Lecture 5: Terminal and UNIX Shell

Terminal, Sessions and Process Groups, and Shell Programming

Xin Liu

Florida State University
xliu15@fsu.edu

COP 4610 Operating Systems
https://xinliulab.github.io/FSU-COP4610-Operating-Systems/

Recap: Objects in an Operating System

We already know how to use system calls related to *file descriptors* to access objects in the operating system:

`open`, `read`, `write`, `lseek`, and `close`.

The OS also provides system calls that can **create** objects within the OS, such as:

`mount`, `pipe`, and `mkfifo`

Today's topic

We know that the objects in an operating system are far more than these. There are still many interesting ones we have not explored in depth— the terminal, for example, is both fascinating and unsettling.
Starting from the terminal emulator you use every day, we will explore what exactly happens when you press `Ctrl-C`.
On this basis, we will then be able to implement our own *multi-task manager*.

Terminal

Typewriter

QWERTY keyboard (1860s)

- The layout aimed to reduce jams by spreading common letter pairs.
- The mechanism was fully mechanical, so each keystroke needed firm force.

Legacy of the Typewriter Era

Shift

- Move the type mechanism or platen by one step to switch the character set.

CR & LF

- `\r` **CR (Carriage Return)**: return the print head to the start of the line. Example: `print("Hel\rlo")`
- `\n` **LF (Line Feed)**: advance the paper by one line.
- On Unix, `\n` represents a newline; many terminals render it as CR+LF.

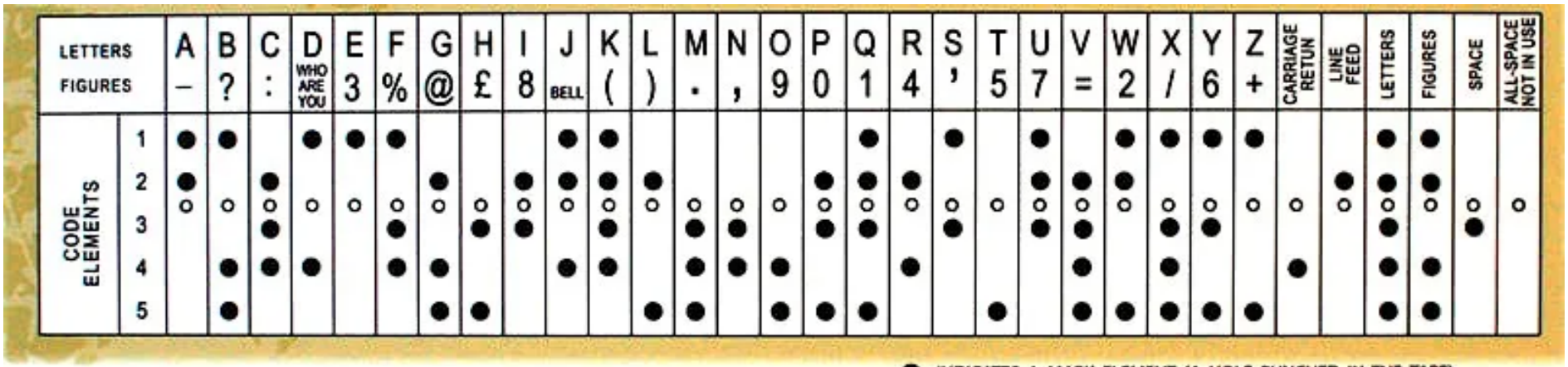
Tab & Backspace

- Cursor movement. A common typo fix on typewriters was `Backspace` then – to strike through a character.

Teletypewriter (TTY)

Built for telegraph service (print at both ends)

- Telex (teleprinter exchange), 1920s, before digital computers
- Uses the **Baudot code** (5-bit)
- Mapped naturally onto early computer terminals



