CSCI 4061: Processes and Environment

Chris Kauffman

Last Updated:
Wed Jan 30 19:02:22 CST 2019

Logistics

Reading

Stevens and Rago, Ch 7-8

Goals Today

- Finish Basics Overview
- Process Lifecycle
- Killing programs
- Process memory layout

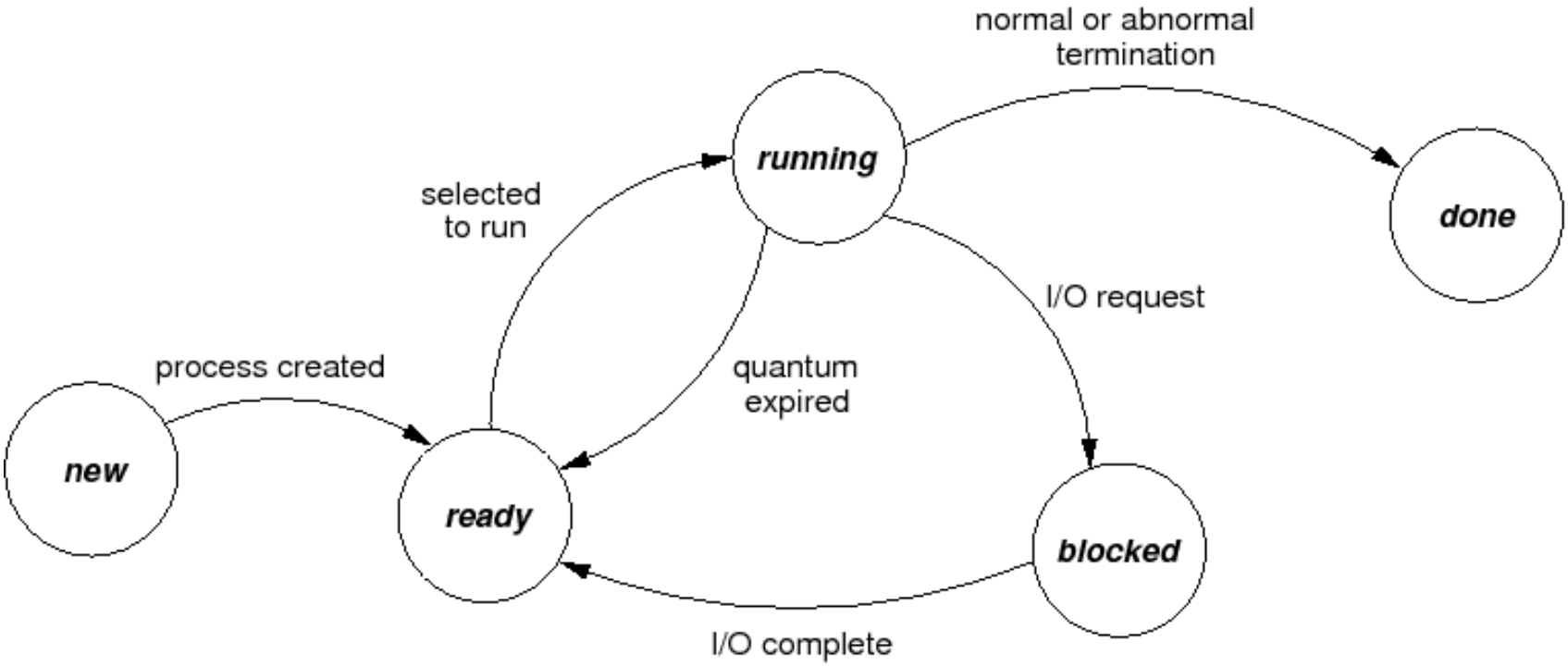
Lab01: Makefiles / Compilation

- Due Monday night
- Feedback?

