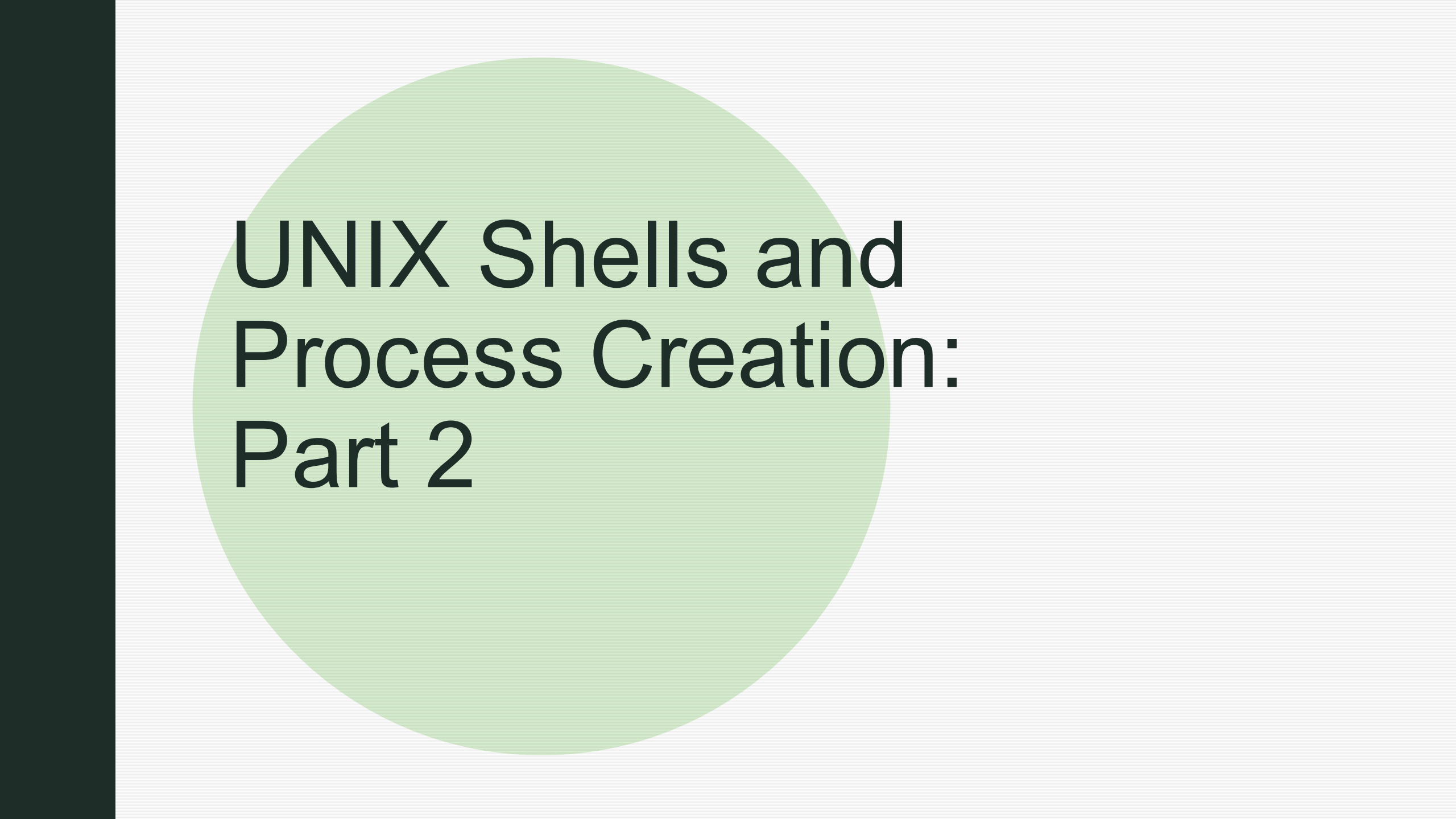


CSC 360: Operating Systems

Tutorial #3



UNIX Shells and Process Creation: Part 2

Review of Shells

- ❑ Shells provide an interface to the user through a command-interpreter
- ❑ Commands are either *built-in* or *external*
- ❑ Built-in commands are part of the shell program itself
- ❑ External commands are found by looking in directories specified by the \$PATH variable
- ❑ We use the fork() system call to request the operating system to clone the existing process and create a new child process. The shell is the parent process and waits for it's child to finish executing before it resumes.

