Project 1: Unix shell

- **Shells are one of the most basic utilities in many OSes**
  - The program that lets you run other programs
  - Also a primitive but powerful scripting language
- **In Unix, a shell is just an ordinary process**
  - Even non-administrators can develop and run their own shells
- **This project focuses on core function of spawning processes**
  - Run programs, redirect input/output, create pipelines
- **Things the project doesn't require:**
  - Programming features of real shells (e.g., conditionals, variables)
  - Job control, signal handling, any fancy terminal handling

UNIX file system calls

- ***System calls* are requests from processes to the OS kernel**
  - The *kernel* (e.g., linux) is a program run in a privileged CPU mode
  - syscall ≈ library call to a function with greater hardware access
- **Original UNIX paper is a great reference on core system calls**
- **Applications "open" files (or devices) by name**
  - I/O happens through open files
- `int open(char *path, int flags, /*int mode*/...);`
  - `flags`: `O_RDONLY`, `O_WRONLY`, `O_RDWR`
  - `O_CREAT`: create the file if non-existent
  - `O_EXCL`: (w. `O_CREAT`) create if file exists already
  - `O_TRUNC`: Truncate the file
  - `O_APPEND`: Start writing from end of file
  - `mode`: final argument with `O_CREAT`
- **Returns file descriptor—used for all I/O to file**

Error returns

- **What if `open` fails? Returns -1 (invalid fd)**
- **Most system calls return -1 on failure**
  - Specific kind of error in global int `errno`
  - In retrospect, bad design decision for threads/modularity
- `#include <sys/errno.h>` **for possible values**
  - 2 = `ENOENT` "No such file or directory"
  - 13 = `EACCES` "Permission Denied"
- `strerror(errno)` **– translate error number to string**
- `perror` **prints error message to** `stderr`
  - `perror ("initfile");`
    → `"initfile: No such file or directory"`

Operations on file descriptors

- `int read (int fd, void *buf, int nbytes);`
  - Returns number of bytes read
  - Returns 0 bytes at end of file (EOF), or -1 on error
- `int write (int fd, const void *buf, int nbytes);`
  - Returns number of bytes written, -1 on error
- `off_t lseek (int fd, off_t pos, int whence);`
  - `whence`: 0 – start, 1 – current, 2 – end
    - ▹ Returns previous file offset, or -1 on error
- `int close (int fd);`
  - Free file descriptor number (can be re-used by subsequent open)
  - May flush pending writes (NFS) or send EOF (pipes/TCP)

File descriptor numbers

- **File descriptors are inherited by processes**
  - When one process spawns another, same fds by default
- **Descriptors 0, 1, and 2 have special meaning**
  - 0 – "standard input" (`stdin` in ANSI C / `std::cin` in C++)
  - 1 – "standard output" (`stdout`, `printf` / `std::cout`)
  - 2 – "standard error" (`stderr`, `perror` / `std::cerr`)
  - Normally all three attached to terminal
- **Example:** `type.cc`
  - Prints the contents of a file to `stdout`

`type.cc`

```
void
typefile(char *filename)
{
  int fd, nread;
  char buf[1024];

  fd = open(filename, O_RDONLY);
  if (fd == -1) {
    perror(filename);
    return;
  }

  while ((nread = read(fd, buf, sizeof (buf))) > 0)
    write(1, buf, nread);

  close (fd);
}
```

- **Can see system calls using strace utility (ktrace on BSD)**

Creating processes

- `int fork (void);`
  - Create new process that is exact copy of current one
  - Returns *process ID* of new process in "parent"
  - Returns 0 in "child"
- `int waitpid (int pid, int *stat, int opt);`
  - `pid` – child process to wait for, or -1 for any
  - `stat` – will contain exit value or signal; See waitpid(2) for macros `WIFEXITED(status)`, `WEXITSTATUS(status)`, ...
  - `opt` – usually 0 or `WNOHANG`
  - Returns process ID or -1 on error

