COMP1521 26T1

Week 8 Lecture 2

Unicode and Processes

Announcements

- **Assignment 1** Marks are available
  - We are still running plagiarism checking
- **Week 10 lab** will be a practice exam for in person labs only
  - We recommend attending other timeslot if usually in online
  - Not worth marks but recommended
  - Normal lab exercises for marks
- Final Exam Draft Timetable out
  - Note: **No course website** available in final exam
  - MIPS docs and cheat sheets like in weekly tests will be
  - Command line man too!

Announcements

Test 7 (C bitwise Operators and Floating Point Representation) **is** due tomorrow: **Thursday 9pm**

Test 8 is out tomorrow. **Topic** is: reading and writing files

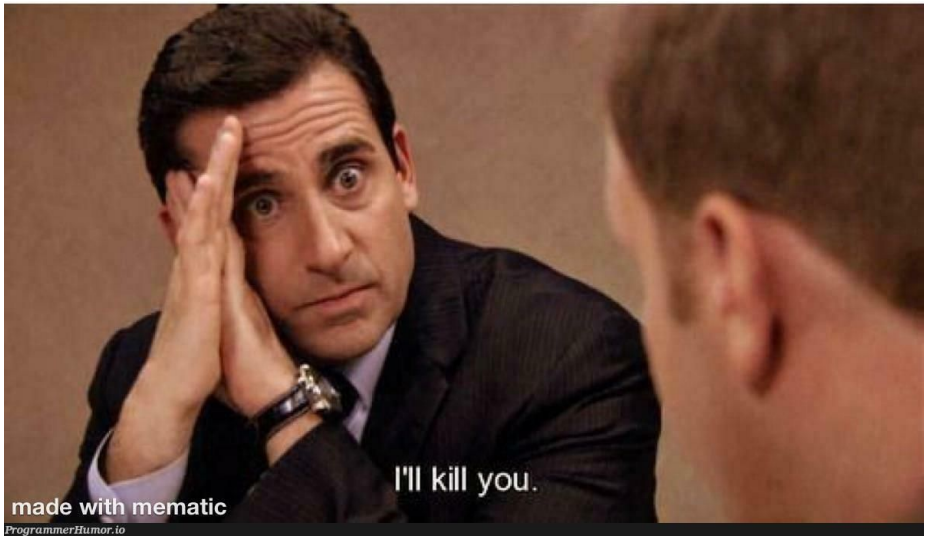
Assignment 2 is out now, including walkthrough video.

- Due **Friday 6pm Week 10**!
- Style worth 20%
- See Week 7 Lecture 2 for basic_stdio.c example to help get started.

Today's Lecture

- Unicode 👍
  - Recap
  - Code Examples
- Processes
  - What are they?
  - Environment Variables
  - System Calls + Functions
    - execv, fork

When a Linux process stops responding



UNICODE

- There are currently 159,800 characters in UNICODE.
- https://en.wikipedia.org/wiki/List_of_Unicode_characters

Terminology	Example
Character	
Code point	U+1F427
Encoding	bytes (e.g. UTF-8: 0x F0 9F 90 A7)

Storing UNICODE characters: UTF-32

- Code points range from **0x0000** to **0x10FFFF**
  - So we need at least 21 bits to represent them.
- We can use 32 bits to represent a single character.
- **UTF-32** is a fixed width encoding
  - Simply take the UNICODE code point and store it in 32 bits.

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

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?
- The most common characters should use less bits!

UTF-8 Layout

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

- A single **UTF-8** character can be anywhere from **1 to 4 bytes** long
- Exercise: How many UTF-8 encoded characters would this represent
    - **110**10111 **10**101111 **1110**1110 **10**111100 **10**001011 **0**1001101

UTF-8 Layout

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

- Exercise: Is this legal **UTF-8**?
  - 11101011 10101111 01001101 10111100 11111110 11110000

UTF-8 Layout

- All ASCII characters can be stored in:
  - 1 byte and it is backwards compatible
- Every UNICODE character can be stored in
  - 4 bytes, which is the same as UTF-32 in the worst case

Exercise: Convert to UTF-8

 → U+1F9A5

UTF-8: More Examples

A → U+0041 → 0b01000001 → 0x41

€ → U+20AC → 0b10 000010 101100

→ 0b11100010 10000010 10101100

→ 0xE282AC

字 → U+5B57 → 0b101 101101 010111

→ 0b11100101 10101101 10010111

→ 0xE5AD97

😀 → U+1F600 → 0b 11111 011000 000000

→ 0b11110000 10011111 10011000 10000000

→ 0xF09F9880

Exercise: Programming with UTF-8

Suppose I have char c, which is a UTF-8 byte:

- How can I tell if it is
  - the first byte of a **1 byte** character?
  - the first byte of a **4 byte** character?
  - How can I tell if it is a continuing byte?