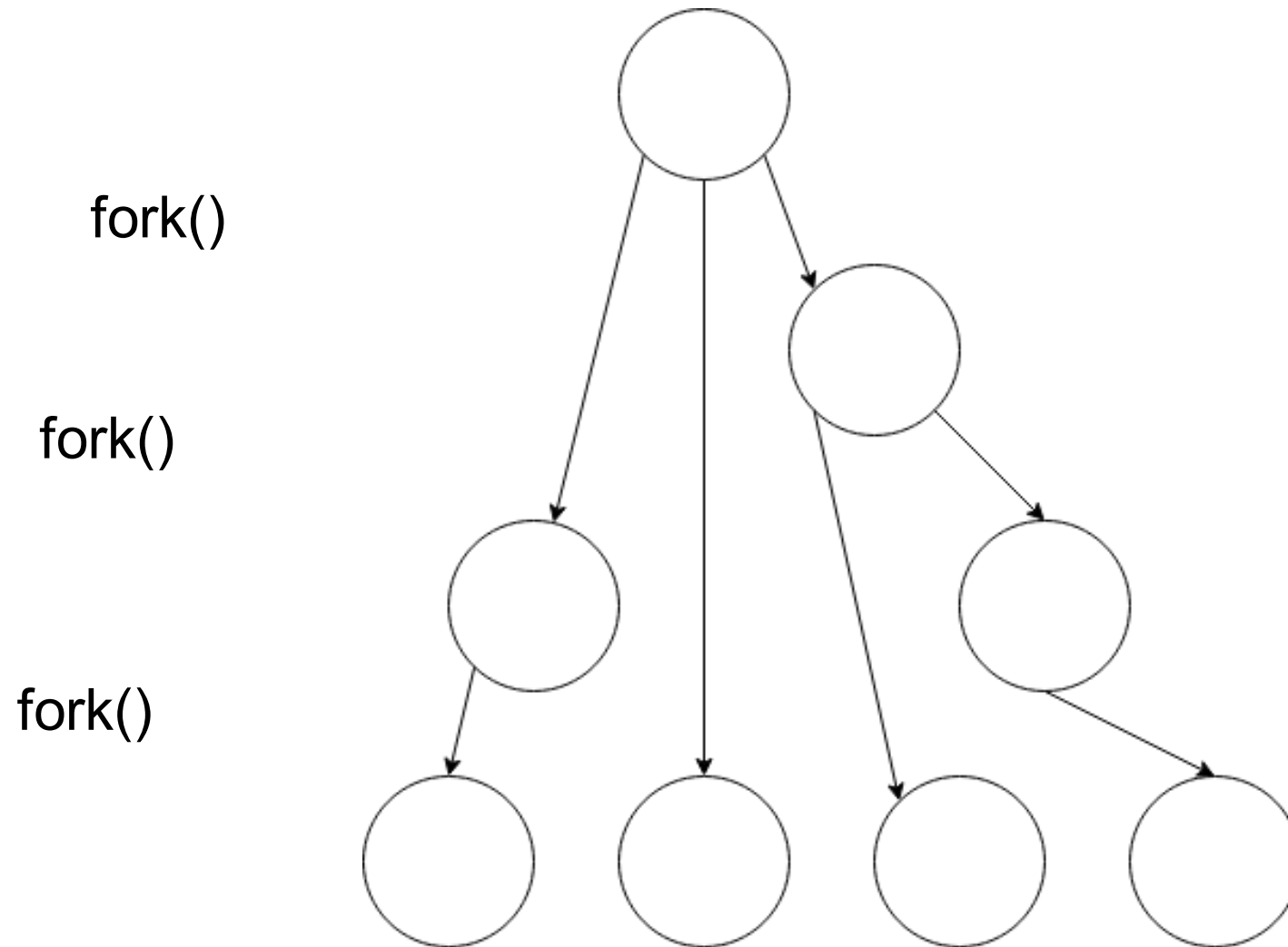
Review of Processes

- ❑ A process is a single instance of a program that is being run on the system
- ❑ The operating system is responsible for managing processes and their access to system resources
- ❑ Processes are continually swapped from different resource queues as they move through different states
- ❑ Processes each have their own address space and cannot access the address space of other processes. Pointers in one process refer to virtual memory addresses in this address space and not actual physical memory.
- ❑ At startup the first process to run is the *init* process. After this all other processes are forked from existing processes.

