# COMP1521 25T2

## Week 8 Lecture 2

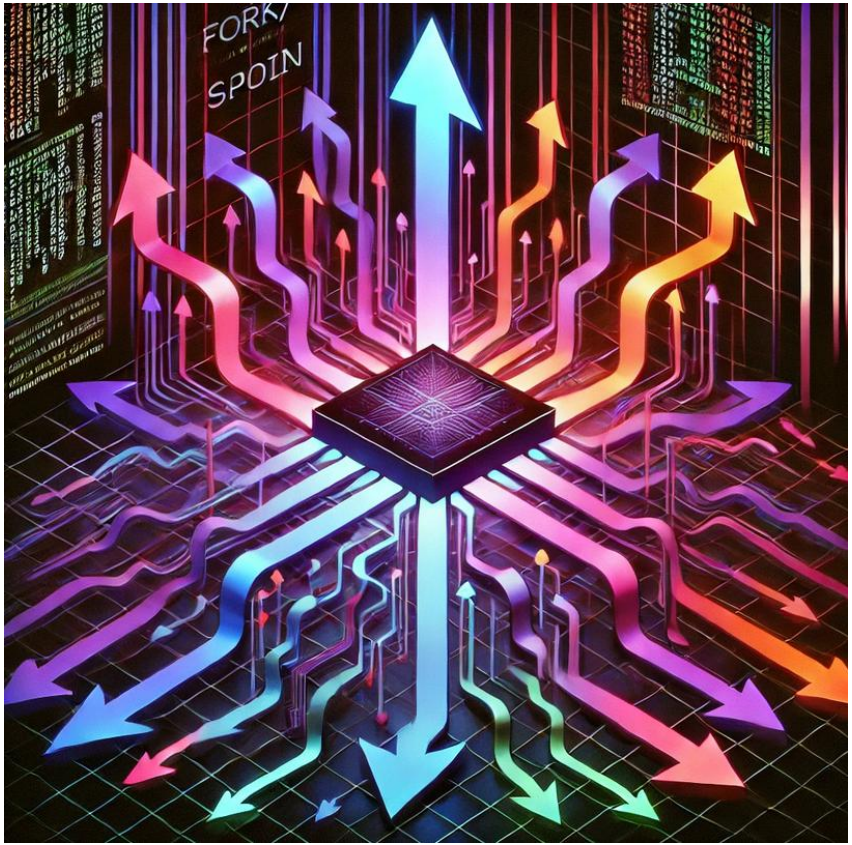
# Processes

Adapted from Angela Finlayson, Abiram Nadarajah,
Hammond Pearce,
Andrew Taylor and John Shepherd's slides

# Today's Lecture

- Unicode
  - Recap

- Processes
  - What are they?
  - Environment Variables
  - System Calls + Functions
    - execv, fork, wait
    - posix_spawn



# UNICODE

- UNICODE is maintained by the Unicode Consortium
- The goal of UNICODE is to create a single encoding that can represent all of the characters in all of the languages in the world.
- There are currently 149,878 characters in UNICODE.
- https://en.wikipedia.org/wiki/List_of_Unicode_characters

# UTF-32: Example

A → U+0041 → 0b00000000000000000000000001000001
€ → U+20AC → 0b00000000000000000010000010101100
字 → U+5B57 → 0b00000000000000000101101101010111
😀 → U+1F600 → 0b00000000000000011111011000000000

U+XXXX is the representation of a raw UNICODE code point

- code points are always at least 4 hex digits.
- 4 digit code points are on the 0th plane
- The 5th digit (if there is one) is the plane number

# UTF-8

- Goal of UTF-8 to increase efficiency
  - Waste less bits!
- Use variable width encoding
  - Why use 4 bytes for every character if we don’t have to?
- Unicode has the most common characters in the first planes
  - These common characters should use less bits!

