

OPERATING SYSTEMS · HY345

# Build a shell

## From a line of text to a running process

Processes, pipes, redirection, and control flow in `hy345sh`

## PART 1

# Background

What a shell is, and the process model behind it

# You already use a shell

When you open a terminal and type commands, you are talking to a **shell** a command-line program that:

## **READS**

Your input line by line

## **INTERPRETS**

What that text means (command, args, operators)

## **ACTS**

Asks the OS to run programs and manage I/O

Common shells: Bash, Zsh, Ksh. On most systems the default is Bash.

# A shell is a program

Bash is an ordinary executable on disk just like `ls` or `cat`.

```
> which bash
/usr/bin/bash
```



`which bash` → `/usr/bin/bash`. In this exercise you write another such program: `hy345sh`.

# A shell is a translator

You type text. The kernel only understands system calls. The shell sits in between: it parses your line, then asks the kernel to create processes and wire their I/O.



`hy345sh` is a smaller cousin of Bash. Same job, fewer features but the same process model.

# Finding programs: `$PATH`

If the command has **no** slash ( `ls`, not `/usr/bin/ls` or `./a.out` ), the shell does not guess a file on disk. It walks `$PATH`: a colon-separated list of directories, in order, until it finds an executable with that name.

```
> echo $PATH  
/usr/local/sbin:/usr/local/bin:/usr/bin
```



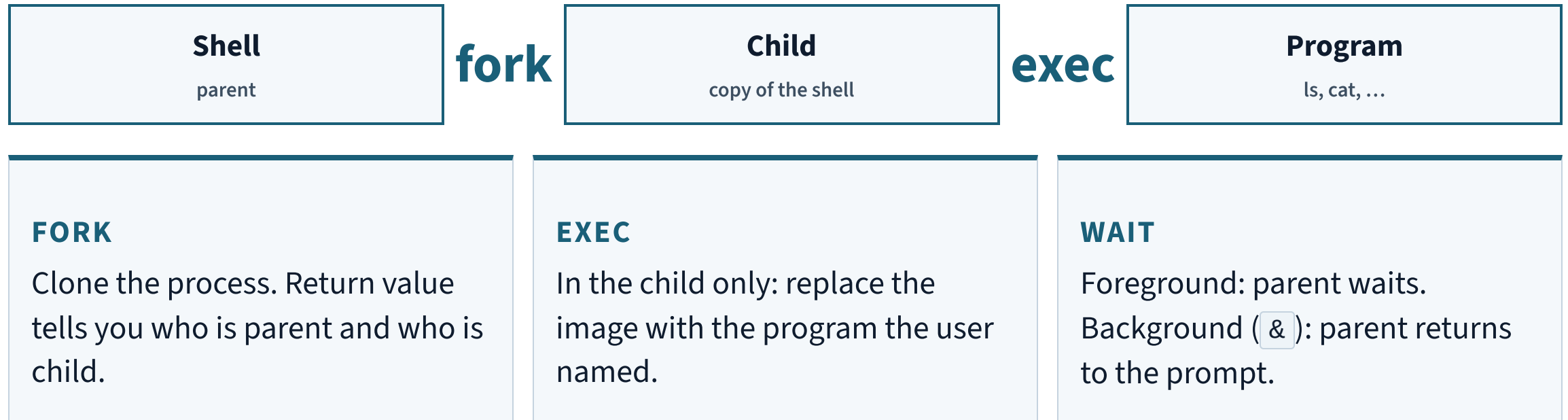
## PART 2

# Useful functions

The syscalls that turn a parsed line into a running process

# The process model

External commands must **not** replace the shell. Pattern:



# Four important calls

Almost every external command in `hy345sh` is some combination of these:

## **`FORK ( )`**

Create a child that is a copy of the shell.

## **`EXEC* ( )`**

In the child: replace that copy with the program to run.

## **`WAIT ( )`**

In the parent: suspend until a child finishes (foreground jobs).

## **`EXIT ( )`**

Terminate a process cleanly and return a status to the parent.