LETTERS / FIGURES	A –	B ?	C :	D WHO ARE YOU	E 3	F %	G @	H £	I 8	J BELL	K (	L)	M .	N ,	O 9	P 0	Q 1	R 4	S '	T 5	U 7	V =	W 2	X /	Y 6	Z +	CARRIAGE RETUN	LINE FEED	LETTERS	FIGURES	SPACE	ALL-SPACE NOT IN USE
CODE ELEMENTS 1	●	●		●	●	●				●	●						●		●		●		●	●	●	●			●	●		
CODE ELEMENTS 2	●	○	●	○	○	○	●	○	●	●	●	●	○	○	○	●	●	●	○	○	●	●	●	○	○	○	○	●	●	●	○	○
CODE ELEMENTS 3	○	○	●	○	○	●	○	●	●	○	●	○	●	●	○	●	●	○	○	○	●	●	○	●	○	●	○	○	●	○	●	○
CODE ELEMENTS 4		●	●	●		●	●			●	●		●	●	●			●				●		●			●		●	●		
CODE ELEMENTS 5		●					●	●				●	●		●	●	●			●		●	●	●	●	●			●	●		

The International Telegraph Alphabet

● INDICATES A MARK ELEMENT (A HOLE PUNCHED IN THE TAPE)
○ INDICATES POSITION OF A SPROCKET HOLE IN THE TAPE

Legacy: `tty`, `stty`, `/dev/tty`, `getty`

Teletypewriter (cont'd)

Teletype Model 28 (1951)

VT100: The Classic Terminal

Video Terminal (DEC, 1978)

- Became the de facto industry standard
- First terminal with full ANSI escape sequence support
- 80×24 character screen became the standard layout
- Many later devices advertised “VT100 compatible”

ANSI Escape Sequences

Format:

- General form: `ESC [ parameters m`
- `ESC` = `\033` (ASCII 27)
- Parameters control text style, color, etc.

Example:

- `\033[01;31mHello, OS World\033[0m`
  - `01` = bold (bright)
  - `31` = red foreground
  - `0` = reset to normal
- Effect: **Hello, OS World** will appear in bold red, then style resets.

Try it: miniHello.s

More Fun Demos with Terminal

- `telnet towel.blinkenlights.nl` (ASCII movie; exit with `Ctrl-]` then `q`)
- `dialog --msgbox 'Hello, OS World!' 8 32` (pop up a message box in the terminal)
- `ssh sshtron.zachlatta.com` (play an online game in your terminal)

Elegant Weapon for a More Civilized Age

- Long ago, on green-screen terminals (like the VT100), pioneers built the powerful and elegant **UNIX** operating system.
- Today, the latest **iPhone** in your pocket has vastly more computing power than all of NASA's **Apollo moon landing** computers combined.
- And we use this epic level of power... to play Angry Birds.
- Don't laugh, I do this too.

Computer Terminals: How They Work

As an output device

- Receives bytes over UART and renders characters on the screen
- ANSI escape sequences control cursor and color because the stream is just text

As an input device

- Sends ASCII codes for key presses over UART
- Many codes are control characters (CR, LF, TAB, BS)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SPC	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Today, we emulate terminal: Pseudo Terminal (PTY)

- A bidirectional channel like a pipe.
- **Master** side: connected to the terminal emulator.
- **Slave** side: connected to a shell or another program.
- Example slave device: `/dev/pts/0`.

How PTYs are created

- Created by `ssh`, terminal apps, or `tmux`.
- `openpty()`: request a new terminal via `/dev/ptmx`
- Returns two file descriptors (master and slave); the slave appears under `/dev/pts/N`.

Have Fun with PTYs:

- Open two terminals
- Run `tty` to see each PTY (e.g., `/dev/pts/2`, `/dev/pts/3`)
- Try: `echo hello > /dev/pts/3`

Terminal Emulator

You can implement this

- Use `openpty` + `fork`.
- Child: redirect `stdin/stdout/stderr` to the PTY slave.
- Parent: read from the PTY master to draw the screen and write keyboard input back to the master.

Extended escape sequences for images

- Some emulators, e.g., Kitty, add image protocols via ESC sequences.
- Start with `\033[...` and end with `\033`.
- You can control size, position, and animation.

Terminal: More Capabilities

Terminal modes

- **Canonical mode**: line-oriented. Data is sent to the program after Enter. The terminal provides line editing.
- **Non-canonical (raw) mode**: character-oriented. Each byte is delivered immediately. Used by tools like `vim` and `ssh`.

Attribute control

- APIs: `tcgetattr` / `tcsetattr`.
- You can control echo, signal generation, and special characters.
- Example: turn off echo when typing a password.

Sessions and Process Groups

Who Is the Parent Process?

- When you launch a program from a terminal, who is its parent process?
- To check, run `ps -p <pid> -o pid,ppid,cmd` and read the `PPID`.