# UTF-8 Layout

| #bytes | #bits | Byte 1 | Byte 2 | Byte 3 | Byte 4 |
|---|---|---|---|---|---|
| 1 | 7 | 0xxxxxxx | - | - | - |
| 2 | 11 | 110xxxxx | 10xxxxxx | - | - |
| 3 | 16 | 1110xxxx | 10xxxxxx | 10xxxxxx | - |
| 4 | 21 | 11110xxx | 10xxxxxx | 10xxxxxx | 10xxxxxx |

- A single UTF-8 character can be anywhere from 1 to 4 bytes long
- Exercise: How many UTF-8 encoded characters would this represent
  - 11010111 10101111 11101101 10111100 10001011 01001101

# UTF-32 vs UTF-8

“Hello 思语” ==

```
0x00000068
0x00000065
0x0000006c
0x0000006c
0x0000006f
0x00000020
0x0000601D
0x00008BED
```

8 x 4 = 32 bytes

“Hello 思语” ==

```
0x68
0x65
0x6c
0x6c
0x6f
0x20
0xE6809D
0xE8AFAD
```

12 bytes only

# Conversion to UTF-8 (1/2)

€ (U+20AC)

- Convert to UTF-32
  0x000020AC
  0b00000000000000000010000010101100
- Remove leading 0s
  0b10000010101100
- Split into 6 bit chunks
  `0b 10 000010 101100`
- Match with appropriate multi-byte encoding
  `0b 11100010 10000010 10101100`
- In hex: `0xE282AC`

# UTF-8: More Examples

A → U+0041

€ → U+20AC

字 → U+5B57

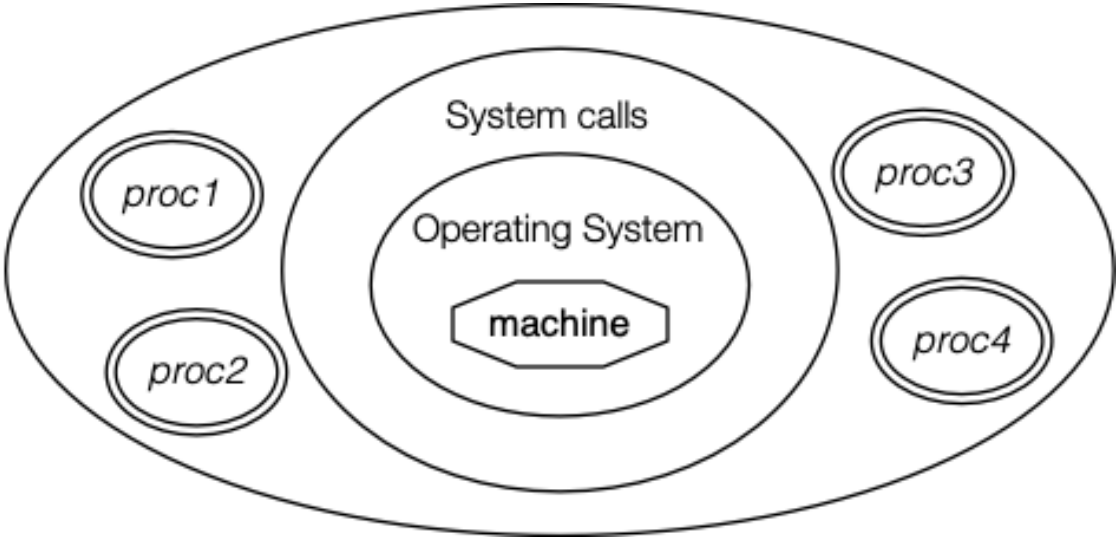
😀 → U+1F600

# Summary of UTF-8

- Compact, but not minimal encoding
- ASCII is a subset of UTF-8 - complete backwards compatibility!
- No byte of multi-byte UTF-8 encoding is valid ASCII
- No byte of multi-byte UTF-8 encoding is 0
  - can still use UTF-8 in null-terminated strings.
- 0x2F (ASCII /) and 0x00 can not appear in multi-byte characters
  - hence can use UTF-8 for Linux/Unix filenames
- C programs can treat UTF-8 similarly to ASCII
  - Beware: number of bytes in UTF-8 string != number of characters.