Also useful later: `waitpid`, `pipe`, `dup2`, `setenv` / `getenv`.

# fork()

Creates a new process that is a **duplicate** of the caller (parent): same code, open file descriptors, register state, and variable values at the moment of the call.

**PID < 0**

Fork failed. Handle the error

**PID == 0**

You are in the **child** execution space.

**PID > 0**

You are in the **parent**. `pid` is the child's process id.

# exec() family

Replaces the **current** process image with another program.

## BY PATH / BY NAME

`execl`, `execv`, `execle` — full path.

`execlp`, `execvp`, `execvpe` — search `$PATH`.

## ARGS STYLE

`l` — list of args, `NULL`-terminated.

`v` — `argv[]` vector.

`e` — also pass a custom `envp[]`.

```
int execl (const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char *const envp[]);
int execv (const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

# exec<sub>l</sub> vs exec<sub>v</sub>

Same job different way to pass arguments.

## EXECL — LIST

Fixed arguments as separate parameters. Ends with `NULL`.

Handy when the command is known at compile time.

## EXECV — VECTOR

Arguments in a `char *argv[]` array. Last element is `NULL`.

What you want when args come from parsing user input.

```
/* run: ls -l /tmp */
execl("/bin/ls", "ls", "-l", "/tmp", NULL);

char *argv[] = {"ls", "-l", "/tmp", NULL};
execv("/bin/ls", argv);
```

# Using `execvp` / `execve`

Build an `argv` array, then hand it to a `v`-style call. Same idea for any command.

```
char *argv[] = {"ls", "-l", NULL};

execvp("ls", argv);           /* searches $PATH */
/* or: */
execve("/bin/ls", argv, environ); /* full path + current env */
```

## **EXECVP**

Looks up `ls` in `$PATH`. Typical choice in a shell.

## **EXECVE**

Needs the full path.

# `wait()`

Forces the parent to **suspend** until a child finishes (or is terminated, e.g. by a signal).

## EXIT STATUS

When a child terminates, the OS holds an exit status.  
`wait` / `waitpid` delivers that status to the parent,  
which then continues.

## IN THE SHELL

Foreground job → wait. Background (`&`) → do **not**  
block, return to the prompt and track the job instead.

Prefer `waitpid` when you need a specific child or non-blocking checks for background jobs.

# exit()

Gracefully terminates the calling process: cleans up and releases resources.

## PARENT IS WAITING

The exit status is delivered via `wait` / `waitpid`. The child disappears.

## PARENT IS NOT WAITING

The child becomes a **zombie** until some parent finally waits

# Putting it together

The core of a simple shell is a loop around `fork` → `exec` → `wait`:

```
while(1){                //repeat forever
    command_prompt();    //display command prompt on the screen
    read_command(command,parameters); //read input for terminal

    int pid = fork();    //fork
    if(pid == 0){        //child process,execute command

        execve(command,parameters); //execute the command

    }else if(pid > 0){ //this is the parent process

        waitpid(-1,&status,0); //wait child process to finish its execution
    }else{

        printf("Fork not executed successfully!\n") //handle failed forks
    }
}
```

## PART 3

# Shell requirements

What `hy345sh` must support

# Prompt

Shell name: `hy345sh`. Prompt format:

```
<username>@- <RN> - hy345sh:<directory> $
```

## USERNAME

Currently logged-in user

## RN

Your registration number, e.g.

`csd9999`

## DIRECTORY

Current working directory

```
myuser@- csd4806 - hy345sh: /spare/csd4806 $
```

# Simple commands and sequences

Each external command runs in a **new child process**. Same behaviour for separate lines or one line with `;`.

## SEPARATE LINES

```
ls  
cd /tmp  
echo "Test"
```

## SAME LINE

```
ls; cd /tmp; echo "Test"
```

Spaces around `;` are optional: `ls;pwd;date` must work too.

# Global variables

Assign and expand shell variables.