Luke, I am your father!

How a Program Is Paired with a Terminal

Where a session starts

- Local login: kernel → init → getty
- Remote login: sshd → fork → openpty
  - `stdin`, `stdout`, `stderr` are attached to the allocated TTY
- VS Code terminal: fork → openpty

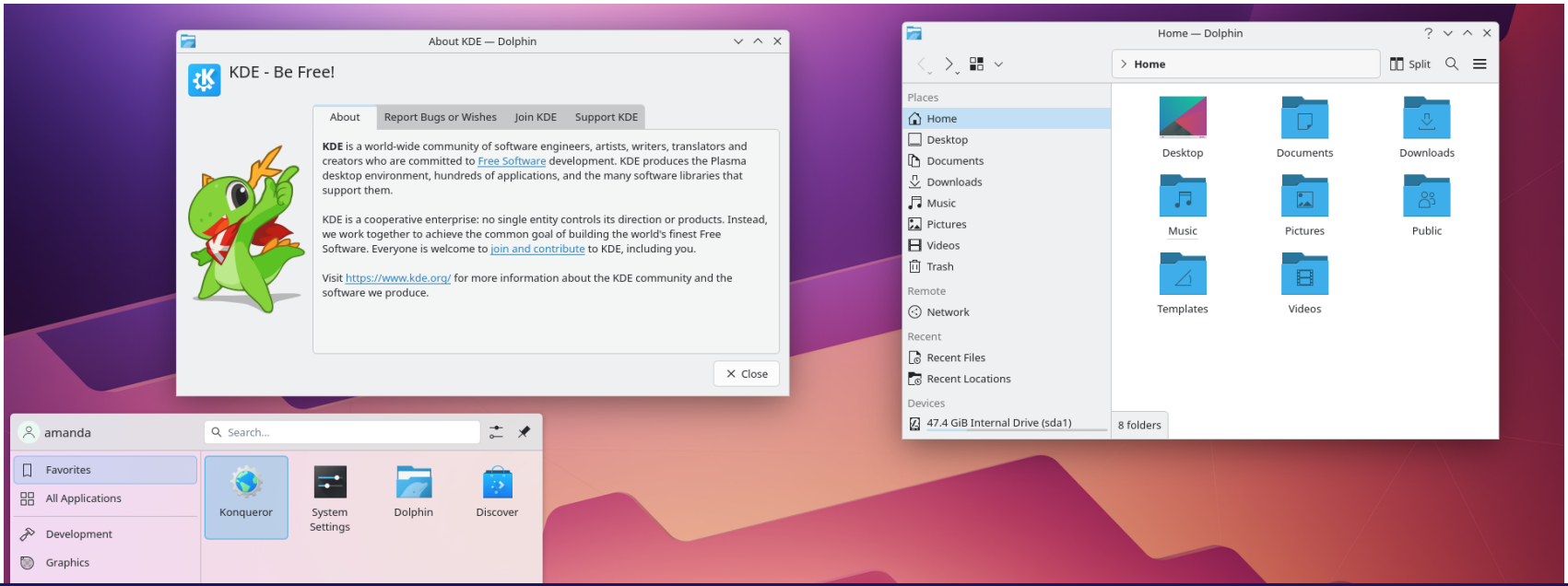
Why the program sees that terminal

- `login` is just a user program
- `fork()` inherits file descriptors
  - The child process keeps the same terminal device

Finally, we can build a *User Interface*

From TTY to GUI

- Kernel provides input, display, and audio drivers.
- Graphics path: DRM/KMS → display server (Wayland or Xorg) → compositor/window manager → desktop environment (KDE, GNOME).
- Apps draw with Qt or GTK using OpenGL or Vulkan.
- Input events go to the compositor then to the focused app.
- Apps talk over IPC (Wayland or X11), each app is a process with its own resources.



Not Graphical, But Command-Line

UNIX Shell: a classic of the terminal era

- The peak of the command-line interface (CLI).

1970

2020

Process Management

Many processes share one terminal. Some run in the foreground, some in the background. Which one is stopped by `Ctrl-C`*?*

The terminal does not decide

- It only sends bytes.
- `Ctrl-C` = End of Text (ETX), `\x03`.
- `Ctrl-D` = End of Transmission (EOT), `\x04`.
- See current bindings with `stty -a`.