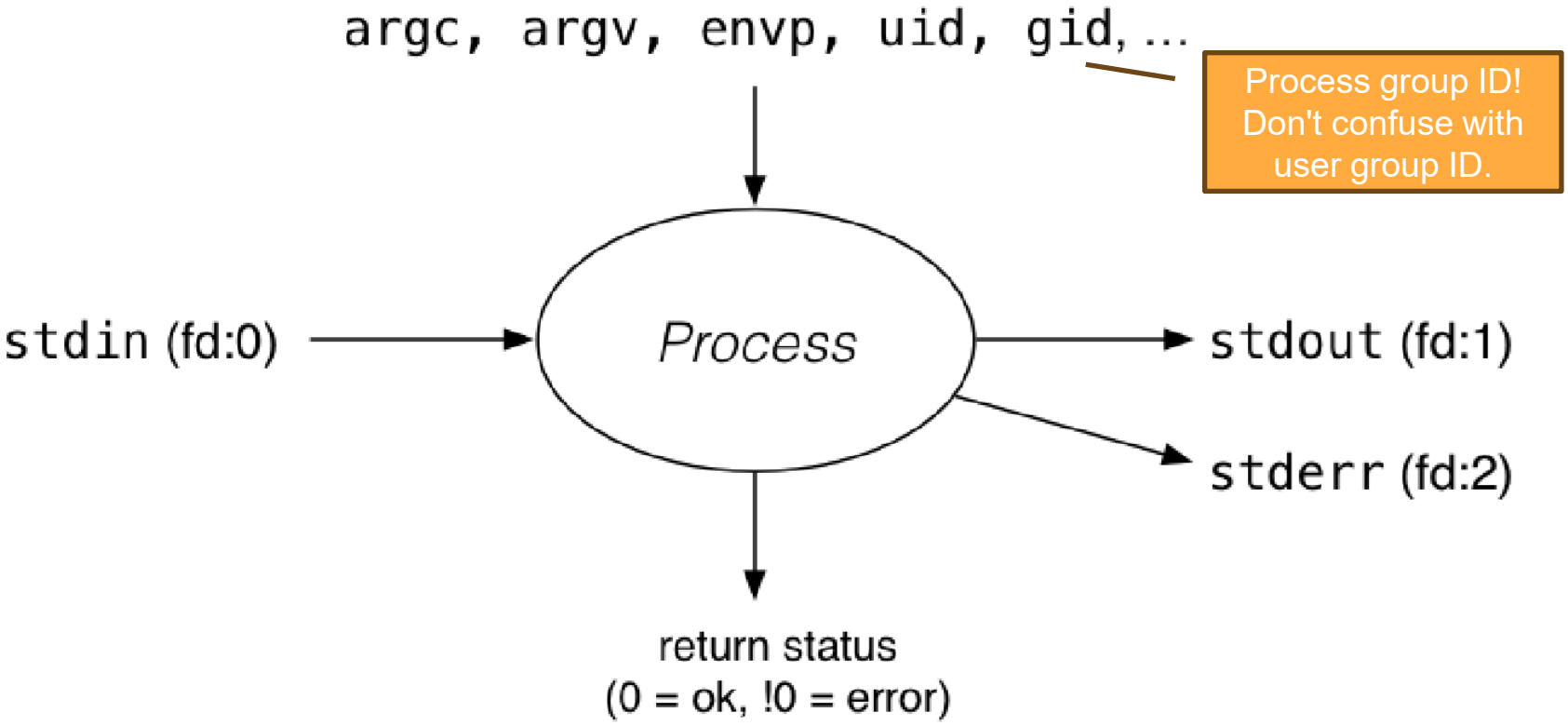
# Processes

# A computer process




- A process is a program running in an environment
- The operating system manages starting, stopping processes

# Environment for Unix/Linux Processes



# Processes

- A process is an instance of an executing program.
- Each process has an execution state, defined by…
  - Current CPU register values
  - Current memory content
  - Information about open files (and other results of system calls)

# Processes on Unix/Linux

- Each process has 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 OS may appear to run as processes
  - many Unix-like systems use PID 0 for the operating system

# Parent Processes

- Each process has a parent process.
  - initially, the process that created it;
  - if a process' parent terminates, its parent becomes init (PID 1)

