

CSCI 331:  
Introduction to Computer Security

Lecture 4: Even more C

Instructor: Dan Barowy

Williams

1

Topics

More pointers

Segmentation faults

Makefiles

Static vs shared libraries

2

Your to-dos

1. Lab 1 **out**.
  - i. Note that it includes some reading.
  - ii. Lab 1 **due Sunday 9/24** by **10:00pm**.
  - iii. If your RPi is not set up, what are you waiting for?
  - iv. Office hours Thur & Fri.
2. Read *On User Choice in Graphical Password Schemes* **by Thur, 9/21** and **take notes** to discuss in class.

3

Announcements

- CS Colloquium this **Friday, Sept 22 @ 2:35pm in Wege Auditorium (TCL 123)**



Your classmates

What I Did Last Summer, Industry Edition

Short presentations by your fellow CS students about internship experiences in industry. CS Colloquium credit awarded for attendance.

4

## Activity

5

## Activity solution: caveat

The C specification says **nothing** about the **location** of a variable.

The words **stack** and **heap** literally **do not appear** in the document.

It only says how short-lived (**automatic**) and long-lived (**allocated**) storage should **behave**.

Virtually every compiler uses the **stack** for **automatic** variables, and the **heap** for **allocated** variables.

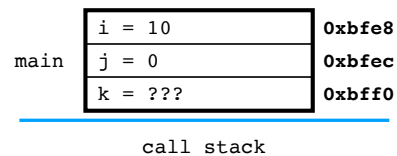
Practically, it **does not matter where** you put your variables as long as you put them in **stack** and **heap** locations as appropriate.

6

## Activity solution: after step 1

```
#include <stdio.h>

int main() {
    int i = 10, j = 0, *k;
    k = &i;
    *k = 20;
    k = &j;
    *k = i;
    printf("i = %d,
           j = %d,
           *k = %d\n",
           i, j, *k);
    return 0;
}
```



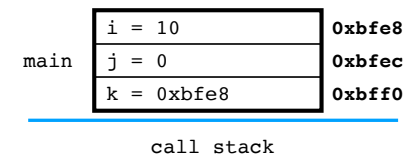
(state **just before** the line indicated by the **arrow** is executed)

7

## Activity solution: after step 2

```
#include <stdio.h>

int main() {
    int i = 10, j = 0, *k;
    k = &i;
    *k = 20;
    k = &j;
    *k = i;
    printf("i = %d,
           j = %d,
           *k = %d\n",
           i, j, *k);
    return 0;
}
```



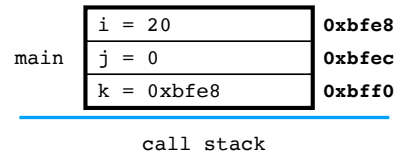
(state **just before** the line indicated by the **arrow** is executed)

8

## Activity solution: after step 3

```
#include <stdio.h>

int main() {
    int i = 10, j = 0, *k;
    k = &i;
    *k = 20;
    k = &j;
    *k = i;
    printf("i = %d,
           j = %d,
           *k = %d\n",
           i, j, *k);
    return 0;
}
```



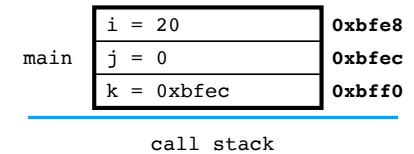
(state **just before** the line indicated by the **arrow** is executed)

9

## Activity solution: after step 4

```
#include <stdio.h>

int main() {
    int i = 10, j = 0, *k;
    k = &i;
    *k = 20;
    k = &j;
    *k = i;
    printf("i = %d,
           j = %d,
           *k = %d\n",
           i, j, *k);
    return 0;
}
```



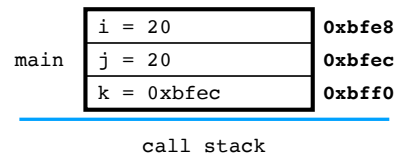
(state **just before** the line indicated by the **arrow** is executed)

10

## Activity solution: after step 5

```
#include <stdio.h>

int main() {
    int i = 10, j = 0, *k;
    k = &i;
    *k = 20;
    k = &j;
    *k = i;
    printf("i = %d,
           j = %d,
           *k = %d\n",
           i, j, *k);
    return 0;
}
```



(state **just before** the line indicated by the **arrow** is executed)

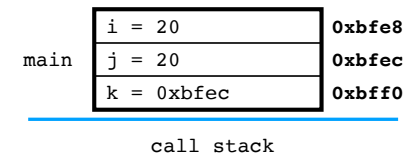
11

## Activity solution: print output

**printf prints "i = 20, j = 20, \*k = 20"**

```
#include <stdio.h>

int main() {
    int i = 10, j = 0, *k;
    k = &i;
    *k = 20;
    k = &j;
    *k = i;
    printf("i = %d,
           j = %d,
           *k = %d\n",
           i, j, *k);
    return 0;
}
```



(state **just before** the line indicated by the **arrow** is executed)