The OS decides

- The TTY driver interprets the control byte.
- It delivers a signal to the **foreground process group** (e.g., **SIGINT** for `Ctrl-C`).

The "current process" on a terminal

As OS designers, when the TTY receives `Ctrl-C` we must identify the *current process set*.

How would you design it?

- `fork()` creates a tree of processes (with possible reparenting).
- A foreground job may contain multiple processes, for example a pipeline.
- `Ctrl-C` should signal all processes in the foreground job.
- Background jobs must not be affected.

Hint: use process groups and sessions to represent these sets on a TTY.

Sessions and Process Groups

At this point, the shell (*bash*), *find*, *wc*, *sort*, and *uniq* are all running.

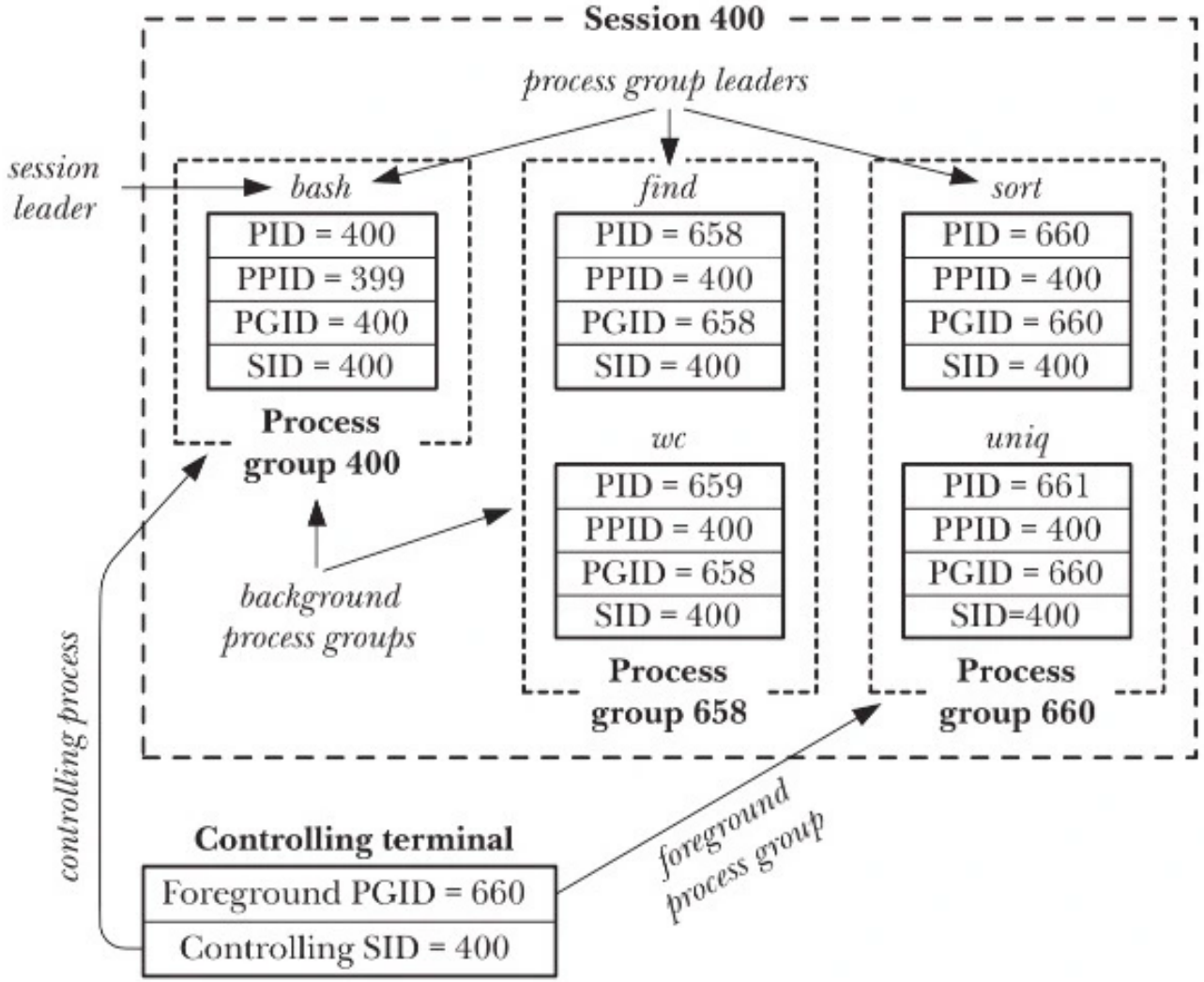


Figure 34-1: Relationships between process groups, sessions, and the controlling terminal

Sessions and Process Groups (Cont.)