- A process may have child processes
  - These are processes that it created

# syscalls to get info 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
- More details: man 2 getpid
- Not used in this course: `getpgid()` … get process group ID

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

```c
#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 tools

Unix provides a range of tools for manipulating processes

Commands:

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

# Environment variables

- 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
    - They perhaps add/edit the value of specific environment variables

# 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)

# Environment variables: code

- When run, a program is passed a set of environment variables:
  - Array of strings of the form name=value, terminated with NULL.
  - Access via global variable **`environ`**

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

Demo: environ.c

# Environment variables: better code

Many C implementations also provide as 3rd parameter to main:

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

Best method: Access using **`getenv(...)`** and **`setenv(...)`**

# getenv() - get an environment variable

```c
#include <stdlib.h>
char *getenv(const char *name);
```

- Reads value from environment variable array by name
- if name is not in the array, returns NULL

Demo: get_status.c

# setenv() - set an environment variable

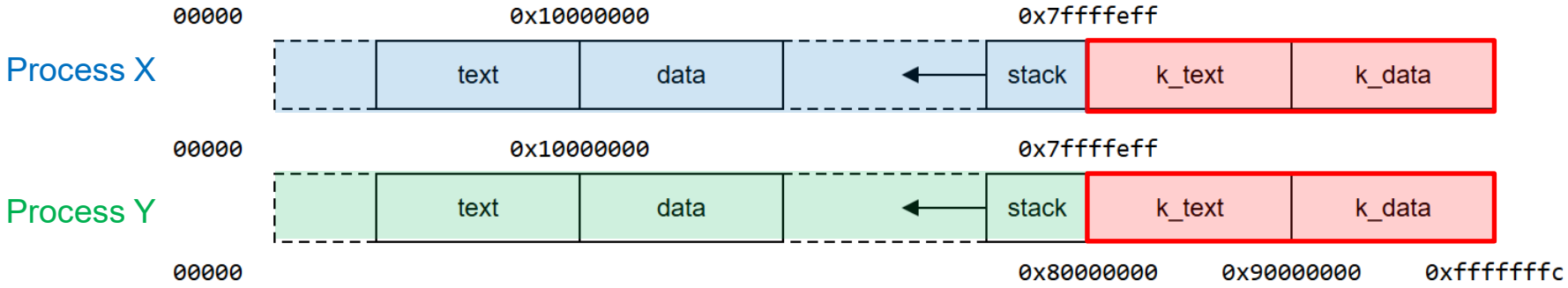
```
#include <stdlib.h>
int setenv(const char *name, const char *value, int overwrite);
```

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

Returns 0 if success, or -1 if error (error stored in *errno*)

# Multi-Tasking

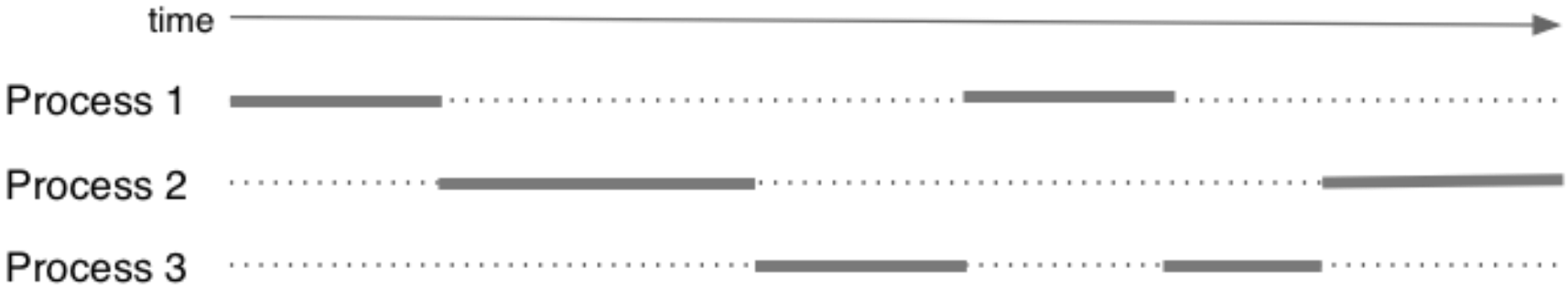
- On a typical modern operating system…
  - Multiple processes are active “simultaneously” (multi-tasking)
  - Operating systems provides a virtual machine to each process:
  - Each process executes as if it is the only process running
  - e.g. each process has its own address space



# Multi-Tasking (cont.)

- 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 (cont.) (cont.)




Overall impression: three programs running simultaneously.
(In practice, these time divisions are imperceptibly small!)

# Preemption — When? How?

- 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 share 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

# Process-related Unix/Linux Functions/syscalls

- 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.
    - (actually, “modern” **`fork()`** is actually **`clone()`**… sshhhhh)
  - **`exec()`** family … replace current process.

# Process-related Unix/Linux Functions/syscalls

- Destroying processes:
  - **`exit()`** … terminate current process, see also
  - **`_exit()`** … terminate immediately
    - (**`atexit`** functions not called, stdio buffers not flushed)
  - **`kill()`** … send signal to a process
- Monitoring changes:
  - **`waitpid()`** … wait for state change in child process

# exec() family - replace yourself