Reminder: AGREEMENT

Download `AGREEMENT.txt` from the schedule page, sign and submit to Canvas

Process Life Cycle



Source: Saverio Perugini, lecture notes

`ps` and `top` show running process status

These shell commands show a `STAT` or `S` columns corresponding loosely to process states.

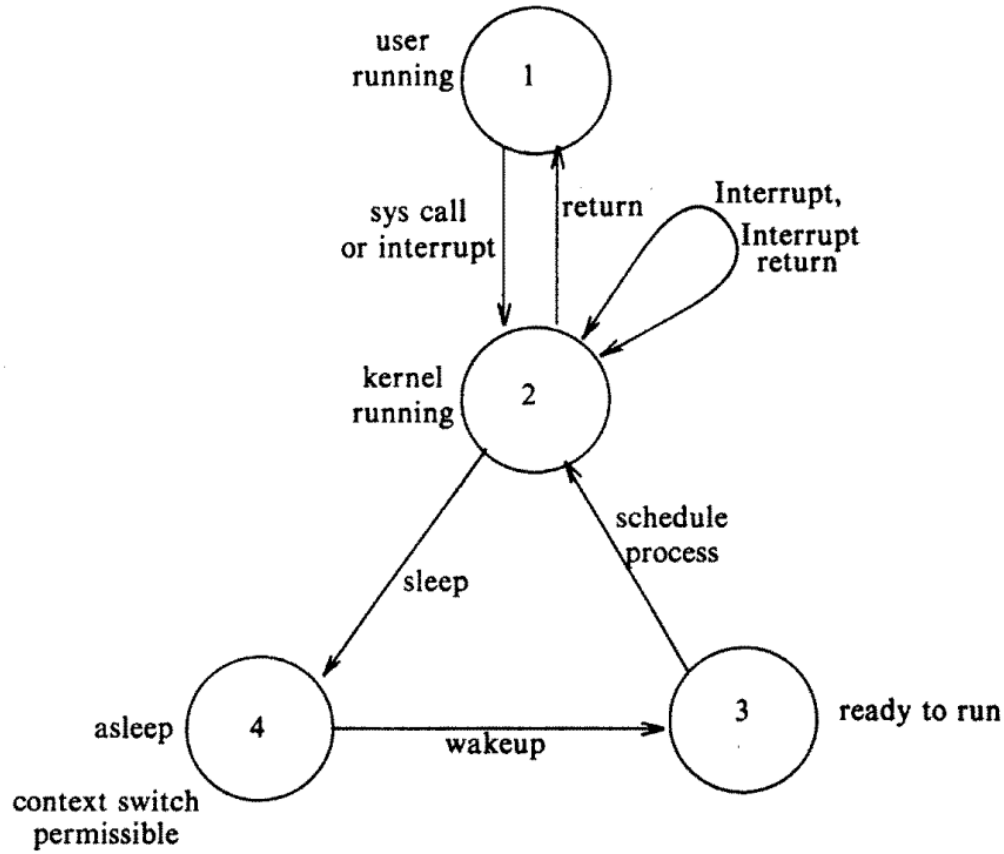
STAT	Meaning
	Common
R	running or runnable (on run queue)
S	interruptible sleep (waiting for an event to complete)
T	stopped, either by a job control signal or being traced.
Z	defunct ("zombie") process, terminated but not reaped by parent.
	Less Common
D	uninterruptible sleep (usually IO)
W	paging (not valid since the 2.6.xx kernel)
X	dead (should never be seen)

Source: `man` page for `ps`

Will talk more about zombies and orphans later

States of a Living Process

- Note inclusion of Kernel/OS here
- **Interrupt and Sys Calls** start running code in the operating system
- Interrupt/Signal can come from software or hardware
- **Context switch** starts running another process, only happens when one process is safely tucked in and put to **sleep**



Source: *Design of the Unix Operating System* by Maurice Bach

Terminal: Foreground/Background Processes

- Type a program into the terminal, press enter
- Stars a process in the **foreground** of the terminal
- Input from user typing, output to terminal screen
- **Suspend** foreground process with `Ctrl-z`, gets prompt back
- **Terminate** foreground process (usually) with `Ctrl-c`
- Run a process in the background
  - Start with `&` at end: `ls &`
  - Move job 2 to bg: `bg %2`
- Move job 2 to foreground: `fg %4`
- Terminals running programs with `jobs`

Murdering Processes

Question: What command ends processes that are misbehaving?