Key ideas

- **Session (SID)**: a set of process groups. Created by a session leader via `setsid()`. A session may have one controlling terminal.
- **Process group (PGID)**: a set of related processes treated as one job. The group leader's PID equals the PGID.
- **Foreground group**: the PGID currently attached to the TTY. It receives job-control signals.
- **Background groups**: run without TTY input.

How shells use them

- The shell is the session leader (SID = shell PID).
- Each pipeline is put into its own PGID; the first child becomes the group leader.
- The TTY stores foreground PGID and the session's SID.

Signal routing

- `Ctrl-C` ⇒ kernel sends `SIGINT` to the foreground PGID.
- `Ctrl-Z` ⇒ `SIGTSTP` to the foreground PGID.

Sessions and Process Groups: Mechanics

Session ID (SID)

- A child inherits its parent's SID.
- One session is associated with one controlling terminal.
- When the session leader exits, the kernel sends `SIGHUP` to the session.

Process Group ID (PGID)

- A session contains multiple process groups. A TTY has exactly one foreground group.
- The terminal records the foreground PGID.
- On `Ctrl-C`, the OS delivers `SIGINT` to all processes in the foreground group.

Sessions & Process Groups: APIs

Session (SID)

- `setsid()` — create a new session and detach from the controlling terminal.
- `getsid(pid)` — query a process's session ID.

Process Group (PGID)

- `setpgid(pid, pgid)` — create/join a process group; `getpgid(pid)` — query.
- `tcsetpgrp(fd, pgid)` / `tcgetpgrp(fd)` — set/get the foreground PGID of a TTY (*quirky but standard*).

Related identity knobs

- `uid`, `euid`, `suid` — real, effective, and saved user IDs.
- For background: *Setuid Demystified*.

Yes, historical baggage

But it is part of POSIX

- The future of software is hard to predict

Rethink the problem

- We do not need to bind a process to a device
- Let managers emulate it: `tmux`, GNOME, others
  - Window manager needs only the idea of a process group
  - Close a window $\rightarrow$ kill the whole group
- Android: each app runs as a different user
  - Force stop $\rightarrow$ kill all processes of that user
- Snap: apps run in a sandbox
  - AppArmor + seccomp + namespaces

What does `Ctrl-C` actually do?

`signal()`

- Register a handler function `f` for a signal.
- The kernel records `f`. When `SIGINT` is delivered, it arranges to run `f` when the process returns to user mode.
- Default action of `SIGINT` is to terminate if no handler is set.

`kill()`

- Send a signal to a PID or a process group.
- On terminals, the TTY sends `SIGINT` to the foreground process group.