```c
#include <unistd.h>
int execv(const char *file, char *const argv[]);
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

# exec() family - replace yourself

```c
#include <unistd.h>
int execv(const char *file, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

- 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");
```

```
$ dcc exec.c
$ a.out
good-bye cruel world
$
```

Demo: exec.c

# fork() — clone yourself

```c
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

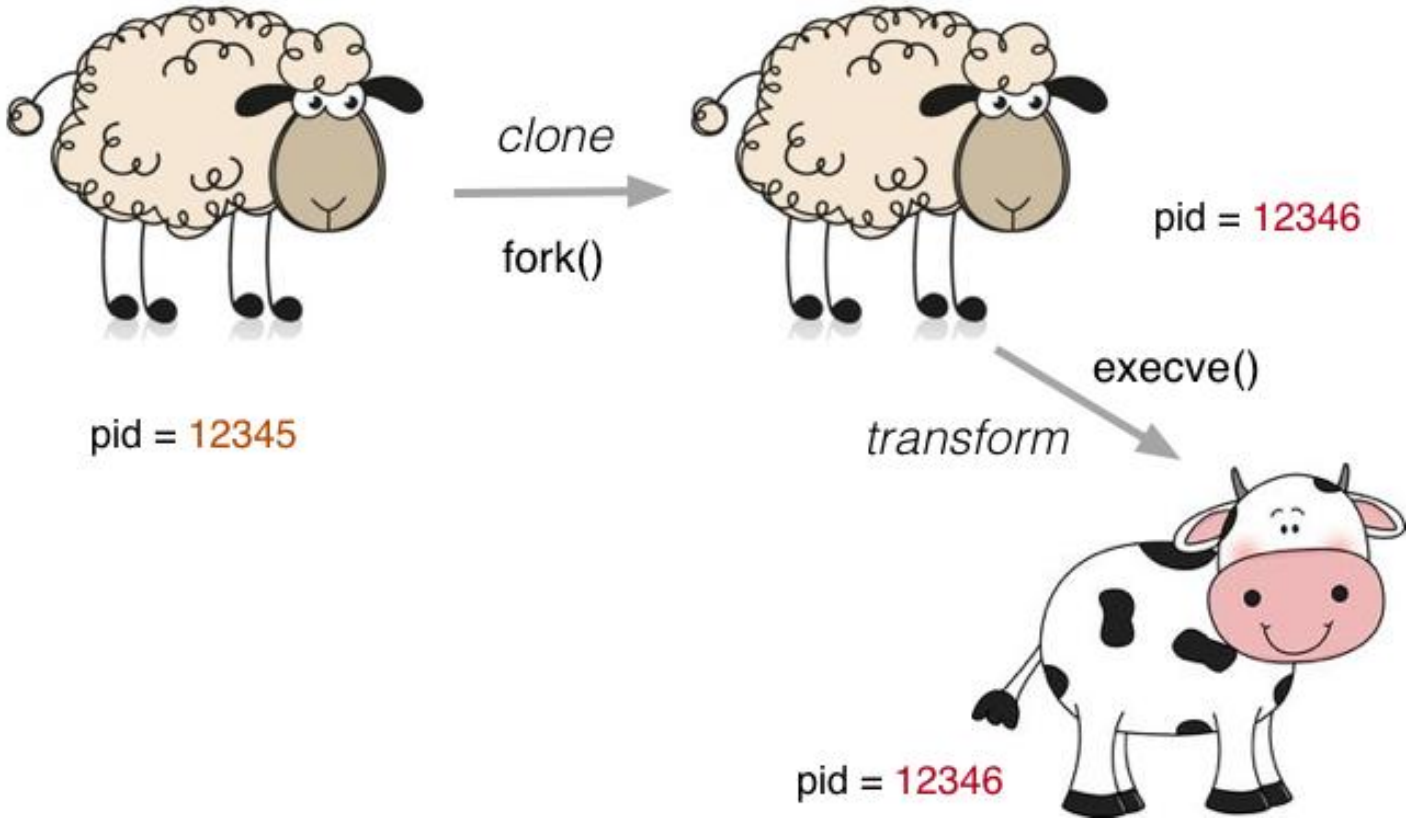
- Creates new process by duplicating the calling process.
  - New process is the child; Calling process is the parent
- Both child and parent return from fork() call… how to distinguish?
  - In the child, fork() returns 0
  - In the parent, fork() returns the pid of the child
  - If the system call failed, fork() returns -1
- Child inherits copies of parent’s address space, open files …

# Example: using fork()