Deleting processes

- `void exit (int status);`
  - Current process ceases to exist
  - `status` shows up in `waitpid` (shifted)
  - By convention, `status` of 0 is success, non-zero error
- `void _exit (int status);`
  - Make the system call while bypassing library features
- `int kill (int pid, int sig);`
  - Sends signal `sig` to process `pid`
  - `SIGTERM` most common value, kills process by default (but application can catch it for "cleanup")
  - `SIGKILL` stronger, kills process always
- **Example:** `forkexit.cc` **(print message and exit in child)**


```
std::ostringstream out;
void
flushout()
{
  std::string output(out.str());
  write(1, output.data(), output.size());
}

int
main()
{
  std::atexit(flushout);
  out << "About to call fork" << std::endl;

  pid_t pid = fork();
  if (pid == 0) {
    std::cout << "Hello from the child" << std::endl;
    exit(0);
  }

  int status;
  waitpid(pid, &status, 0);
  out << "pid " << pid << " status 0x" << std::hex << status << std
}
```



```
std::ostringstream out;
void
flushout()
{
  std::string output(out.str());
  write(1, output.data(), output.size());
}

int
main()
{
  std::atexit(flushout);
  out << "About to call fork" << std::endl;

  pid_t pid = fork();
  if (pid == 0) {
    std::cout << "Hello from the child" << std::endl;
    _exit(0);
  }

  int status;
  waitpid(pid, &status, 0);
  out << "pid " << pid << " status 0x" << std::hex << status << std
}
```


Running programs

- `int execve (char *prog, char **argv, char **envp);`
  - `prog` – full pathname of program to run
  - `argv` – argument vector that gets passed to `main`
    Note: end of vector must be marked by NULL pointer
  - `envp` – environment variables, e.g., `PATH`, `HOME` (also NULL-terminated)
- **Generally called through a wrapper functions**
  - `int execvp (char *prog, char **argv);`
    Search `PATH` for prog, use current environment
  - `int execlp (char *prog, char *arg, ...);`
    List arguments one at a time, finish with `NULL`
- **Examples:** `decrypt1.cc`, `decrypt2.cc`

Manipulating file descriptors

- `int dup2 (int oldfd, int newfd);`
  - Closes `newfd`, if it was a valid descriptor
  - Makes `newfd` an exact copy of `oldfd`
  - Two file descriptors will share same offset (`lseek` on one will affect both)
- `int fcntl (int fd, int cmd, ...)` **– misc fd configuration**
  - `fcntl (fd, F_SETFD, val)` – sets close-on-exec flag
    When `val == 1`, `fd` not inherited by exec'ed programs
  - `fcntl (fd, F_GETFL)` – get misc fd flags
  - `fcntl (fd, F_SETFL, val)` – set misc fd flags

Pipes

- `int pipe (int fds[2]);`
  - Returns two file descriptors in `fds[0]` and `fds[1]`
  - Data written to `fds[1]` will be returned by `read` on `fds[0]`
  - When last copy of `fds[1]` closed, `fds[0]` will return EOF
  - Returns 0 on success, -1 on error
- **Operations on pipes**
  - `read`/`write`/`close` – as with files
  - When `fds[1]` closed, `read(fds[0])` returns 0 bytes
  - When `fds[0]` closed, `write(fds[1])`:
    - Kills process with `SIGPIPE`
    - Or if signal ignored, fails with EPIPE
- **Example:** `decrypt3.cc`

Terminal handling (not for project)

- **Unix provides abstraction of a *terminal* (a.k.a. tty)**
  - Historically serial ports with physical terminals attached
  - Nowadays, emulated terminal on console, or pseudoterminal
  - Lots of vestigial functionality (try `stty olcuc`)
  - OS kernel supports erasing characters, sends whole line to apps
  - Also sends signals when certain keys are pressed (e.g., Ctrl-C)
- **termios(3) lets you control terminal in a portable way**
  - E.g., set "raw" mode to read every character, not wait for lines
- **OS lets a shell foreground, background, and stop jobs**
  - Each process has a *process group ID* (pgid) set by setpgid(2)
  - Only one PG can be foreground on a given tty tcsetpgrp(3)
  - Only processes in foreground PG get signals from Ctrl-C
  - Processes in background get stopped if they try to read from tty

