CS 214 / 2021-03-08
==========

Regarding dirent.h and readdir

The only field that Posix requires in struct dirent is d_name
	d_name is guaranteed to point to a terminated string
	NAME_MAX from limits.h is the maximum length of this string
		It is not recommended to create an array of length NAME_MAX
		NAME_MAX may be extremely long, depending on your OS/architecture

If you are using -std=c90 or -std=c99, you won't have access to other
fields, like d_type
	-> but you can use stat to get this information

If you are using GNU extensions (the default for GCC), then d_type is available
-> it does not matter whether you use d_type or stat

	d_type is technically non-standard, but it is available on BSD and
	Linux systems, which is most of the Unix-like systems you are likely
	to encounter

	d_type is faster than stat, because you don't need to call the OS again
	but the speed difference is not significant for us

How are you verifying your program?
- just eyeballing the results
- made a bunch of sample output and compare program output

- write a program to test the output of your program
	- what properties should your program output have?
	- Only three whitespace sequences: " ", "\n", "\n\n"
		- in particular, no spaces adjacent to newlines
	- The number of characters between two newlines should not exceed the page width
		- exception if input contains words that are too long
	- If you run ww again on the output of ww with the same width, you should
		get the same text

		./ww 80 some_input > output1
		./ww 80 output1 > output2

		output1 and output2 should be identical

Useful unix command: tee

	$ tee <file_name>

	tee reads from standard input and writes to standard output and a file
		lets me split the input and send two copies to different places
		handy for saving data from the middle of a pipeline

	$ program1 | tee p1_out | program2 > p2_out

		Output from program1 is sent to tee
		tee passes the output to program2 and saves a copy to p1_out
		Output of program2 is written to p2_out

Reminder about paths

-> the file names we give to open, opendir, fopen, etc. are really paths

	A path is a sequence of names separated by slashes (/)
		Paths starting with / are absolute
		Other paths are relative
			A relative path says how to reach a file, starting from the
			current directory (also called the working directory)

	Eg., foo/bar/baz

		means "baz" inside "bar" inside "foo" in the current directory

	Our processes have a current working directory:
		the directory the user was in when they started the program
		that is, we inherit the working directory from the shell

		-> thus, file names are interpreted relative to the directory we started in

How can we open files in another directory?

	1. make a path: directory/filename
		when reading from a directory, just append a slash and the file name to
		the name we used to open the directory
		-> a little obnoxious, since strcat requires us to allocate space in advance
		-> but eminently doable

		There is a limit on path length, but this is not generally a problem
		-> open will just return -1 if the name is too long
		-> the max path length may be very long, so use dynamic (heap) allocation

	2. use chdir to change the working directory
		now all my filenames are relative to the new directory
		no need for concatenating path names

		int chdir(char *path);

			returns 0 for success
			on error, returns -1 and sets errno

--

/dev directory contains "files" that correspond to hardware devices

	/dev/null - a device that just throws away its input
		- commonly used as a way to throw out/ignore the output from a program

		./program_with_lots_of_output > /dev/null

On Linux, /proc contains an entry for every running process

Processes

What is a "process"?
- for us, a process is a program that is currently executing

	process = program + state

		state includes contents of memory, registers, open files, OS data

In principle, we can have multiple processes running concurrently that all use
the same program

The OS keeps track of each running process
	On Unix, we have the Process Control Block
		-> data structure used by the OS
			- process ID (pid)
			- what virtual memory address space we are using
			- what files we have open
			- environment information (e.g., current working directory)
			- permissions, process user/group, etc.

Where do new processes come from?
In general, the OS has some mechanism
In Unix-like systems, new processes are created by fork()

	pid_t fork();

		pid_t is an integer type (width is OS-dependent)

	fork clones or duplicates the current process
		-> the original process is called the "parent"
		-> the new process is called the "child"

		The OS has a special program that starts first (possibly "init")
			every other process is the child of some other process

	When we call fork, the current process is duplicated
		- both duplicates have the same program
		- initially have the same memory contents / program state
			- but the child gets a copy of the memory
		- after fork returns, the parent and child may diverge

	What fork returns will be different between the parent and child
	-> parent receives the PID of the child
	-> child receives 0
			(0 is a valid PID, but it always identifies the root process, so it
			can never by the PID of a child process)

	Typical usage

	pid_t pid = fork();
	if (pid < 0) {
		perror("fork");
		abort();
	} else if (pid == 0) {
		puts("In child!");
	} else {
		printf("In parent! Child is %d\n", pid);
	}

	Remember: both child and parent start with the same program in the same state
		(aside from the return value from fork)
	we can think of fork as returning twice
		-> once to parent
		-> once to child
	both parent and child resume executing after the return from fork

	Fork can fail if the OS (or the user) has too many processes
		fork returns -1 and sets errno on failure

What happens when a process ends?
	-> who collects our return code?

	- Much of the data for our process is forgotten
	- The process control block holds our exit status, in case our parent wants it

When a process ends, it becomes a "zombie"
	A zombie process has died, but it's PCB still lives in the process list

The parent must call wait() to clean up the zombie process
	wait is also how we get the exit status of the child

If a parent process ends before its child, the child becomes an "orphan"
	An orphan process has no parent process to wait for it when it ends
	-> OS will make sure the orphan is "adopted" by some process (usually init)
		- the adopter will wait for the child to terminate

If the parent and child both terminate, the child becomes a "zombie orphan"
	-> The OS will have some process adopt the child and wait

Only specialized processes can adopt orphans
		- process do not necessarily adopt distant descendents
		- there is no "grandparent" relationship

wait
----

	pid_t wait(int *wstatus);

	wait suspends a program (blocks) until one of its child processes terminates
		It returns the PID of the terminated process
		If wstatus is not NULL, it writes information about how the process exited
			into wstatus
				-> exit status
				-> whether the process was terminated by a signal

	wait (and related functions) are how we prevent orphans

	any time we call fork, the parent should call wait

When should we call wait?
	It depends on what we're doing

Why does any of this matter?
Next time: exec and multiprocessing