information
- Final exam on 20th November(will review in W12)
- Assignemnt 2 is out. Due Week 12 Friday
- No more Friday lectures (Tue and Thur only).

## Processes

A *process* is an instance of an executing program

Each process has an *execution state*, defined by

- current execution point (PC register)
- current values of CPU registers
- current contents of its virtual address space
- information about open files, sockets, etc.

Every process in Unix/Linux is allocated a unique process ID (PID)

- a +ve integer, unique among currently executing processes
- with type `pid_t` (defined in `<unistd.h>`)
- PID 0 is often used for the *Operating System* Kernel process. Can't be killed by users
- PID 1 is `init` ("used to boot the system")
- low PIDs are typically system-related as they start when the system is booted (but PIDs are recycled so this is not always the case)

## Parent Processes

Each process has a *parent process*

- typically, the process that created the current process
- if the parent of the process dies, it becomes an orphan and is inherited by process 1

A process may have *child processes*

- these are processes that it created

## Unix Tools

Unix provides a range of tools for manipulating processes

Commands:

- **ps** ... show process information
  - ▶ `ps`
  - ▶ `ps -ef`
  - ▶ `ps -u z1234567 -o pid,ppid,time,stat,args`
- **top** ... show high-cpu-usage process information
- **kill** ... send a signal to a process

## System Calls to Get information about a process

### `pid_t getpid()`

- requires `#include <sys/types.h>`
- returns the process ID of the current process

### `pid_t getppid()`

- requires `#include <sys/types.h>`
- returns the parent process ID of the current process

For more details: `man 2 getpid`

## System Calls to Get information about a process

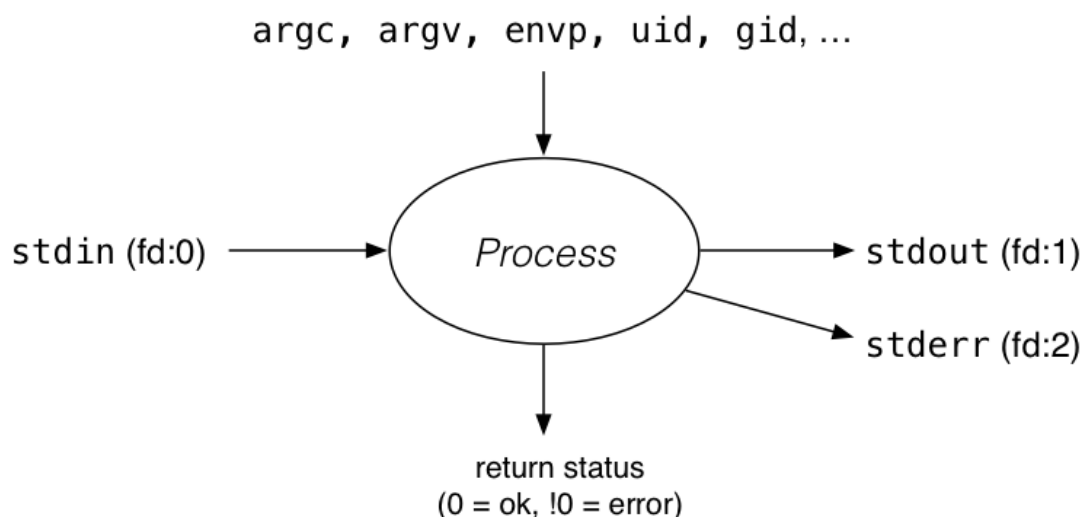
Minimal example for `getpid()` and `getppid()`:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(void){
    printf("My PID is (%d)\n", getpid());
    printf("My parent's PID is (%d)\n", getppid());
    return 0;
}
```

## Unix/Linux Processes

Environment for processes running on Unix/Linux systems



# Environment Variables

Every process is passed a set of *environment variables* as an array of strings of the form `name=value`, terminated with `NULL`.

These can be accessed via

- access via global variable `environ`
- many C implementation also provide as 3rd parameter to `main`: `int main(int argc, char *argv[], char *env[])`

```
// print all environment variables
extern char **environ;
for (int i = 0; environ[i] != NULL; i++) {
    printf("%s\n", environ[i]);
}
```

source code for `environ.c`

- Most programs instead use `getenv()` and `setenv()` to access environment variables

## `getenv()` – get an environment variable

```
#include <stdlib.h>

char *getenv(const char *name);
```

- search environment variable array for **name=value**
- returns **value**
- returns **NULL** if **name** not in environment variable array

```
char *value = getenv("PATH");
printf("Environment variable 'PATH' has value '%s'\n", value);
```

source code for `get_env.c`

## `setenv()` – set an environment variable