Review of fork()

- ❑ fork() is a system call which requests the operating system to clone the calling process and starts running a new instance of the same program
- ❑ The new child process is put on the ready queue and begins running concurrently with the parent process
- ❑ Execution resumes in both programs from the instruction immediately following the fork()
- ❑ We don't know which order they will be scheduled to run in
- ❑ fork() returns the child's PID to the parent and returns 0 to the child

Last Week's Quiz: Answer



Review of exec()

- ❑ exec() is the system call used when we want to run a new program
- ❑ It overwrites the existing address space of the process that calls exec() with the code of the new program specified in the arguments
- ❑ The heap and stack pointers are all reset
- ❑ There are a variety of exec functions with different argument options

Environment Variables

- ❑ Environment variables store information that is useful to a running program
- ❑ Each process has its own set of environment variables
- ❑ They can include things such as the user's home directory, the working directory and...of course... the \$PATH.
- ❑ We can access and change them through system calls such as `setenv()`, `unsetenv()` and `getenv()`

- ❑ We can list all the environment variables of the terminal by typing:
env
- ❑ We can set them by typing:
env VAR=something

```
Last login: Mon Jan 27 17:36:49 on ttys000
[iaiemslie@Iains-MacBook-Pro ~ % env
TMPDIR=/var/folders/bh/ddmyy0s55v9597ys4wxbh2y80000gn/T/
XPC_FLAGS=0x0
TERM_PROGRAM_VERSION=433
LANG=en_CA.UTF-8
TERM_PROGRAM=Apple_Terminal
XPC_SERVICE_NAME=0
TERM_SESSION_ID=7420B8F1-B766-4E7E-9EB0-380B9CAF3836
TERM=xterm-256color
SSH_AUTH_SOCK=/private/tmp/com.apple.launchd.r0D9EB1F74/Listeners
SHELL=/bin/zsh
HOME=/Users/iaiemslie
LOGNAME=iaiemslie
USER=iaiemslie
PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Library/TeX/texbin
SHLVL=1
PWD=/Users/iaiemslie
OLDPWD=/Users/iaiemslie
_=/usr/bin/env
[iaiemslie@Iains-MacBook-Pro ~ %
[iaiemslie@Iains-MacBook-Pro ~ %
[iaiemslie@Iains-MacBook-Pro ~ %
[iaiemslie@Iains-MacBook-Pro ~ %
[iaiemslie@Iains-MacBook-Pro ~ % env USER=iaie
TMPDIR=/var/folders/bh/ddmyy0s55v9597ys4wxbh2y80000gn/T/
XPC_FLAGS=0x0
TERM_PROGRAM_VERSION=433
LANG=en_CA.UTF-8
TERM_PROGRAM=Apple_Terminal
XPC_SERVICE_NAME=0
TERM_SESSION_ID=7420B8F1-B766-4E7E-9EB0-380B9CAF3836
TERM=xterm-256color
SSH_AUTH_SOCK=/private/tmp/com.apple.launchd.r0D9EB1F74/Listeners
SHELL=/bin/zsh
HOME=/Users/iaiemslie
LOGNAME=iaiemslie
USER=iaie
PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Library/TeX/texbin
SHLVL=1
PWD=/Users/iaiemslie
OLDPWD=/Users/iaiemslie
_=/usr/bin/env
iaiemslie@Iains-MacBook-Pro ~ %
```

Wait...so what does wait() do?

- ❑ wait() suspends the execution of the parent process until one of its children terminates
- ❑ waitpid() suspends the execution of the parent process until the process with the PID supplied as an argument terminates
- ❑ When wait() is called, the calling process is blocked and removed from the ready queue by the operating system
- ❑ We can use this in our shell to suspend the parent process until the new child process finishes executing

Signals?

- ❑ Signals are a form of inter-process communication
- ❑ They serve a similar purpose as *interrupts*
- ❑ When a signal is received it must be dealt with by a *signal-handler*
- ❑ We can specify something called a default signal handler or create our own
- ❑ If we're running a terminal program and press <CTRL-Z> this will cause the running program to terminate
- ❑ In our shell program we can use the `signal()` function

```
typedef void (*signalhandler_t) (int);    //This is a function pointer

// First argument indicates some signal
// Second argument indicates what to do when the process receives that signal
// We can create our own handler or use a default one by giving argument SIG_DFL
// We can ignore a signal by giving the argument SIG_IGN

signalhandler_t signal(int signum, signalhandler_t handler)
```