```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);
}
```

Demo: fork.c

# Fork and Exec Together!



# Fork has some dangers, e.g. 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;
}
```

# waitpid() — wait for process to change state

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

- **`wstatus`** 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: 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);
}
```

Demo: fork_exec.c

# system(): convenient but unsafe

```c
#include <stdlib.h>
int system(const char *command)
```

Creates another process and
runs command via /bin/sh.
Waits for command to finish and returns exit status

# system() — convenient but risky

- Convenient … but extremely dangerous —
  - very brittle; highly vulnerable to security exploits
  - https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=OS+Command+Injection
  - 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;
```

Demo: system.c

# Making Processes

- Old-fashioned way **fork()** then **exec()**
  - **fork()** duplicates the current process (parent+child)
  - **exec()** “overwrites” the current process (run by child)
- New, standard way **posix_spawn()**

# 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[]);
```

- pid: returns process id of new program
- path: path to the program to run
- file_actions: specifies file actions to be performed before running the program
    - can be used to redirect stdin, stdout to file or pipe

# posix_spawn() — Run a new process

```c
#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[]);
```

- attrp: specifies attributes for new process (not covered in COMP1521)
- argv: arguments to pass to new program
- envp: environment to pass to new program
- can also use **posix_spawnp** which searches PATH

# Example: 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
int ret = posix_spawn(&pid, "/bin/date", NULL, NULL, date_argv, environ);
if (ret != 0) {
    errno = ret; //posix_spawn returns error code, does not set errno
    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);
```

Demo: spawn.c

# Example: posix_spawn() versus system()

## Running ls -ld via posix_spawn()

```
char *ls_argv[2] = {"/bin/ls", "-ld", NULL};
pid_t pid; int ret;
extern char **environ;
if((ret = posix_spawn(&pid, "/bin/ls", NULL,
    NULL, ls_argv, environ)) != 0)
{
    errno = ret; perror("spawn"); exit(1);
}
int exit_status;
if (waitpid(pid, &exit_status, 0) == -1) {
    perror("waitpid"); exit(1);
}
```

## Running ls -ld via system()

```
system("ls -ld");
```

Demo: lsld_spawn.c
lsld_system.c

# Setting environment var for child process

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

Demo: [get|set]_status.c

# Change behaviour with an environment var

```
pid_t pid;
char *date_argv[] = { "/bin/date", NULL };
char *date_environment[] = { "TZ=Australia/Perth", NULL };
// print time in Perth
int ret = posix_spawn(&pid, "/bin/date", NULL, NULL, date_argv,
date_environment);
if (ret != 0) {
    errno = ret; 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);
```

Demo: spawn_environment.c

# 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)

# _exit() — terminate yourself without …

```c
#include <stdlib.h>

void _exit(int status);
```

- terminates current process without triggering functions registered as atexit()
- stdio buffers not flushed
- sometimes used by children of fork() when exiting

# What we learnt Today

- Recap on UTF-8 Encoding

- Processes
  - Environment Variables
  - system(...), fork(...), execv(...), posix_spawn(...)
  - waitpid(...)
  - exit(...), _exit(...)