```
x=world  
echo hello $x
```

Declares `x` with value `world`, then prints `hello world`.

Also works in sequences: `age=20; echo I am $age years old`

# Pipelines

| connects stdout of one process to stdin of the next. Support two or more stages.

```
cat file.txt | grep "world"
```

Spaces around | are optional.

# Redirection

Redirect the three standard streams. Operators may be combined on one command.

## OPERATORS

- < stdin from file
- > stdout overwrite
- >> stdout append
- 2> stderr overwrite

## EXAMPLES

```
./prog < test.in  
./prog > /dev/null  
./prog >> log.txt  
ls missing 2> errors.txt  
./prog < in > out 2> err
```

# Background execution

A command or pipeline ending with `&` runs in the background, the shell returns to the prompt immediately.

```
sleep 10 &  
cat file.txt | sort | uniq &
```

## FOREGROUND

No `&` → shell waits until the job finishes.

## JOB CONTROL

Track background jobs. Support built-ins `jobs` and `wait`.

# Terminal control and signals

## CTRL-C

Interrupts the **foreground** command or pipeline. The shell itself must **not** exit.

## CTRL-S

Stops terminal output. Does not kill the shell or the process.

## CTRL-Q

Resumes terminal output after Ctrl-S.

# Control flow: `if` and `for`

Single-line and multiline forms are both required.

## IF

```
x=5; if [ $x == 5 ]; then  
    echo "hello world"; fi
```

`[` is `test(1)`. Spaces inside the brackets matter.

## FOR

```
for i in 1 2 3; do  
    echo $i; done
```

Also multiline `for` / `do` / `done`.

## PART 4

# How to build and test

Order of work, and how to verify against Bash

# Grow the shell in milestones

1. **Prompt + read a line** — print username, RN, cwd, accept input.
2. **One external command** — `fork`, `exec`, `wait`.
3. **Sequences + variables** — `;`, assignments, `$` in the parent.
4. **Built-ins** — built-in command support such as `cd`, `ls` in the shell process.
5. **Redirection** — parse operators (`<`, `>`, `2>`)
6. **Pipelines** — pipe commands support `|`
7. **Background + signals** — `&`, job table, Ctrl-C safe for the shell.
8. `if` / `for` — single-line and multiline forms.

# Compare with Bash early and often

Pick a few lines that exercise composition then invent more yourself.

## COMPOSITION

```
ls -al>y1; cat y1; rm y1  
ls|sort|uniq|wc  
echo hello > yy; echo world >> yy  
cat < yy | sort | uniq | wc > y1
```

## STATE + CONTROL

```
age=20; echo I am $age years old  
x="5|5;10"; echo $x>f; cat<f  
sleep 10 &
```

Match Bash on stdout, files created, and whether the prompt comes back.

## PART 5

# Execution environment

Isolate crashes. Keep your quota. Transfer files safely.

# Why QEMU (and `/spare`)

A buggy shell can spawn too many processes. On a shared machine that would be an issue. Inside QEMU, the worst case is restarting the VM.

## DISK IMAGE

Copy `~hy345/qemu-linux/hy345-development.img` under `/spare/<your-RN>`, not into home (quota).

`/spare` is machine-local: always use the **same** host.

## BOOT

```
qemu-system-i386 \  
-hda ${MYSPARE}/hy345-development.img \  
-display curses
```

Login: `hy345` / `hy345`. Exit cleanly with `turnoff`.

# Files between host and guest

Guest sees the host at `10.0.2.2`. Keep sources on the host, copy in what you need.

## HOST → GUEST

```
scp user@10.0.2.2:~/file.txt .
```

Run from inside the guest. Note the trailing `.`

## GUEST → HOST

```
scp file.txt user@10.0.2.2:~/
```

## PART 6

# Deliverables

What to submit on E-Learn

# What to submit

ZIP on the CS345 E-Learn page:

## INCLUDE

Source files

`Makefile` with `make all` and `make clean`

`README` with design notes

## RULES

**No** disk image in the ZIP

Name + student ID in source comments

Individual work · no AI-generated code

# **Thank you for your attention**

## **Questions?**