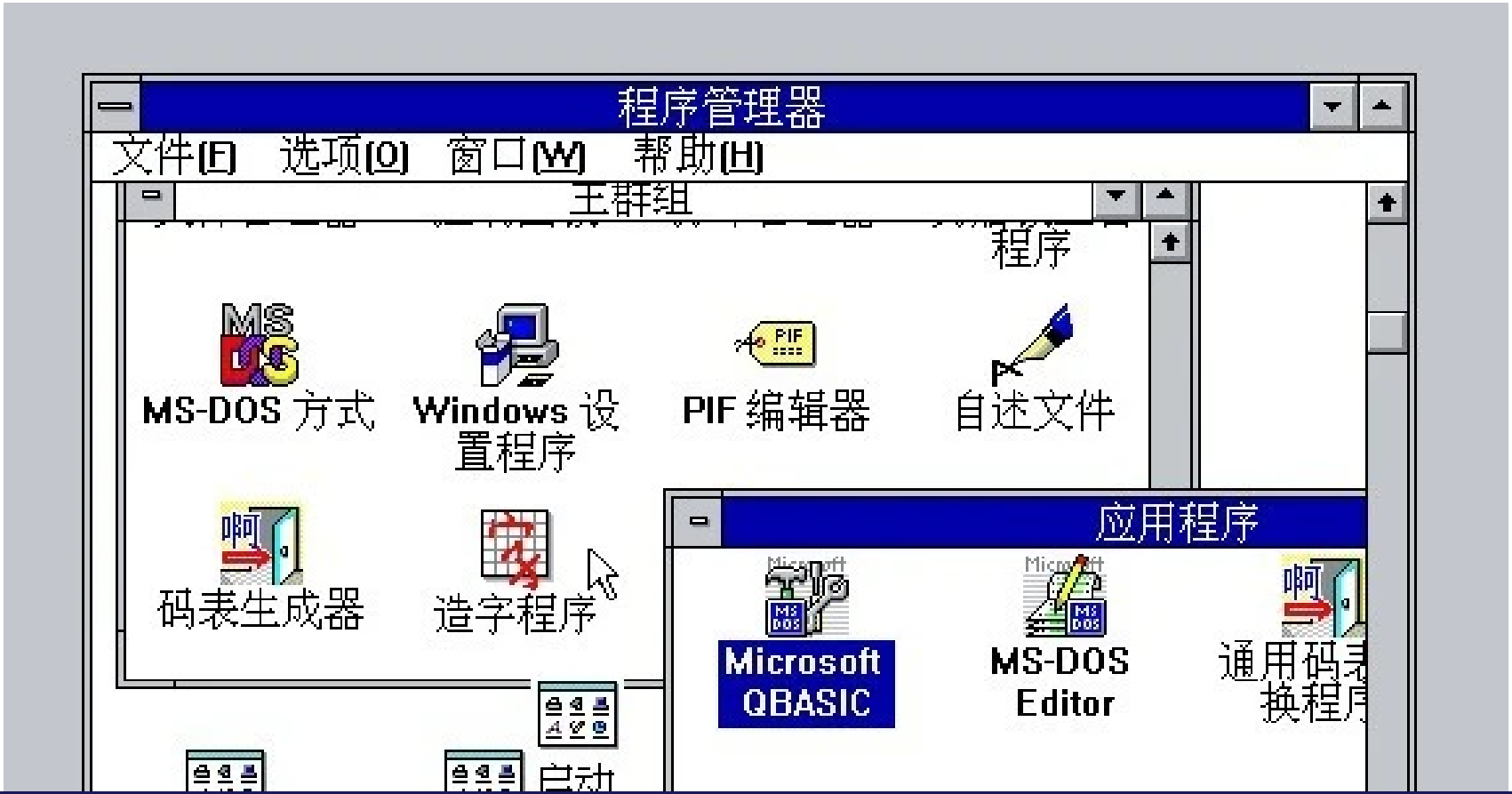
Prefer `sigaction()`

- More reliable control than `signal()`.
- Lets you set a mask and flags such as `SA_RESTART`.

(UNIX) Shell: A Programming Language

Multitasking is not all of HCI

- **Windows 3.2 (1992):** Program Manager showed many windows, not great workflows.
- More windows do not mean better interaction.
- Users still had to break goals into steps and keep switching context.
- Good UI is task- and goal-oriented, not window-oriented.



The Shell Programming Language

UNIX users are hackers
UNIX Shell is a tiny text-substitution language

- One data type: string.
- Do not support arithmetic.

Mechanics

- Command substitution: `$( ... )` Process substitution: `<( ... )`.
- Redirection: `cmd > file` `cmd < file` `cmd 2> /dev/null`.
- Sequencing: `cmd1 ; cmd2` `cmd1 && cmd2` `cmd1 || cmd2`.
- Pipe: `cmd1 | cmd2`.
- These expand to syscalls: `open`, `dup2`, `pipe`, `fork`, `execve`, `waitpid`.

Example: Implementing Redirection

Using the property of child process inheriting file descriptors

- The parent process opens files, then passes them to the child process
  - It turns out Windows API is more "elegant"

```
int fd_in = open(..., O_RDONLY | O_CLOEXEC);
int fd_out = open(..., O_WRONLY | O_CLOEXEC);

int pid = fork();
if (pid == 0) {
    dup2(fd_in, 0);
    dup2(fd_out, 1);
    execl(...);
} else {
    close(fd_in);
    close(fd_out);
    waitpid(pid, &status, 0);
}
```

Read The FXXK Manual (RTFM)

man sh: dash — command interpreter (shell)

- **dash** is the standard command interpreter for the system. The current version of dash is in the process of being changed to conform with the POSIX 1003.2 and 1003.2a specifications for the shell.
- The shell is a command that reads lines from either a file or the terminal, interprets them, and generally executes other commands. It is the program that is running when a user logs into the system (although a user can select a different shell with the `chsh(1)` command).

