Environment Variables

- When run, a program is passed a set of **environment variables** an array of strings of the form `name=value`, terminated with NULL.

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

    ```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*

- Recommended you use `getenv()` and `setenv()` to access environment variables
#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
int main(void) {
    // 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
# setenv() — set an environment variable

```c
#include <stdlib.h>

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

- adds \texttt{name=value} to environment variable array
- if \texttt{name} in array, value changed if \texttt{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

https://www.cse.unsw.edu.au/~cs1521/23T1/COMP1521_23T1 — Processes
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 programs 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 programs, e.g
  - timezone (TZ)
  - user’s preferred language (LANG)
  - directories to search for programs (PATH)
  - user’s home directory (HOME)
Example: 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
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 memory
- information about open files (and other results of system calls)

On Unix/Linux:

- each process had a unique process ID, or PID: a positive integer, type `pid_t`, defined in `<unistd.h>`
- PID 1: `init`, used to boot the system.
- low-numbered processes usually system-related, started at boot
  - ... but PIDs are recycled, so this isn’t always true
- some parts of the operating system may appear to run as processes
  - many Unix-like systems use PID 0 for the operating system
Each process has a *parent process*.

- initially, the process that created it;
- if a process’ parent terminates, its parent becomes *init* (PID 1)

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

- `sh` ... creating processes via object-file name
- `ps` ... showing process information
- `w` ... showing per-user process information
- `top` ... showing high-cpu-usage process information
- `kill` ... sending a signal to a process
Multi-Tasking

On a typical modern operating system...

- multiple processes are active “simultaneously” (*multi-tasking*)
- operating systems provides a virtual machine to each process:
  - 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,

- a process uses the CPU, until it is *preempted* or exits;
- then, another process uses the CPU, until it too is preempted.
- eventually, the first process will get another run on the CPU.
Multi-tasking

Overall impression: three programs running simultaneously. (In practice, these time divisions are imperceptibly small!)
What can cause a process to be preempted?

- it ran “long enough”, and the OS replaces it by a waiting process
- it needs to wait for input, output, or other some other operation

On preemption...

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

Which process runs next? The *scheduler* answers this. 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
Unix/Linux Processes

Environment for processes running on Unix/Linux systems

```
argc, argv, envp, uid, gid, ...

Process

stdin (fd:0)  stdout (fd:1)
            stderr (fd:2)

return status
(0 = ok, !0 = error)
```
Process-related Unix/Linux Functions/System Calls

Process information:
- getpid() ... get process ID
- getppid() ... get parent process ID
- getpgid() ... get process group ID

Creating processes:
- system(), popen() ... create a new process via a shell - convenient but major security risk
- posix_spawn() ... create a new process.
- fork() vfork() ... duplicate current process. (do not use in new code)
- exec() family ... replace current process.

Destroying processes:
- exit() ... terminate current process, see also
  - _exit() ... terminate immediately
    atexit functions not called, stdio buffers not flushed
- waitpid() ... wait for state change in child process
exec() family - replace yourself

```c
#include <unistd.h>

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

- Run another program in place of the current process:
  - `file`: an executable — either a binary, or script starting with `#!
  - `argv`: arguments to pass to new program
- Most of the current process is re-initialized:
  - e.g. new address space is created - all variables lost
- open file descriptors survive
  - e.g, stdin & stdout remain the same
- PID unchanged
- if successful, exec does not return ... where would it return to? 
- on error, returns -1 and sets `errno`
Example: using `exec()`

```c
int main(void) {
    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
$`

https://www.cse.unsw.edu.au/~cs1521/23T1/
fork() — clone yourself (OBSOLETE)

```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

Both child and parent return from `fork()` call... how do we tell them apart?

- in the child, `fork()` returns 0
- in the parent, `fork()` returns the pid of the child
- if the system call failed, `fork()` returns -1

Child inherits copies of parent’s address space, open file descriptors, ...

Do not use in new code! Use `posix_spawn()` instead.

`fork()` appears simple, but is prone to subtle bugs
Example: using fork() (OBSOLETE)

```c
// 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.
$
waitpid() — wait for a process to change state

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

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

- `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 specified process

```c
pid_t wait(int *wstatus);
```

- equivalent to `waitpid(-1, &status, 0)`
- pauses until any child processes terminates.
waitpid() — wait for a process to change state

```c
pid_t waitpid(pid_t pid, int *wstatus, int 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.
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

https://www.cse.unsw.edu.au/~cs1521/23T1/
Example: one of the dangers of fork - a fork bomb

```c
#include <stdio.h>
#include <unistd.h>
int main(void) {
    // creates $2^{10} = 1024$ processes
    // which all print fork bomb then exit
    for (int i = 0; i < 10; i++) {
        fork();
    }
    printf("fork bomb\n");
    return 0;
}
```

source code for fork_bomb.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 extremely dangerous —
very brittle; 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
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 a new process.

- **path**: path to the process to run
- **argv**: arguments to pass to new program
- **envp**: environment to pass to new program
- **pid**: returns process id of new program
- **file_actions**: specifies *file actions* to be performed before running program
  - can be used to redirect *stdin*, *stdout* to file or pipe
- **attrp**: specifies attributes for new process
  - not used/covered in COMP1521
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);
```

Example: `posix_spawn()` versus `system()`

```c
source code for spawn.c
```

Running `ls -ld` via `posix_spawn()`

```c

pid_t pid;

extern char **environ;

if (posix_spawn(&pid, "/bin/ls", NULL, NULL, ls_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/ls exit status was %d\n", exit_status);
```

Running `ls -ld` via `system()`

```c
system("ls -ld");
```
getpid(), getppid() — get process IDs

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

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

g getpid returns the process ID of the current process.

g getppid returns the process ID of the current process’ parent.
Aside: Zombie Processes (advanced)

A process cannot terminate until its parent is notified. - notification is via wait/waitpid or SIGCHLD signal

**Zombie process** = exiting process waiting for parent to handle notification

- parent processes which don’t handle notification create long-term zombie processes
  - wastes some operating system resources

**Orphan process** = a process whose parent has exited

- when parent exits, orphan assigned PID 1 (*init*) as its parent
- *init* always accepts notifications of child terminations
# exit() — terminate yourself

```c
#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)`

```c
void _exit(int status);

- terminates current process without triggering functions registered as `atexit()`
- stdio buffers not flushed
```
Pipe is a unidirectional byte stream provided by the 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 file descriptors can be used with stdio via `fdopen`. 
popen() — a convenient but unsafe way to set up pipe

```c
#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
Example: capturing process output with `popen()`

```c
// 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("");
    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
```

source code for `read_popen.c`

https://www.cse.unsw.edu.au/~cs1521/23T1/COMP1521_23T1_-_Processes
Example: sending input to a process with popen()

```c
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 only

  // 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;
}
```

Source code for `write_popen.c`

https://www.cse.unsw.edu.au/~cs1521/23T1/COMP1521-23T1---Processes
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 and 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]