Writing C that uses Unicode

unicode_recap.c

utf8_strlen.c

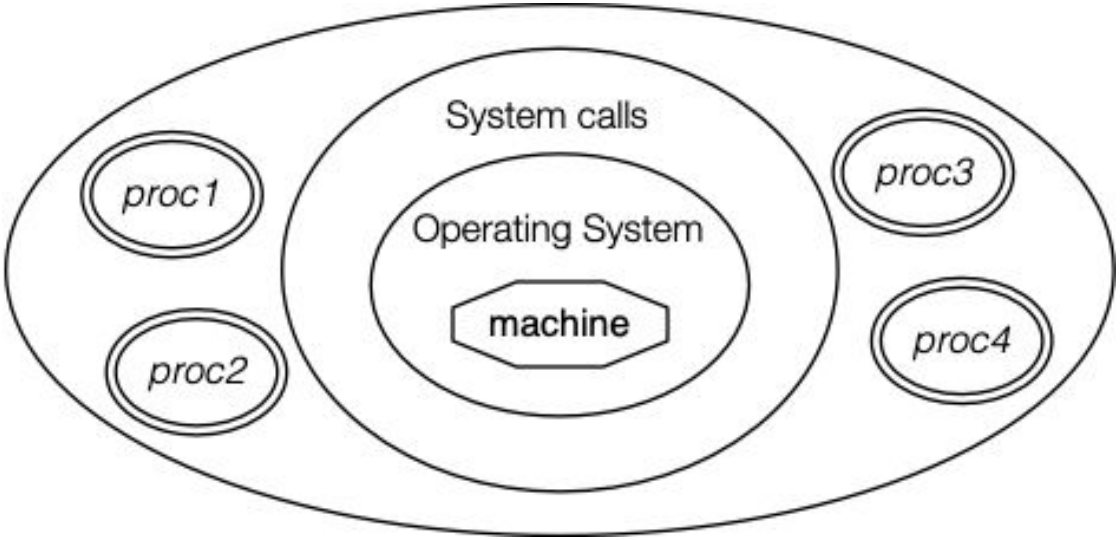
trojan3.c

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 store 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**