Example of some Signals

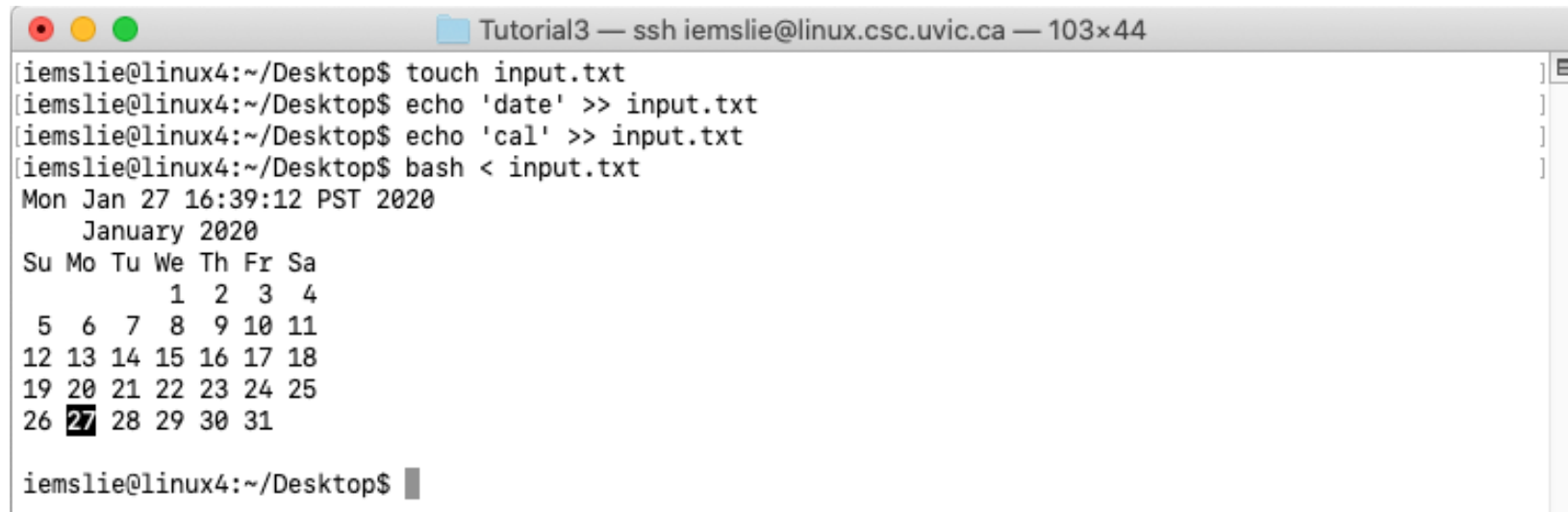
Signal	Standard	Action	Comment
SIGABRT	P1990	Core	Abort signal from abort(3)
SIGALRM	P1990	Term	Timer signal from alarm(2)
SIGBUS	P2001	Core	Bus error (bad memory access)
SIGCHLD	P1990	Ign	Child stopped or terminated
SIGCLD	-	Ign	A synonym for SIGCHLD
SIGCONT	P1990	Cont	Continue if stopped
SIGEMT	-	Term	Emulator trap
SIGFPE	P1990	Core	Floating-point exception
SIGHUP	P1990	Term	Hangup detected on controlling terminal or death of controlling process
SIGILL	P1990	Core	Illegal Instruction
SIGINFO	-		A synonym for SIGPWR
SIGINT	P1990	Term	Interrupt from keyboard
SIGIO	-	Term	I/O now possible (4.2BSD)
SIGIOT	-	Core	IOT trap. A synonym for SIGABRT
SIGKILL	P1990	Term	Kill signal
SIGLOST	-	Term	File lock lost (unused)
SIGPIPE	P1990	Term	Broken pipe: write to pipe with no readers; see pipe(7)
SIGPOLL	P2001	Term	Pollable event (Sys V). Synonym for SIGIO
SIGPROF	P2001	Term	Profiling timer expired
SIGPWR	-	Term	Power failure (System V)
SIGQUIT	P1990	Core	Quit from keyboard
SIGSEGV	P1990	Core	Invalid memory reference
SIGSTKFLT	-	Term	Stack fault on coprocessor (unused)
SIGSTOP	P1990	Stop	Stop process
SIGTSTP	P1990	Stop	Stop typed at terminal
SIGSYS	P2001	Core	Bad system call (SVr4); see also seccomp(2)
SIGTERM	P1990	Term	Termination signal

.seeshrc???