Keystrokes to Remember

Ctrl-c	Send the interrupt signal, kills most processes
Ctrl-z	Send the stop signal, puts process to sleep

Easy to Kill

- `yes` spits output to the screen continuously
- End it from the terminal its started in
- Suspend it then, end it
- Kill it from a different terminal

Harder to Kill

- Consider the program `no_interrupts.c`
- Ignores some common signals
- Need to use the big stick for this one:

```
kill -9 1234     OR
pkill -9 a.out
```

Exercise: Basic Job Control

Give a sequence of commands / keystrokes to...

Misbehaving

- Compile `no_interrupts.c` to a program named `invincible`
- Run `invicible`
- Try to end the process by sending it the interrupt signal
- In a separate terminal, end the `invicible` program

Edit / Build Seq

- Edit `Makefile` with `vi`
- Suspend `vi` (don't quit it)
- Run the `Makefile`
- Terminate before completing build
- Bring back `vi` to continue editing

Recall: Program Memory

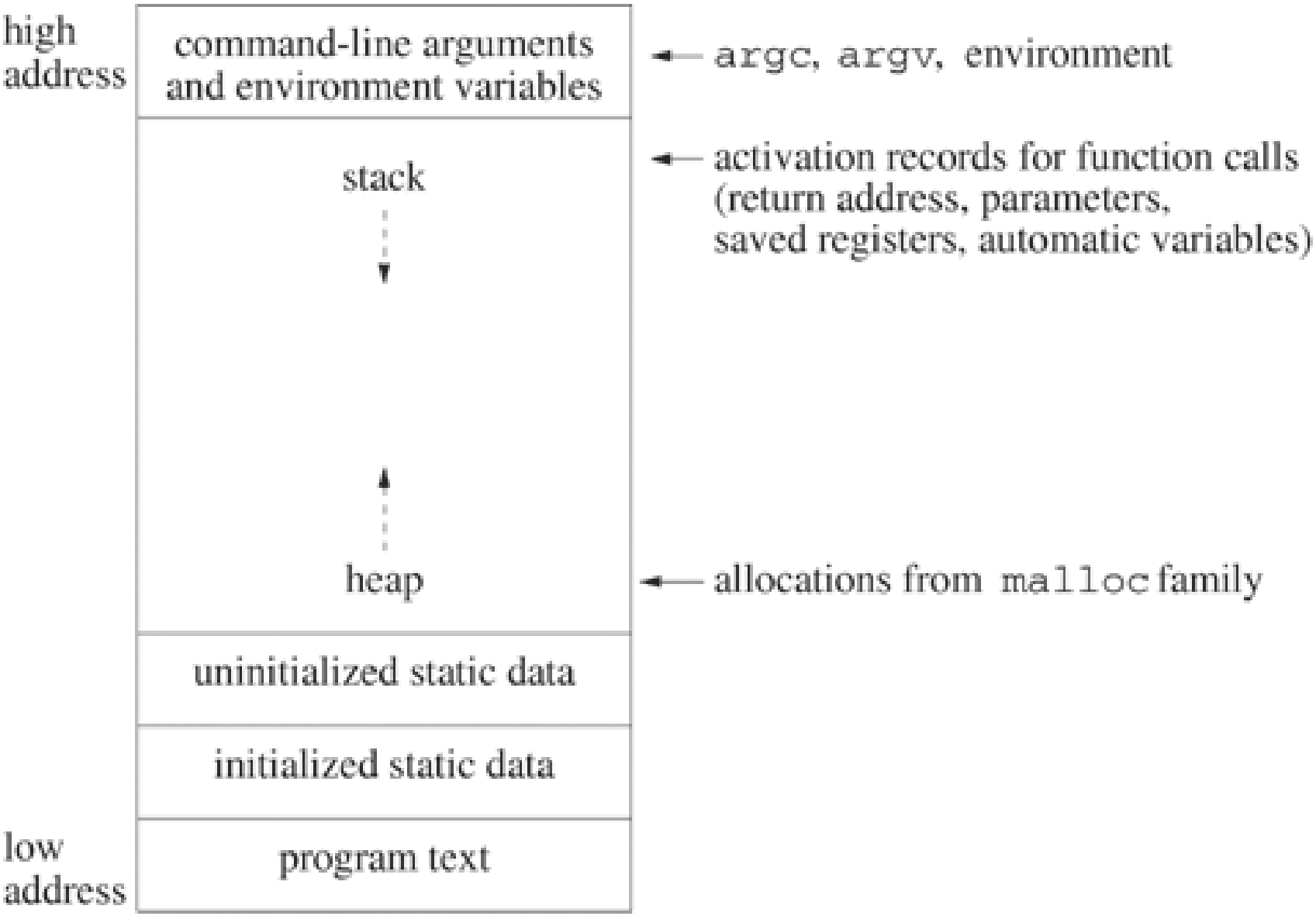
- What are the 4 memory areas to a C program we've discussed OR that you know from previous courses?
- Give an example of how one creates variables/values in each area of memory