```
#include <stdlib.h>

int setenv(const char *name, const char *value, int overwrite);
```

- adds **name=value** to environment variable array
- if **name** in array, value changed if **overwrite** is non-zero

# Environment Variables - Why are they useful

- Unix-like shells have simple syntax to set environment variables
  - ▶ common to set environment in startup files (e.g. `.profile`)
  - ▶ then passed to any programs they run
- Almost all program pass the environment variables they are given to any programs they run
  - ▶ perhaps adding/changing the value of specific environment variables
- Provides simple mechanism to pass settings to all processes, e.g.
  - ▶ timezone (TZ)
  - ▶ user's preferred language (LANG)
  - ▶ directories to search for programs (PATH)
  - ▶ user's home directory (HOME)

## Multi-tasking

Process management is a critical OS functionality

On a typical modern operating system

- multiple processes are active "simultaneously" (*multi-tasking*)

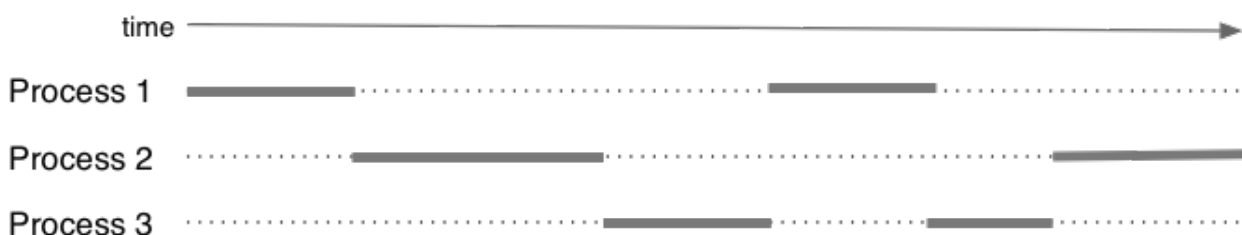
The operating system provides a virtual machine to each process:

- each process executes as if it is the only process running on the machine
- each process has its own address space (N bytes, addressed 0..N-1)

## Multi-tasking

When there are multiple processes running on the machine

- each process uses the CPU until *pre-empted* or exits
- then another process uses the CPU until it too is pre-empted
- eventually, the first process will get another run on the CPU



Overall impression: three programs running simultaneously

## Multi-tasking

What can cause a process to be pre-empted?

- it runs "long enough" and the OS replaces it by a waiting process
- it needs to wait for input/output or some other operation

On pre-emption ...

- the process's entire state must be stored
- the new process's state must be restored
- this change is called a **context switch** (these are very expensive)

## Multi-tasking

The context (or *state*) for each process is store in a Process Control Block (PCB).

Typical contents of a PCB:

- PID
- static information: program code and constant data
- dynamic state: heap, stack, registers, program counter
- OS-supplied state: environment variables, stdin, stdout
- status running, ready, suspended, exited
- privileges: effective user ids
- memory management info: (reference to) page table
- accounting: CPU time used, amount of I/O done
- I/O: open file descriptors

The operating system maintains a table of PCBs. One for each active process.

## More Process-related System Calls

Unix/Linux system calls:

- **fork()** ... create a new process
- **\_exit()** ... terminate an executing process
- **wait()** ... wait for state change in child process
- **execve()** ... convert one process into another

## pid\_t fork(void)

- requires `#include <unistd.h>`
- creates new process by duplicating the calling process
- new process is the *child*, calling process is the *parent*
- child has a different process ID (pid) to the parent
- in the child, `fork()` returns 0
- in the parent, `fork()` returns the pid of the child
- if the system call fails, `fork()` returns -1
- child inherits copies of parent's address space and open fd's

Typically, the child pid is a small increment over the parent pid

# fork

Minimal example for `fork()`:

```
#include <stdio.h>
#include <unistd.h>

int main(void){
    pid_t pid;
    pid = fork();
    if (pid < 0)
        perror("fork() failed");
    else if (pid == 0)
        printf("I am the child.\n");
    else
        printf("I am the parent.\n");
    return 0;
}
```

# \_exit

## void \_exit(int status)

- terminates current process
- closes any open file descriptors
- a SIGCHLD signal is sent to parent
- returns `status` to parent (via `wait()`)
- any child processes are inherited by `init` (pid=1)
- termination may be delayed waiting for i/o to complete

On final exit, process's process table and page table entries are removed

**void exit(int status)**

- triggers any functions registered as `atexit()`
- flushes stdio buffers; closes open FILE \*'s
- then behaves like `_exit()`

## Exercise: The `_exit()` Function

What do you think the difference in output will be between the following 2 programs?