- ❑ It's just a shell script stored in a specified directory that is used to customize the features of your shell
- ❑ Upon starting your shell you should read in the .seeshrc file
- ❑ Then parse list of commands as if you were typing them on the keyboard and pressing enter
- ❑ This just uses your existing shell functionality
- ❑ It's kind of like redirection

Redirection in UNIX

- ❑ Every process in UNIX inherits 3 open files from its parent.
- ❑ stdin, stdout, stderr
- ❑ In UNIX *everything* is a file that we write to. The screen, network, keyboard input etc.
- ❑ We could also just redirect a text file to stdin and the program will treat it as if it's the keyboard. Or output from stdout and write to a text file.



```
Tutorial3 — ssh iemslie@linux.csc.uvic.ca — 103x44
iemslie@linux4:~/Desktop$ touch input.txt
iemslie@linux4:~/Desktop$ echo 'date' >> input.txt
iemslie@linux4:~/Desktop$ echo 'cal' >> input.txt
iemslie@linux4:~/Desktop$ bash < input.txt
Mon Jan 27 16:39:12 PST 2020
  January 2020
Su Mo Tu We Th Fr Sa
                1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31

iemslie@linux4:~/Desktop$
```


Makefiles

- ❑ Makefiles are used to compile large programs that span multiple files.
- ❑ In C we can define functions in multiple different .c files
- ❑ We can then use .h header files to access the functions in these .c files from other files
- ❑ So for example we might have a linkedlist.c file with a bunch of linked-list functions. Then we create a .h file with prototype function definitions of these same functions.
- ❑ To access these linked-list functions in some other .c file we #include them
- ❑ Makefiles allow us to set up how these files are linked together when they are compiled.
- ❑ Make is just a program that we call when we type 'make' in the command-line
- ❑ It looks for a file called makefile in your current directory and executes the instructions in it

We have 3 files: main.c, linkedlist.c and linkedlist.h. The file called makefile compiles them

main.c

```
1 #include "linkedlist.h"
2
3 int main() {
4     // Function call to another file
5     print_list();
6
7     return 0;
8 }
```

linkedlist.c

```
1 #include <stdio.h>
2 #include "linkedlist.h"
3
4 void print_list() {
5     printf("List is empty\n");
6 }
```

linkedlist.h

```
1
2
3 /* Define a function prototype here */
4 void print_list();
```

makefile

```
1 CFLAGS=-Wall
2
3 example: linkedlist.o main.o
4     gcc -o program linkedlist.o main.o
```

GDB

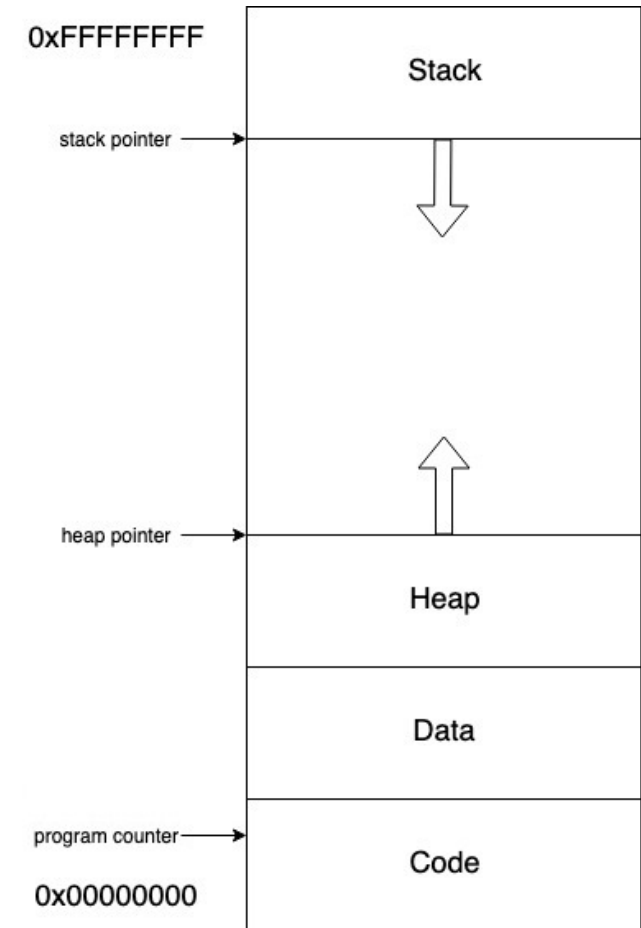
- ❑ GDB gives us access to useful debugging tools such as break points or step-by-step
- ❑ When we want to use GDB we should compile our program using `-g`
- ❑ `gcc -g -o my_program my_program.c`
- ❑ This gives us important information such as variable names etc.
- ❑ We begin GDB by typing the following:
- ❑ `gdb my_program`
- ❑ We can run our program on step at a time within GDB by typing the following:
- ❑ `start`
- ❑ We can then step through using the command:
- ❑ `step`