Answers: Program Memory

- What are the 4 memory areas to a C program we've discussed OR that you know from previous courses?
  1. Stack: automatic, push/pop with function calls
  2. Heap: `malloc()` and `free()`
  3. Global: variables outside functions, `static` vars
  4. Text: Assembly instructions
- Give an example of how one creates variables/values in each area of memory

```
 1  #include <stdlib.h>
 2  int glob1 = 2;                    // global var
 3  int func(int *a){                 // param stack var
 4    int b = 2 * (*a);               // local stack var
 5    return b;                       // de-allocate locals in func()
 6  }
 7  int main(){                       // main entry point
 8    int x = 5;                      // local stack var
 9    int c = func(&x);               // local stack var
10    int *p = malloc(sizeof(int)); // local stack var that points into heap
11    *p = 10;                        // modify heap memory
12    glob1 = func(p);                // allocate func() locals and run code
13    free(p);                        // deallocate heap mem pointed to p
14    return 0;                       // deallocate locals in main()
15  }
16  // all executable code is in the .text memory area as assmebly instructions
```

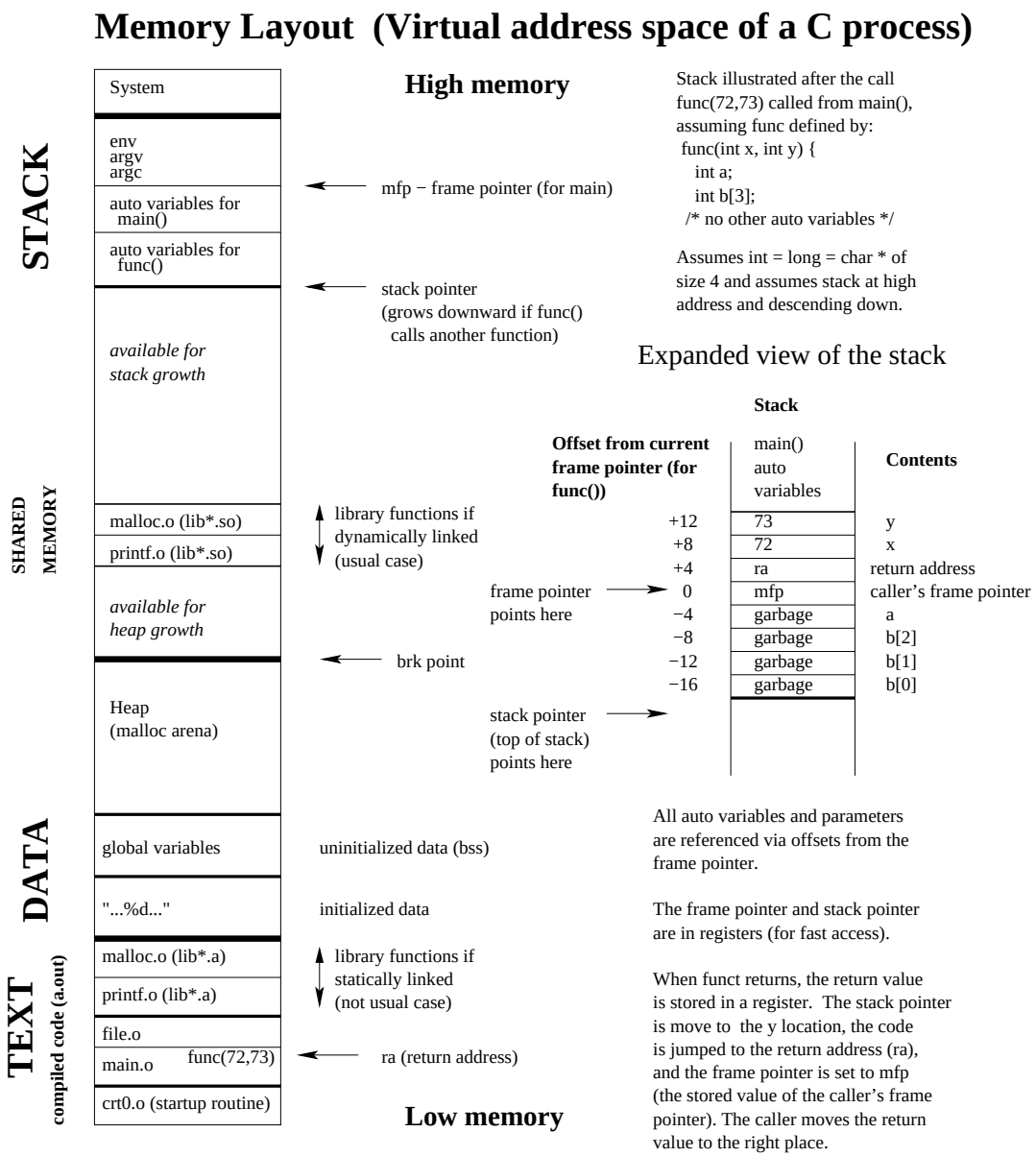
More Detailed Process Memory



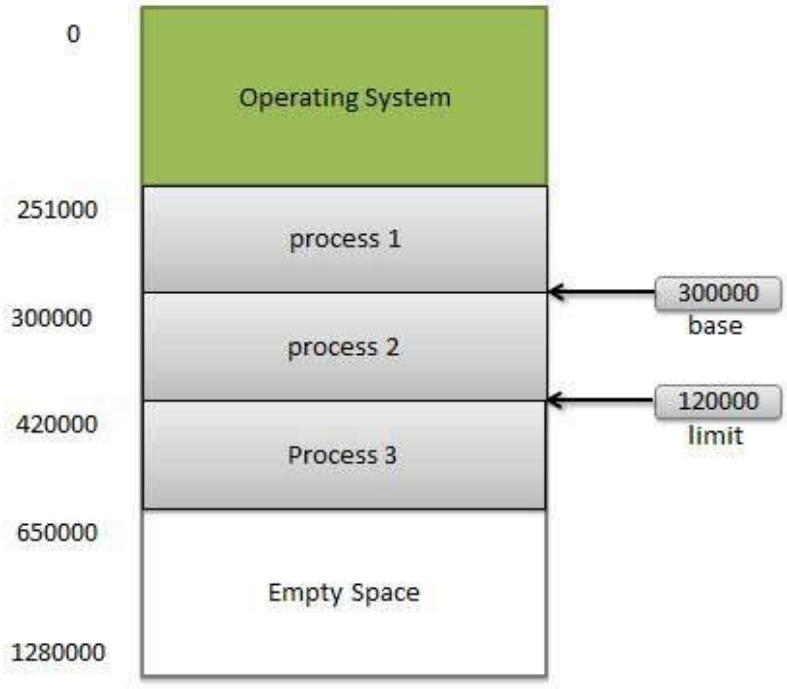
Source: Unix Systems Programming, Robbins & Robbins

