A process is an instance of an executing program

Each process has an execution state, defined by

- current values of CPU registers
- current contents of its (virtual) memory
- information about open files, sockets, etc.

On Unix/Linux:

- each process had a unique process ID (pid)
- positive integer - type `pid_t` defined in `unistd.h`
- process 0 is effectively part of the operating system
- process 1 (`init`) - used to boot the system
- some parts of operating system may run as processes
- low-numbered processes are typically system-related process started at boot-time
Each process has a *parent process*

- initially it is the process that created it
- if a process’ parent terminates, its parent becomes process 1

Unix provides a range of commandss for manipulating processes, e.g.:

- `sh` ... for creating processes via object-file name
- `ps` ... show process information
- `w` ... show per-user process information
- `top` ... show high-cpu-usage process information
- `kill` ... send a signal to a process
Aside: Zombie Process

Zombie Process?

Photo credit: Kenny Louie, Flickr.com
Aside: zombie Processes

- a process can’t 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 zombies until SIGCHLD handled
- bug in parent that ignores SIGCHLD creates long-term zombie processes
  - wastes some operating system resources
- *Orphan process* = a process whose parent has exited
  - when parent exits, orphan is assigned pid=1 (init) as its parent
  - init should always handles SIGCHLD when process exits
Multi-Tasking

On a typical modern operating system

- multiple processes are active "simultaneously" (*multi-tasking*)
- operating systems provides a virtual machine to each processs
  - each process executes as if the only process running on the machine
  - e.g. each process has its own address space (N bytes, addressed 0..N-1)

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
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, or output or ...

On pre-emption ..

- the process’s entire state must be saved
- the new process’s state must be restored
- this change is called a *context switch*
- *context switches* are expensive

The operating system’s process scheduling attempts to:

- fairly sharing the CPU(s) among competing processes
- minimize response delays (lagginess) for interactive users
- meet other real-time requirements (e.g. self-driving car)
- minimize number of expensive context switches
Environment for processes running on Unix/Linux systems

- argv
- envp
- uid
- gid
- stdin (fd:0)
- stdout (fd:1)
- stderr (fd:2)
- return status
  (0 = ok, !0 = error)
Process-related Unix/Linux Functions/System Calls

- **posix_spawn()** ... create a new process, see also
  - **clone()** ... duplicate current process
    - address space can be shared to implement threads
    - only use clone if posix_spawn can’t do what you want
- **fork()** ... duplicate current process - do not use in new code
- **execvp()** ... replace current process
- **system() popen()** ... create a new process via a shell (unsafe)
- **exit()** ... terminate current process, see also
  - **_exit()** ... terminate current process immediately
    - atexit functions not be called: stdio buffers not flushed
- **getpid()** ... get process ID
- **getpgid()** ... get process group ID
- **waitpid()** ... wait for state change in child process
posix_spawn() - run a new process

```
#include <spawn.h>

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
Simple example using posix_spawn() to run /bin/date

```c
pid_t pid;

extern char **environ;

char *date_argv[] = {"/bin/date", "--utc", NULL};

// spawn "/bin/date" as a separate process
if (posix_spawn(&pid, "/bin/date", NULL, NULL, date_argv, environ) != 0) {
    perror("spawn");
    exit(1);
}

// wait for spawned processes to finish
int exit_status;
if (waitpid(pid, &exit_status, 0) == -1) {
    perror("waitpid");
    exit(1);
}

printf("/bin/date exit status was %d\n", exit_status);
```
fork() - clone yourself

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

pid_t fork(void);
```

- 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 file descriptors
- do not use in new code use `posix_spawn` instead
  - `fork` appears simple but prone to subtle bugs
Simple example of using fork()

// fork creates 2 identical copies of program
// only return value is different

pid_t pid = fork();
if (pid == -1) {
    perror("fork");  // print why the fork failed
} else if (pid == 0) {
    printf("I am the child because fork() returned %d.\n", pid);
} else {
    printf("I am the parent because fork() returned %d.\n", pid);
}

source code for fork.c

$ dcc fork.c
$ a.out
I am the parent because fork() returned 2884551.
I am the child because fork() returned 0.
$
execvp() - replace yourself

```c
#include <unistd.h>

int execvp(const char *file, char *const argv[]);
```

- replaces current process by executing `file`
  - `file` must be an executable: binary or script starting with `#!/`
- `argv` specifies argv of new program
- most of the current process is reset
  - 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
Simple example of using exec()

```c
char *echo_argv[] = {"/bin/echo","good-bye","cruel","world",NULL};
execv("/bin/echo", echo_argv);
// if we get here there has been an error
perror("execv");
```

source code for exec.c

```
$ dcc exec.c
$ a.out
good-bye cruel world
$ 
```
Simple example using fork() and exec() to run /bin/date

```c
pid_t pid = fork();
if (pid == -1) {
    perror("fork"); // print why fork failed
} else if (pid == 0) { // child
    char *date_argv[] = {"/bin/date", "--utc", NULL};
    execv("/bin/date", date_argv);
    perror("execvpe"); // print why exec failed
} else { // parent
    int exit_status;
    if (waitpid(pid, &exit_status, 0) == -1) {
        perror("waitpid");
        exit(1);
    }
    printf("/bin/date exit status was %d\n", exit_status);
}
```

source code for fork_exec.c
system() - convenient but unsafe way to run another program

#include <stdlib.h>

int system(const char *command);

- runs command via /bin/sh
- waits for command to finish and returns exit status
- convenient but brittle and highly vulnerable to security exploits
- use for quick debugging and throw-away programs only