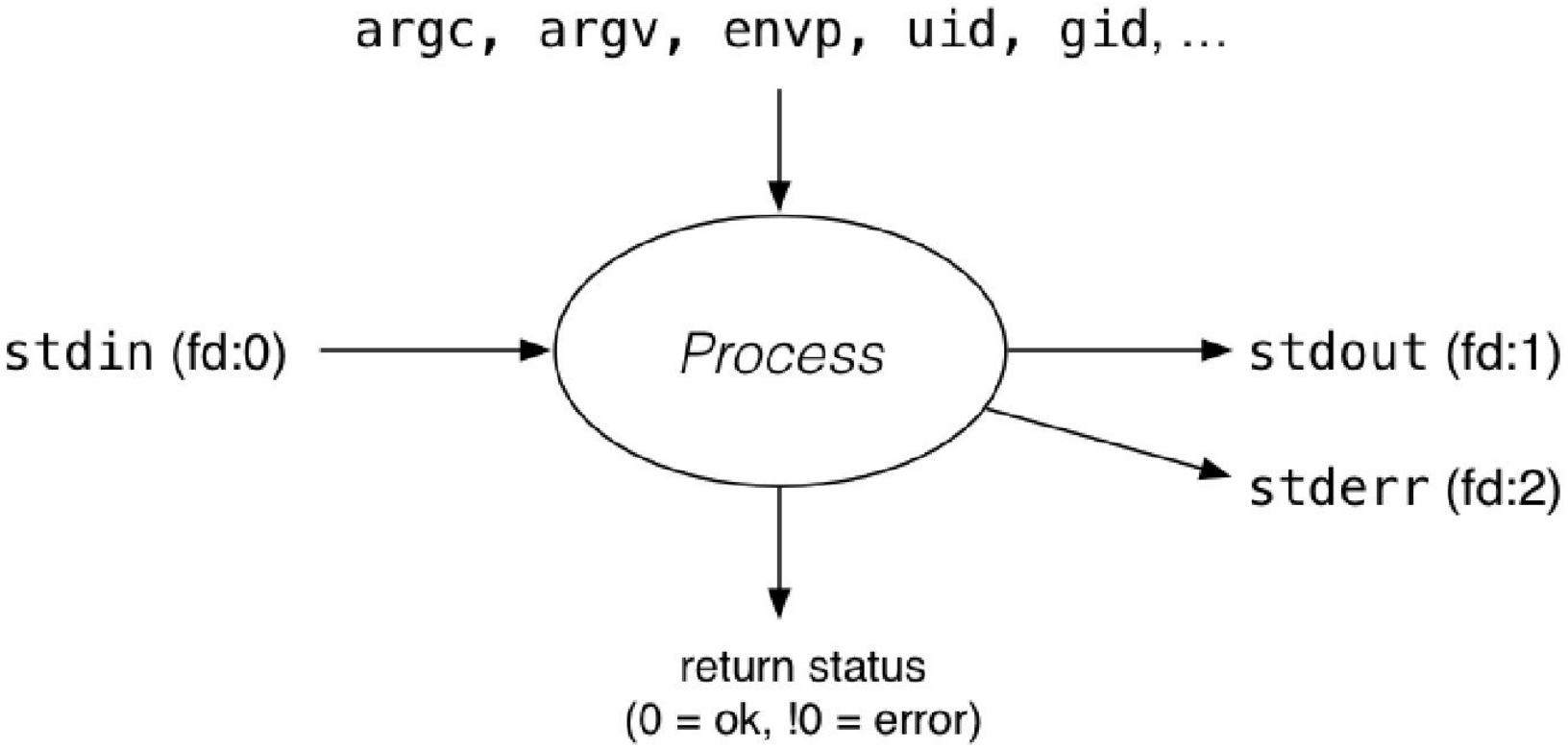
Processes

A computer process



- A process is an instance of 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 values of CPU registers
  - Current contents of its memory
  - 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
  - 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
- For 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
  - You can type `env` on the command line to see them all
  - 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
    - 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

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

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

This will not always work.

getenv() - get an environment variable

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

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

Example: to get home_directory (aka ~)

```
printf("%s", getenv("HOME"));
```

Demo: get_env.c, 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*)

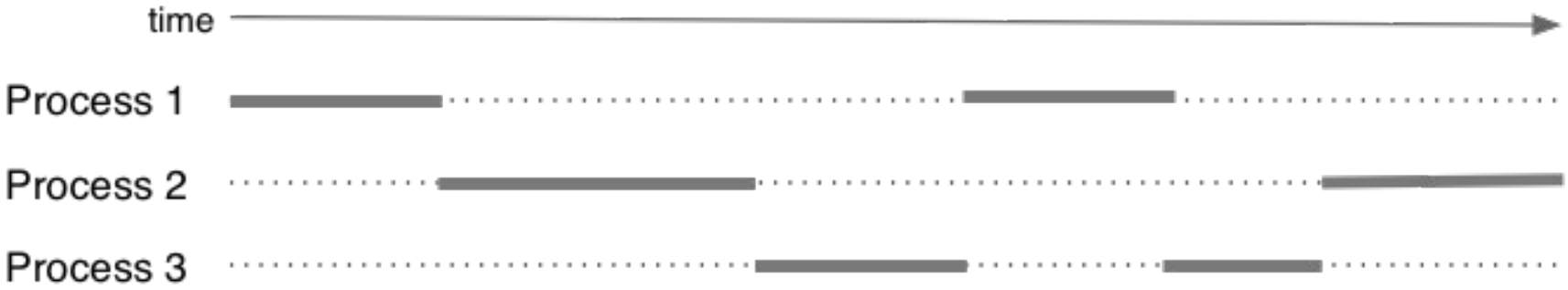
Multi-Tasking

- 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** (N bytes, addressed 0..N-1)

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 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

Process-related Unix/Linux Functions/syscalls

- **`execve()`** … replace current process.
- Creating processes:
  - **`system(), popen()`** … create a new process via a shell
    - convenient but major security risk
  - **`fork()`** … duplicate current process
  - **`posix_spawn()`** … create a new 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

execve(2)

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

- 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
  - **envp**: environment variables to pass to new program
- if successful, exec does not return … where would it return to?
  - on error, returns -1 and sets errno

execve(2) family - replace yourself

- 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

Example: using execve()

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

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

Demo: exec.c

fork(2) — 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()

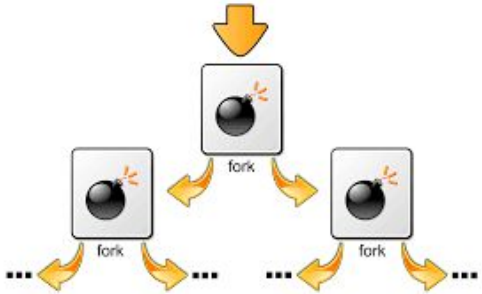
```c
// fork creates 2 identical copies of program
// only return value is different
pid_t pid = fork();
if (pid == -1) {
    perror("fork"); // print why the fork failed
} else if (pid == 0) {
    printf("I am the child because fork() returned %d.\n", pid);
} else {
    printf("I am the parent because fork() returned %d.\n", pid);
}
```

Demo: fork.c, fork2.c fork_ex1.c

Fork dangers, e.g. a fork bomb

WARNING!
DON'T EVEN THINK ABOUT IT!

```c
#include <stdio.h>
#include <unistd.h>
// DO NOT RUN CODE LIKE THIS!!!!!
// creates 2 ** 10 = 1024 processes
int main(void) {
    for(int i = 0; i < 10; i++) {
        printf("Fork Bomb: %d forked %d\n", getpid(), fork());
        sleep(1);
    }
    return 0;
}
```



What we learnt Today

- Unicode
  - UTF-8 Encoding recap and coding example
- Processes
  - Environment Variables
  - execve, fork

Next Lecture

- Processes
  - wait
  - posix_spawn
- Inter Process Communication
  - pipes

Feedback Please!

Your feedback is valuable!

If you have any feedback from today's lecture, please follow the link below or use the QR Code.

Please remember to keep your feedback constructive, so I can action it and improve your learning experience.

https://forms.office.com/r/ftBjaPW2Cn

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

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**