Yet *more* detailed view (Link)

A detailed picture of the virtual memory image, by Wolf Holzman



Unix Processes In Memory



Source: Tutorials Point

- Separate Memory Image for Each Process
- OS + Hardware keeps processes inside their own address space
- This is a gross simplification but will suffice until later when we discuss **virtual memory** vs physical memory

This picture should bother you

- Consequence for program dynamic memory allocation?
- Problems with running system calls?

Exercise: Memory Problems in C Programs

What you're up against

- Stack problems: References to stack variables that go away
- Segmentation Faults: Access memory out of bounds for whole program, via heap or via stack
- Null pointers dereference: Often results in a segfault as `NULL` translates to `0x0000` which is off limits
- Use of uninitialized: variables don't have values by default, assign or get something random
- Memory Leaks: `malloc()` memory that is not used but never `free()`'d, program gobbles more and more memory
- Examine results of running `overflow.c`, **EXPLAIN OUTPUT**

Solutions

- Don't program in C
- Use a tool to help identify and fix problems
- Memory Tools on Windows: Discussion here. Synposis → $$$
- Memory Tools on Linux/Mac: **Valgrind** → FREE

Valgrind: Memory Tool on Linux/Mac

Valgrind[1] has Memcheck

- Catches most memory errors[2]
  - Use of uninitialized memory
  - Reading/writing memory after it has been free'd
  - Reading/writing off the end of malloc'd blocks
  - Memory leaks
- Source line of problem happened (but not cause)
- Super easy to use, installed on lab machines
- Slows execution of program *way* down

[1] http://valgrind.org/

[2] http://en.wikipedia.org/wiki/Valgrind

Valgrind in Action

See some common problems in `badmemory.c`

Debuggers

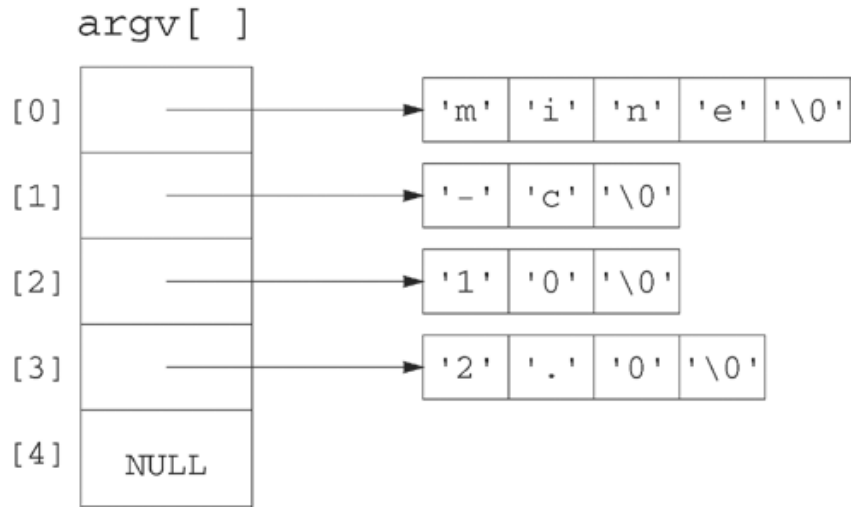
- There comes a day when `printf` just isn't enough
- On that day you will start compiling with `-g` to turn on the debugger
- Then you will run `gdb myprog`, set some breakpoints, and get to the root of the problem
- More on debugger later in a lecture/lab