# Next Lecture

- Inter Process Communication
  - Pipes!
- Concurrency
- Parallelism
- Threads

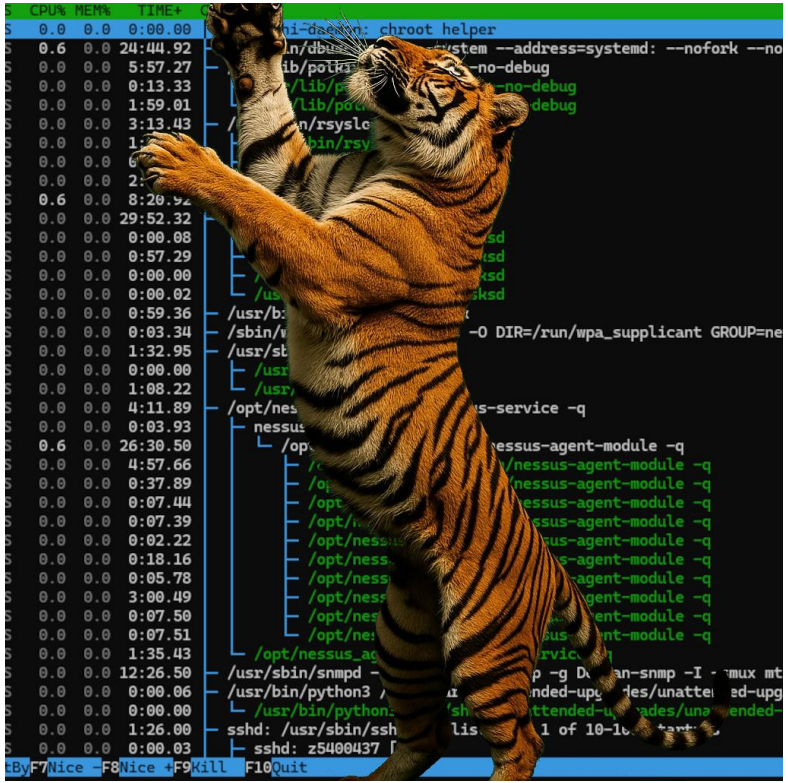
# Reach Out

Content Related Questions:

Forum

Admin related Questions email:

cs1521@cse.unsw.edu.au



# Student Support | I Need Help With…

**My Feelings and Mental Health**
Managing Low Mood, Unusual Feelings & Depression

- **Mental Health Connect** — student.unsw.edu.au/**counselling** Telehealth
- **Mind HUB** — student.unsw.edu.au/**mind-hub** Online Self-Help Resources

**In Australia Call Afterhours UNSW Mental Health Support Line** — 1300 787 026 5pm-9am

**Outside Australia Afterhours 24-hour Medibank Hotline** — +61 (2) 8905 0307

**Uni and Life Pressures**
Stress, Financial, Visas, Accommodation & More

- **Student Support** — student.unsw.edu.au/**advisors**
- **Indigenous Student Support**

**Reporting Sexual Assault/Harassment**

- **Equity Diversity and Inclusion (EDI)** — edi.unsw.edu.au/**sexual-misconduct**

**Educational Adjustments**
To Manage my Studies and Disability / Health Condition

- **Equitable Learning Service (ELS)** — student.unsw.edu.au/**els**

**Academic and Study Skills**

- **Academic Language Skills** — student.unsw.edu.au/**skills**

**Special Consideration**
Because Life Impacts our Studies and Exams

- Special Consideration — student.unsw.edu.au/**special-consideration**