GDB

- ❑ We can simply run the program as normal by typing:
 - ❑ `run`
- ❑ We can list out our source code using the command:
 - ❑ `list`
- ❑ We can print the value of variables at any point by just saying
 - ❑ `print variable_name`
- ❑ We can set a breakpoint by typing:
 - ❑ `break linenumber -- or b` i.e `break 9` will stop executing at line 9
- ❑ We can disable all breakpoints by just typing
 - ❑ `disable`

Valgrind

- ❑ Variables on the stack are local variables for functions (even main). The stack grows and shrinks as functions are called. Variables are “pushed” and “popped”.
- ❑ If we want to grow an array at runtime we can use malloc to dynamically change its size
- ❑ Memory on the heap is useful because we can reference it from anywhere using pointers and it sticks around until we free it
- ❑ We can request the operating system to create memory on the heap using malloc()
- ❑ malloc() returns a pointer to the beginning of a segment of memory in the heap
- ❑ In C there is no “garbage collection” like in languages like Java or Python
- ❑ This means we must manually call free() for each segment of memory given by malloc



Valgrind

- ❑ If we have a program that runs for a long time and do not free up heap memory then we may get a stack/heap collision
- ❑ Valgrind can be used to determine if we have forgotten to free up any dynamically allocated memory. Use `valgrind --help` to list all commands.
- ❑ `valgrind --tool=memcheck --leak-check=yes --show-reachable=yes --num-callers=20 --track-fds=yes ./test`

```
iemslielinux5c:~/Desktop/CSC360/Tutorial3$ valgrind ./program
==10150== Memcheck, a memory error detector
==10150== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==10150== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==10150== Command: ./program
==10150==
List is empty
==10150==
==10150== HEAP SUMMARY:
==10150==     in use at exit: 1 bytes in 1 blocks
==10150==   total heap usage: 2 allocs, 1 frees, 1,025 bytes allocated
==10150==
==10150== LEAK SUMMARY:
==10150==    definitely lost: 1 bytes in 1 blocks
==10150==    indirectly lost: 0 bytes in 0 blocks
==10150==    possibly lost: 0 bytes in 0 blocks
==10150==    still reachable: 0 bytes in 0 blocks
==10150==    suppressed: 0 bytes in 0 blocks
==10150== Rerun with --leak-check=full to see details of leaked memory
==10150==
==10150== For counts of detected and suppressed errors, rerun with: -v
==10150== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
iemslielinux5c:~/Desktop/CSC360/Tutorial3$
```


Useful Functions

`setenv()`

`unsetenv()`

`signal()`

`waitpid()`

`chdir()`

Quiz

On a sheet of paper with your full name and student ID

Please provide a description of the difference between the heap and the stack.

In case you missed anything...

- ❑ Shells

- ❑ <https://brennan.io/2015/01/16/write-a-shell-in-c/>

- ❑ .bashrc Example

- ❑ <https://www.tldp.org/LDP/abs/html/sample-bashrc.html>

- ❑ Wait

- ❑ <http://man7.org/linux/man-pages/man2/waitid.2.html>

- ❑ Signals

- ❑ <https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/19/lec.html>

- ❑ <http://man7.org/linux/man-pages/man7/signal.7.html>

- ❑ <http://man7.org/linux/man-pages/man2/signal.2.html>

- ❑ <http://man7.org/linux/man-pages/man2/sigaction.2.html>

- ❑ Makefiles

- ❑ <http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/>

- ❑ <https://opensource.com/article/18/8/what-how-makefile>

- ❑ GDB

- ❑ <https://www.youtube.com/watch?v=xQ0ONbt-qPs&t=1s>

- ❑ Valgrind

- ❑ <https://www.youtube.com/watch?v=bb1bTJtqXrI>