Command Line Arguments

`int main(int argc, char *argv[])`

2-arg version of `main()` will be set up to have number of arguments and array of strings in it by whatever started it

```
> cat print13.c
#include <stdio.h>
int main(int argc, char *argv[]){
  printf("%s\n",argv[1]);
  printf("%s\n",argv[3]);
}
> gcc print13.c
> ./a.out -c 10 2.0
-c
2.0
```



argc is 4 in this case

Exercise: Print Args

Write a quick C program which prints all of its argv elements as strings. There will be argc elements in this array.

Environment Variables

All programs can access **environment** variables, name/value pairs used to communicate and alter behavior.

Shell show/set variables

Done with `echo $VARNAME`

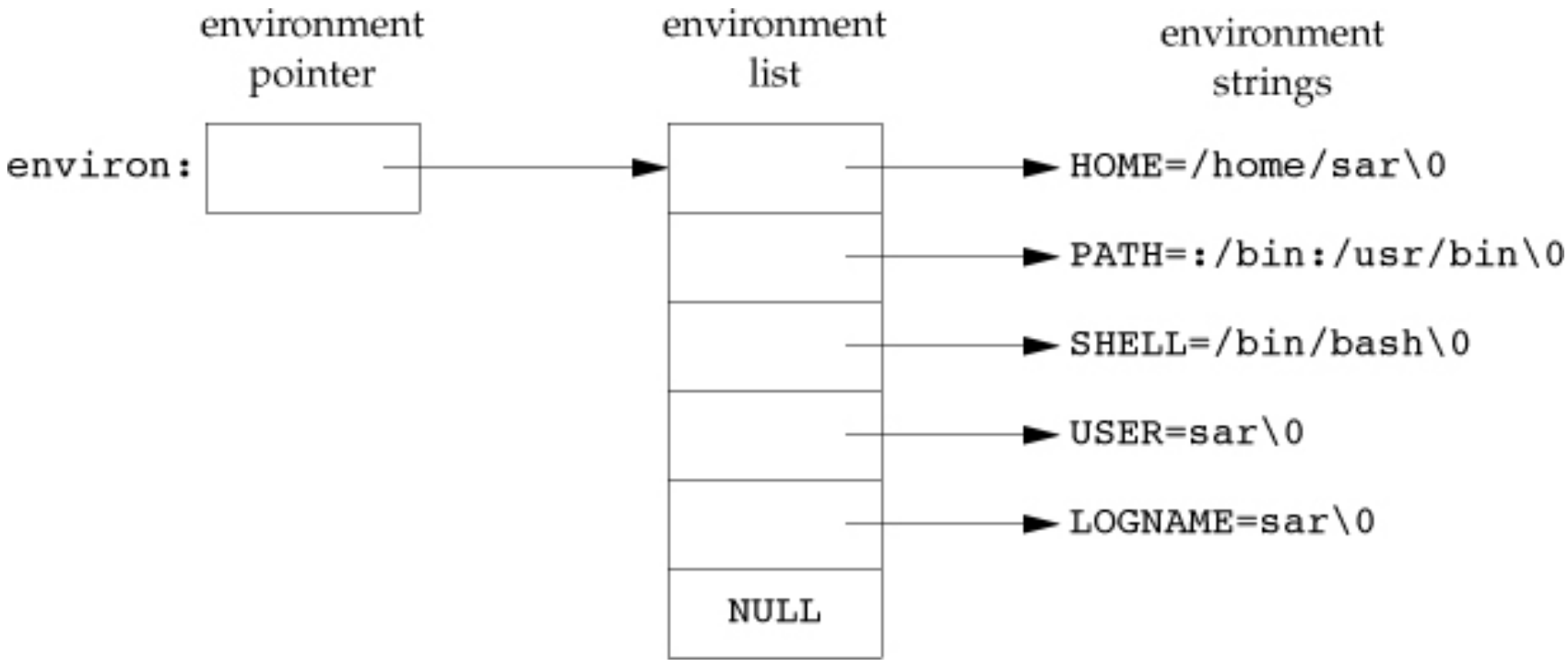
```
> echo $PAGER
less
> PAGER=cat
> echo $PAGER
cat
> echo $PS1
>
> PS1='wicked$ '
wicked$
```