Key C++ library types for project

- `std::vector` – **variable-size array in memory**
  - Use `push_back()` to add to end
  - Use `data()`, `size()` to get pointer and size
- `std::string` **often more convenient, safer than** `char*`
  - `using namespace std::string_literals;` enables `operator""s`: E.g., `std::string s = "hello "s + "world"s;`
  - But system calls take `char*`, available via `s.c_str()`
  - `char*` from `c_str()` no longer valid when `string` destroyed!

```
// Compiles, but produces undefined behavior!
const char *greeting(char *name) {
  return ("Hello "s + name).c_str();
}
```


C++ quiz

- **Is this code okay?**

```
pid_t pid = spawn([devnull,pipefds]{
  close(pipefds[1]);
  dup2(devnull, 2);
}, "gpg", "--passphrase-fd",
    std::to_string(pipefds[0]).c_str(),
    "--pinentry-mode=loopback", "-d", "secret.gpg");
```


C++ quiz

- **Is this code okay?**

```
pid_t pid = spawn([devnull,pipefds]{
  close(pipefds[1]);
  dup2(devnull, 2);
}, "gpg", "--passphrase-fd",
    std::to_string(pipefds[0]).c_str(),
    "--pinentry-mode=loopback", "-d", "secret.gpg");
```

- **Yes: lifetime of temporary objects extends to evaluation of "full expression"—in this case the call to `spawn`**
- **But this produces undefined behavior:**

```
const char *fdstr = std::to_string(pipefds[0]).c_str();
pid_t pid = spawn(..., fdstr, ...); // fdstr invalid
```

- **Use this instead:**

```
std::string fdstr = std::to_string(pipefds[0]);
pid_t pid = spawn(..., fdstr.c_str(), ...);
```


C++ resources

- **C++ one of the hardest languages to learn (so unprincipled)**
  - Huge difference in code written by experts and beginners
  - Learn it well now, can pay dividends for your whole career
- **Try to *understand*, not just pattern match or use trial and error**
- **Useful resources:**
  - `cppreference.com` – assign a search shortcut (e.g., cpp)
  - The ground truth: standards documents
  - Incomprehensible error? Switch between g++ and clang++
  - Good sites for analyzing code: cppinsights.io and godbolt.org

Debugging

- **What if you want to debug code after fork?**
  - In gdb, use `set follow-fork-mode child`
  - Demo: debug `decrypt3`
    - Use `info inferiors` to get PID
    - Use `lsof -p` to see how fds get manipulated
- **Other useful debugging tools:**
  - `strace` – print all syscalls (use `-f` to trace across fork)
  - `ps`, `pstree` print processes your shell has spawned
  - Useful programs for testing your pipe implementation: `cat, tee, yes, head, tail, wc`
  - Compile with `-ggdb -O0`
  - Compile with `-fsanitize=address`
  - Search for memory errors with valgrind

Accessing farmshare machines

- **You may wish do assignments on farmshare machines**
  - Can develop on your own linux box, but test/submit on farmshare
- **2FA requirement prevents SSH public key authentication**
- **I suggest connection multiplexing for passwordless login**
  - Sample extract from `$HOME/.ssh/config`:

```
Host myth.stanford.edu myth??.stanford.edu
  HostKeyAlias myth.stanford.edu
  ControlMaster auto
  ControlPath ~/.ssh/%r@%h:%p.sock
  ControlPersist yes
  User YOUR-SUNET-ID
```

  - First login: `ssh -M myth.stanford.edu` (or `ssh -fNM` ...)
  - Subsequent login: `ssh myth.stanford.edu` (no password)
  - Kill master: `ssh -O exit myth.stanford.edu`
  - May need to delete sockets in `~/.ssh` if your machine reboots