UNIX Shell: Advantages

Advantages: Efficient, Concise, Precise

- A kind of ”natural programming language”: one command line, coordinating multiple programs
  - `make -nB | grep ...`
  - Best suited for quick & dirty hackers

UNIX Shell: Advantages Come with Drawbacks

Inherent Limitations

- The shell's design was constrained by the computing power, algorithms, and engineering capability of the 1970s.
  - Later generations had to live with the flaws (PowerShell: "I am good, but nobody uses me")

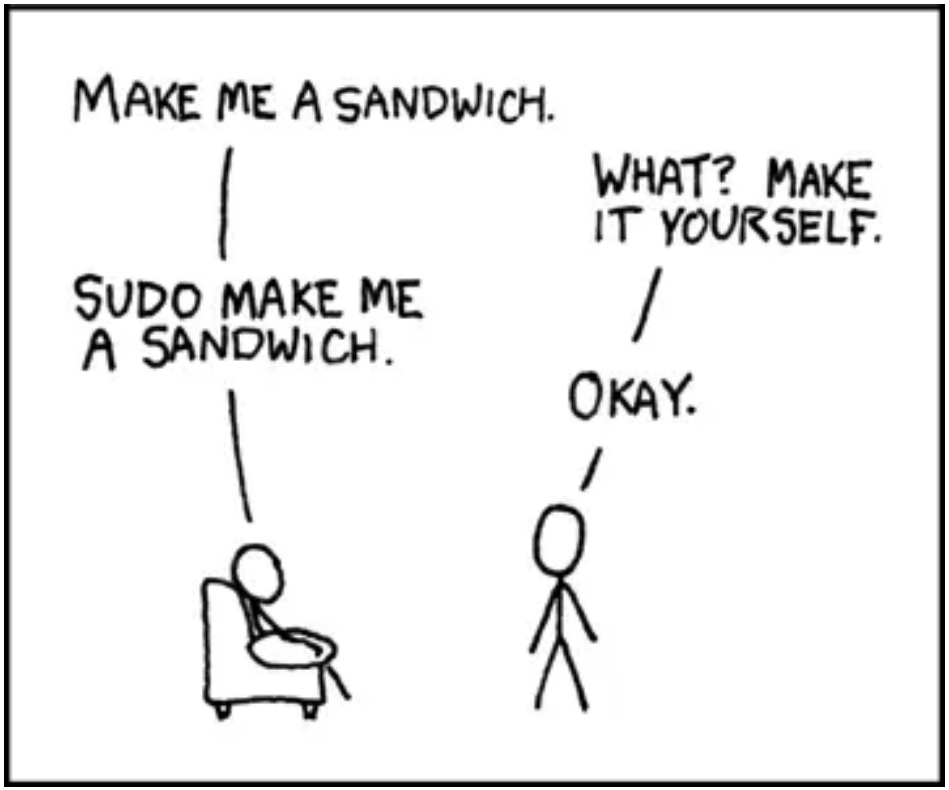
Example: Operation "Precedence"?

- `ls > a.txt | cat`
  - I already redirected output to `a.txt`, so does `cat` receive no input?
- Behavior differs between bash and zsh
  - That's why scripts often use `#!/bin/bash`, or even `#!/bin/sh`, to maintain compatibility

Text Data: "Use at Your Own Risk"

- Whitespace = disaster

Another Interesting Example




```
$ echo hello > /etc/a.txt
bash: /etc/a.txt: Permission denied

$ sudo echo hello > /etc/a.txt
bash: /etc/a.txt: Permission denied
```

Takeaways

Through the shell, we illustrate the true meaning of *"building an entire world of operating system applications on top of system calls"*:

The OS API and applications evolve together in a spiral manner. New application needs lead to new operating system features.

UNIX provides us with a very concise and stable interface (`fork`, `execve`, `exit`, `pipe`, ...).
Despite its heavy historical baggage, it still works remarkably well today.

- Looking Ahead: What is the future of Shell (CLI/GUI)?