```
int main(void){
    printf("Hello");
    exit(0);
}
```

```
int main(void){
    printf("Hello");
    _exit(0);
}
```

## Zombie Processes



Photo credit: kenny Louie, Flickr.com



A process cannot terminate until its parent is notified.

- if `exit()` called, operating system sends `SIGCHLD` signal to parent
- `exit()` will not return until parent handles `SIGCHLD`

*Zombie process* = exiting process waiting for parent to handle `SIGCHLD`

- all processes become zombie until `SIGCHLD` handled
- bug in parent that ignores `SIGCHLD` creates long-term zombies
- note that zombies occupy a slot in the process table and wastes resources

*Orphan process* = a process whose parent has exited

- when parent exits, orphan is assigned `pid=1` as its parent
- `pid=1` always handles `SIGCHLD` when process exits

## waitpid

### `pid_t waitpid(pid_t pid, int *status, int options)`

- pause current process until process `pid` changes state
  - ▶ where state changes include finishing, stopping, re-starting, ...
- ensures that child resources are released on exit (ie does not become a zombie)
- special values for `pid` ...
  - ▶ if `pid = -1`, wait on any child process
  - ▶ if `pid = 0`, wait on any child in process group
  - ▶ if `pid > 0`, wait on the specified process

### `pid_t wait(int *status)`

- equivalent to `waitpid(-1, &status, 0)`
- pauses until one of the child processes terminates

## waitpid

### More on `waitpid(pid, &status, options)`

- `status` is set to hold info about `pid`
  - ▶ e.g. exit status if `pid` terminated
  - ▶ macros allow precise determination of state change (e.g. `WIFEXITED(status)`, `WCOREDUMP(status)`)
- `options` provide variations in `waitpid()` behaviour
  - ▶ default: wait for child process to terminate
  - ▶ `WNOHANG`: return immediately if no child has exited
  - ▶ `WCONTINUED`: return if a stopped child has been restarted

For more information: `man 2 waitpid`

## waitpid

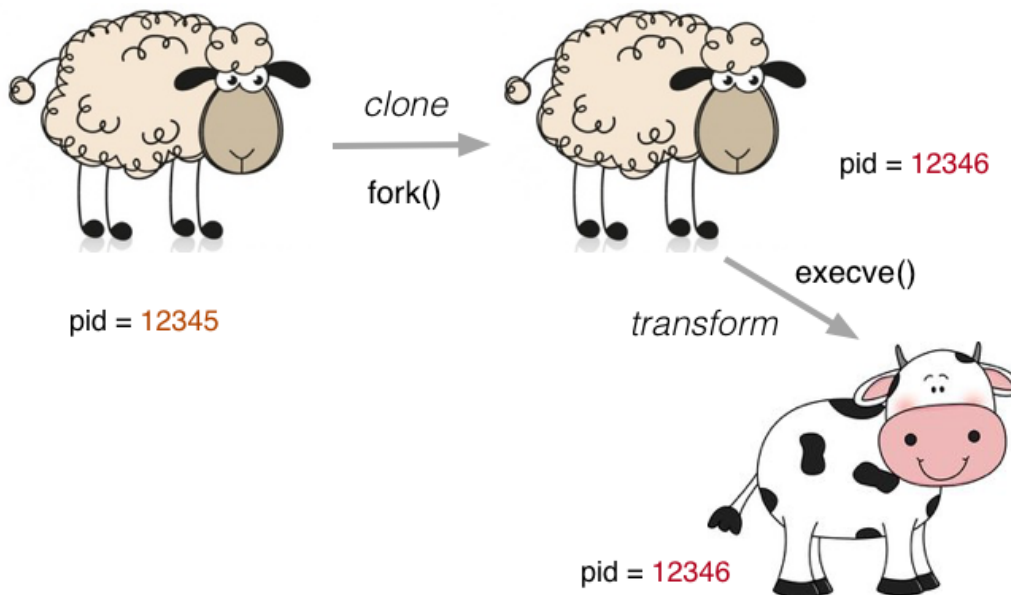
Minimal example for wait():

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(void)
{
    pid_t pid;
    pid = fork();
    if (pid == 0)
        printf("I am the child.\n");
    else {
        wait(NULL);
        printf("I am the parent.\n");
    }
    return 0;
}
```

## execve

How Unix creates processes:



## execve

```
int execve(char *Path, char *Argv[], char *Envp[])
```

- transforms current process by executing Path object
  - ▶ Path must be an executable, binary or script (starting with # !)
- passes arrays of strings to new process
  - ▶ both arrays terminated by a NULL pointer element
  - ▶ envp[] contains strings of the form key=value
- much of the state of the original process is lost, e.g.
  - ▶ new virtual address space is created, signal handlers reset, ...
- new process inherits open file descriptors from original process
- on error, returns -1 and sets errno
- if successful, does not return

On Unix, processes create new different processes via:

```
pid_t pid = fork();
if (pid > 0)
    // parent ...
    wait(NULL); // wait for child to complete
else {
    // child ...
    char *cmd = "/x/y/z"; // name of executable
    char **args;
    ... // set up command-line arguments
    char **env;
    ... // set up environment variables

    execve(cmd, args, env); // child is transformed
}
```

## Exercise: Executor

Write a small program that will run other programs

- reads, one per line, values for command-line arguments
- trims each line and stores pointer to it in array `args []`
- uses `args [0]` as the path of the program to run
- uses `args []` as `argv []` in the exec'd process
- passes no `envp []` values (i.e. `envp=NULL`)
- invokes the specified program then waits for it to complete
- displays the exit status of the invoked process

## posix\_spawn

```
int posix_spawn(pid_t *pid, const char *path, const posix_spawn_file_actions_t *file_actions, const posix_spawnattr_t *attrp, char *const argv[], char *const envp[]);
```

- creates new process, running program at path
- `argv` specifies `argv` of new program
- `envp` specifies environment of new program
- `*pid` set to process id of new program
- `file_actions` specifies file actions to be performed before running program
  - ▶ can be used to re-direct stdin or stdout to file or pipe
  - ▶ advanced topic
- `attrp` specifies attributes for new process

# Review: Processes and Multi-tasking

Multi-tasking = multiple processes are "active" at the same time

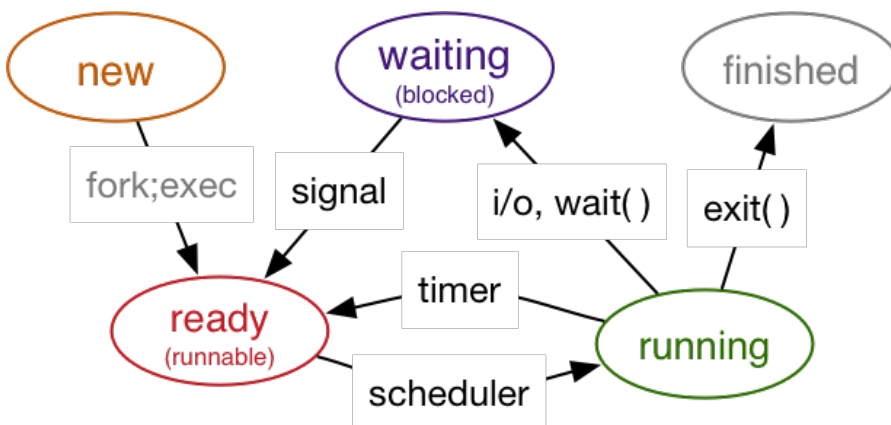
- processes are not necessarily *executing* simultaneously
  - ▶ although this could happen if there are multiple CPUs
- more likely, have a mixture of processes
  - ▶ some are *blocked* waiting on a signal (e.g. i/o completion)
  - ▶ some are *runnable* (ready to execute)
  - ▶ one is running (on each CPU)

Aims to give the appearance of multiple simultaneous processes

- by switching process after one runs for a defined *time slice*
- after timer counts down, current process is *pre-empted*
- a new process is selected to run by the system *scheduler*

## Process States

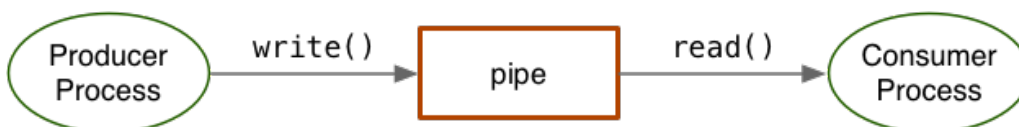
How process state changes during execution ...



## pipe() – stream bytes between processes

A common style of inter process interaction (communication)

- producer process writes to byte stream (cf. stdout)
- consumer process reads from same byte stream



A *pipe* provides buffered i/o between producer and consumer

- producer blocks when buffer full; consumer blocks when buffer empty

Pipes are bidirectional unless processes close one file descriptor.

It is a good idea to do this and only use pipes for unidirectional communication. If you need two way communication, use two pipes.