// run date --utc to print current UTC

int exit_status = system("/bin/date --utc");
printf("/bin/date exit status was %d\n", exit_status);

return 0;

source code for system.c
running `ls -ld` via `posix_spawn`

```c
char *ls_argv[argc + 2];
ls_argv[0] = "/bin/ls";
ls_argv[1] = "-ld";
for (int i = 1; i <= argc; i++) {
    ls_argv[i + 1] = argv[i];
}

pid_t pid;
extern char **environ;
if (posix_spawn(&pid, "/bin/ls", NULL, NULL, ls_argv, environ) != 0) {
    perror("spawn");
    exit(1);
}
```
-running ls -ld via posix_spawn

```c
int exit_status;
if (waitpid(pid, &exit_status, 0) == -1) {
    perror("waitpid");
    exit(1);
}
// exit with whatever status ls exited with
return exit_status;
```

(source code for lsd_spawn.c)
char *ls = "/bin/ls -ld";
int command_length = strlen(ls);
for (int i = 1; i < argc; i++) {
    command_length += strlen(argv[i]) + 1;
}

// create command as string
char command[command_length + 1];
strncpy(command, ls);
for (int i = 1; i <= argc; i++) {
    strcat(command, " ");
    strcat(command, argv[i]);
}
int exit_status = system(command);
getpid & getppid

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

pid_t getpid(void);
pid_t getppid(void);

- getpid returns the process ID of the current process
- getppid returns process ID of the parent of current process
waitpid

#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *wstatus, int options);

pid_t wait(int *wstatus);

- **waitpid** pauses current process until process pid changes state
  - where state changes include finishing, stopping, re-starting, ...
- ensures that child resources are released on exit
- 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
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`
• when Linux/Unix program are passed **environment variables**
• **environment variables** are array of strings of form name=value
• array is NULL-terminated
• access via global variable **environ**
• many C implementation also provide as 3rd parameter to **main**:

```c
int main(int argc, char *argv[], char *env[])
```

• most program use **getenv** & **setenv** to access environment variables
• can access environment variables directly, eg:

```c
// 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)
accessing an environment variable with getenv

```
#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

```c
// print value of environment variable STATUS
char *value = getenv("STATUS");
printf("Environment variable 'STATUS' has value '%s'\n", value);
```

source code for get_status.c
setting an environment variables with `setenv`

```c
#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

```c
// set environment variable STATUS
setenv("STATUS", "great", 1);
char *getenv_argv[] = {"./get_status", NULL};
pid_t pid;
extern char **environ;
if (posix_spawn(&pid, "/get_status", NULL, NULL, 
               getenv_argv, environ) != 0) {
    perror("spawn");
    exit(1);
}
```

(source code for `set_status.c`)
changing behaviour with an environment variable

```c
pid_t pid;
char *date_argv[] = { "/bin/date", NULL };
char *date_environment[] = { "TZ=Australia/Perth", NULL };
// print time in Perth
if (posix_spawn(&pid, "/bin/date", NULL, NULL, date_argv,
                date_environment) != 0) {
    perror("spawn");
    return 1;
}
int exit_status;
if (waitpid(pid, &exit_status, 0) == -1) {
    perror("waitpid");
    return 1;
}
printf("/bin/date exit status was %d\n", exit_status);
```

source code for spawn_environment.c
exit() - terminate yourself

#include <stdlib.h>

void exit(int status);

- triggers any functions registered as atexit()
- flushes stdio buffers; closes open FILE *’s
- terminates current process
- a SIGCHLD signal is sent to parent
- returns status to parent (via waitpid())
- any child processes are inherited by init (pid 1)

Also void _exit(int status)

- terminates current process without triggering functions registered as atexit()
- stdio buffers not flushed
#include <unistd.h>

int pipe(int pipefd[2]);

- a pipe is a unidirectional byte stream provided by operating system
- pipefd[0] - set to file descriptor of read end of pipe
- pipefd[1] - set to file descriptor of write end of pipe
- bytes written to pipefd[1] will be read from pipefd[1]
- child processes (by default) inherit file descriptors including for pipe
- parent can send/receive bytes (not both) to child via pipe
- parent and child should both close the pipe file descriptor they are not using
  - e.g. if bytes being written (sent) parent to child
    - parent should close read end pipefd[0]
    - child should close write end pipefd[1]
- pipe (and other) file descriptors can be used with stdio via fdopen
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);

- runs command via /bin/sh
- if type is "w" pipe to stdin of command created
- if type is "r" pipe from stdout of command created
- FILE * stream returned - get then use fgetc/fputc etc
- NULL returned if error
- close stream with pclose (not fclose)
  - pclose waits for command and returns exit status
- convenient but brittle and highly vulnerable to security exploits
- use for quick debugging and throw-away programs only
// popen passes string to a shell for evaluation
// brittle and highly-vulnerable to security exploits
// popen is suitable for quick debugging and throw-away programs only

FILE *p = popen("/bin/date --utc", "r");
if (p == NULL) {
    perror("\n");
    return 1;
}

char line[256];
if (fgets(line, sizeof line, p) == NULL) {
    fprintf(stderr, "no output from date\n");
    return 1;
}

printf("output captured from /bin/date was: '%s'\n", line);
pclose(p); // returns command exit status
int main(void) {
    // popen passes command to a shell for evaluation
    // brittle and highly-vulnerable to security exploits
    // popen is suitable for quick debugging and throw-away programs
    //
    // tr a-z A-Z - passes stdin to stdout converting lower case to upper case
    FILE *p = popen("tr a-z A-Z", "w");
    if (p == NULL) {
        perror(""");
        return 1;
    }
    fprintf(p, "plz date me\n");
    pclose(p); // returns command exit status
    return 0;
}
posix_spawn and pipes (advanced topic)

```c
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