Shell `env`

Show *all* environment

```
> env
JAVA8_HOME=/usr/lib/jvm/java-8-openjdk
PAGER=less
PWD=/home/kauffman/4061-F2017/lectures/03
HOME=/home/kauffman
BROWSER=chromium
COLUMNS=79
MAIL=/var/spool/mail/kauffman
MANPATH=:/home/kauffman/local/man:/home/k
PATH=/usr/local/sbin:/usr/local/bin:/usr/
PS1=>
...
```

C Programs and Environment Vars



- Global variable `char **environ` provides array of environment variables in form `VARNAME=VALUE`, null terminated
- Easier to use the library functions to check/change environment

C Library for Environment Vars

The C Library Provides standard library functions for manipulating environment variables.

```
#include <stdlib.h>

char *getenv(const char *name);
// returns pointer to value associated with name, NULL if not found

int setenv(const char *name, const char *value, int rewrite);
// sets name to value. If name already exists in the environment, then
// (a) if rewrite is nonzero, the existing definition for name is
// first removed; or (b) if rewrite is 0, an existing definition for
// name is not removed, name is not set to the new value,and no error
// occurs. return: 0 if OK, -1 on error

int unsetenv(const char *name);
// removes any definition of name. It is not an error if such a
// definition does not exist. return: 0 if OK, -1 on error

int putenv(char *str);
// str is of form NAME=VALUE, alters environment accordingly. If name
// already exists, its old definition is first removed. Don't use with
// stack strings. Returns: 0 if OK, nonzero on error.
```

Exercise: Manipulate Environment Vars

Write a short C program which behaves as indicated in the demo

- Prints `ROCK` and `VOLUME` environment variables
- If `ROCK` is set to anything, change `VOLUME` to "11"

Use these functions

```
char *getenv(const char *name);
// NULL if name not sot
// otherwise pointer to value

int setenv(const char *name,
           const char *value,
           int rewrite);
// Change name value pair,
// if rewrite is 1,
// overwrite previous definitions
```

Note the use of `export` to ensure child processes see the environment variables

```
> unset ROCK
> unset VOLUME
> gcc environment_vars.c
> a.out
ROCK not set
VOLUME is not set
> export VOLUME=7
> a.out
ROCK not set
VOLUME is 7
> export ROCK=yes
> a.out
ROCK is yes
Turning VOLUME to 11
VOLUME is 11
> echo $VOLUME
7
```

Note also that the program does not change the shell's values for `ROCK`: no child can change a parent's values (or mind)

Answers: Manipulate Environment Vars

See 03-process-basics-code/environment_vars.c

```
 1  // environment_vars.c: solution to in-class exercise showing how to
 2  // check and set environment variables via the standard getenv() and
 3  // setenv() functions.
 4  #include <stdlib.h>
 5  #include <stdio.h>
 6
 7  int main(int argc, char *argv[]){
 8
 9    char *verbosity = getenv("ROCK");
10    if(verbosity == NULL){
11      printf("ROCK not set\n");
12    }
13    else{
14      printf("ROCK is %s\n",verbosity);
15      printf("Turning VOLUME to 11\n");
16      int fail = setenv("VOLUME","11",1);
17      if(fail){
18        printf("Couldn't change VOLUME\n");
19      }
20    }
21    char *volume = getenv("VOLUME");
22    if(volume == NULL){
23      volume = "not set";
24    }
25    printf("VOLUME is %s\n",volume);
26    return 0;
27  }
```