## pipe() – stream bytes between processes

### int pipe(int fd[2])

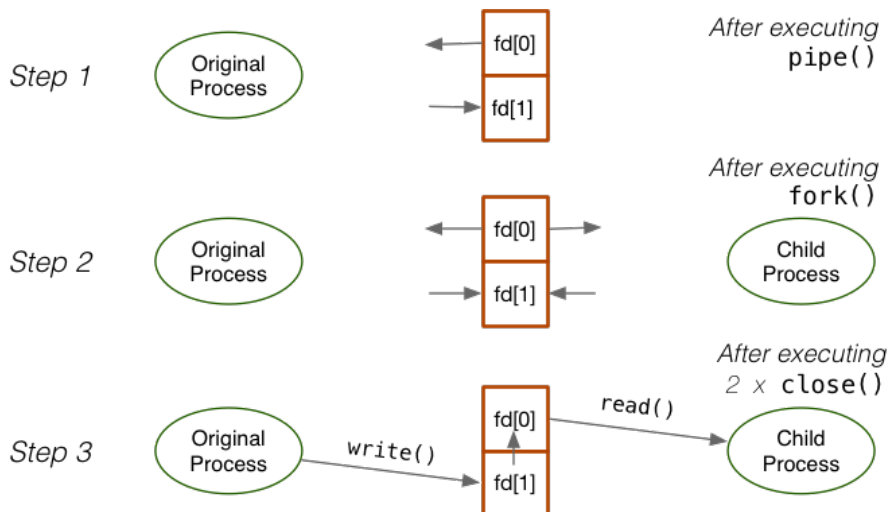
- open two file descriptors (to be shared by processes)
- fd[0] is opened for reading; fd[1] is opened for writing
- return 0 if OK, otherwise return -1 and sets errno

Creating the pipe would then be followed by

- fork() to create a child process
- both processes have copies of fd[]
- one can write to fd[1], the other can read from fd[0]

## pipe() – stream bytes between processes

Creating a pipe ...



## pipe() – stream bytes between processes

Example: setting up a pipe

```
int main(void) {
    int fd[2], pid; char buffer[10];
    assert(pipe(fd) == 0);
    pid = fork();
    assert(pid >= 0);
    if (pid != 0) { // parent
        close(fd[0]); // writer; don't need fd[0]
        write(fd[1], "123456789", 10);
        close(fd[1]);
    }
    else { // child
        close(fd[1]); // reader; don't need fd[1]
        read(fd[0], buffer, 10);
        printf("got \"%s\"\n", buffer);
        close(fd[0]);
    }
    return 0;
}
```

## pipe() – stream bytes between processes

It is important to close unused duplicate pipe file descriptors

- If there are open write end file descriptors, read(2) won't return 0 and will wait for more input
- If all read end file descriptors have been closed, then a write(2) will cause a SIGPIPE signal to be generated.

## popen() – a convenient but unsafe way to set up pipe

A common pattern in pipe usage

- set up a pipe between parent and child
- exec() child to become a new process talking to parent

Because so common, a library function is available for it ...

### **FILE \*popen(char \*Cmd, char \*Mode)**

- analogous to fopen, except first arg is a command
- Cmd is passed to shell for interpretation
- returns FILE\* which be read/written depending on Mode
- returns NULL if can't establish pipe or invalid Cmd

## popen() – a convenient but unsafe way to set up pipe

Example of popen()

```
//popen is a convenient but unsafe way to set up a pipe
//It passes a string to a shell for evaluation
//It is brittle and highly vulnerable to security exploits
//Suitable for quick debugging or throw away programs only
int main(void)
{
    FILE *p = popen("ls -l", "r");
    assert(p != NULL);
    char line[200], a[20], b[20], c[20], d[20];
    long int tot = 0, size = 0;
    while (fgets(line, 199, p) != NULL) {
        sscanf(line, "%s %s %s %s %ld",
               a, b, c, d, &size);
        fputs(line, stdout);
        tot += size;
    }
    printf("Total: %ld\n", tot);
}
```

## posix\_spawn and pipes (advanced topic)

- `int posix_spawn_file_actions_destroy( posix_spawn_file_actions_t *file_actions);`
- `int posix_spawn_file_actions_init( posix_spawn_file_actions_t *file_actions);`
- `int posix_spawn_file_actions_addclose( posix_spawn_file_actions_t *file_actions, int fildes);`
- `int posix_spawn_file_actions_adddup2( posix_spawn_file_actions_t *file_actions, int fildes, int newfildes);`
- functions to combine file operations with posix\_spawn process creation
- awkward to understand & use - but robust
- example: capturing output from a process - source code for `spawn_read_pipe.c`
- example: sending input to a process - source code for `spawn_write_pipe.c`