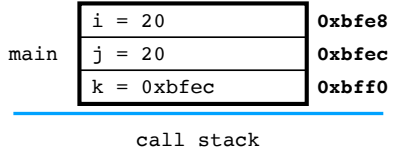
12

## Activity solution: static data?

Yes. "i = %d,\n j = %d,\n \*k = %d\n"

```
#include <stdio.h>

int main() {
    int i = 10, j = 0, *k;
    k = &i;
    *k = 20;
    k = &j;
    *k = i;
    printf("i = %d,
           j = %d,
           *k = %d\n",
           i, j, *k);
    return 0;
}
```



i = 20	0xbfe8
j = 20	0xbfec
k = 0xbfec	0xbff0

call stack

(state **just before** the line indicated by the **arrow** is executed)

13

How might you verify my solution?

14

gdb

15

What is a variable (in a program)?

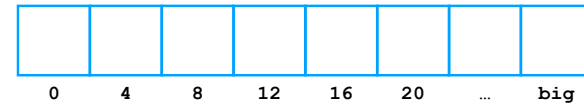
16

What is a segmentation fault?

17

## Question

Physical memory:



Problem: we typically want to run **multiple programs at the same time**.

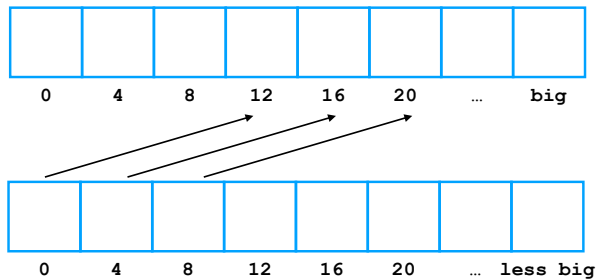
Any program that uses memory must say **where in memory** its variables are stored.

**How do we know** that another program isn't already using the space we reserve for our variable?

18

Answer: virtual memory

Physical memory:



Virtual memory maps **virtual addresses** to **physical addresses**.

19

## OS view of memory

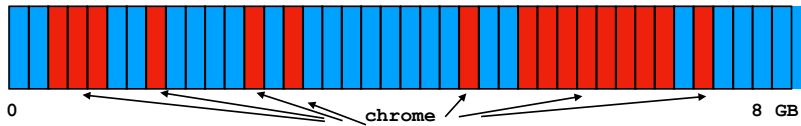


Each of these programs is given the **illusion** that it controls **all of memory**.

20

## OS view of memory

A virtual memory map is not necessarily contiguous.



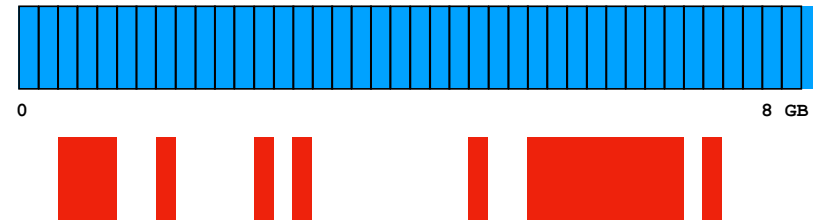
Memory is broken into chunks called **pages**.

**Allocation requests** for programs are filled from pages; when a page fills, the memory manager maps a new page to the program's **virtual address space**.

21

## OS view of memory

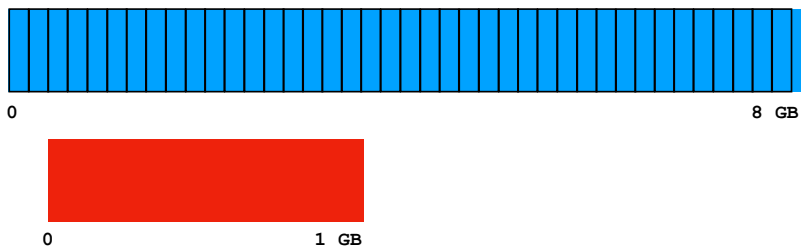
The program "sees" a contiguous address space.



22

## OS view of memory

The program "sees" a contiguous address space.



23

What is a segmentation fault?

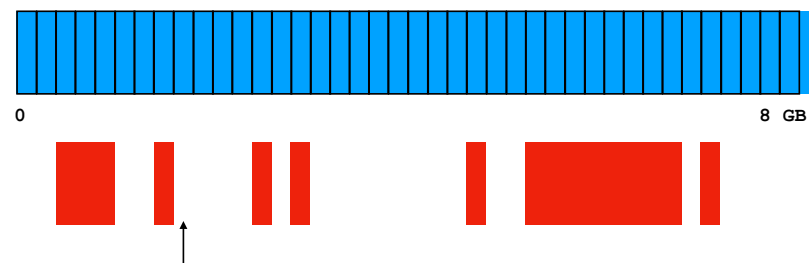
24

## Segmentation fault

A **segmentation fault** (often shortened to **segfault**) or **access violation** is a fault, or failure condition, raised by hardware with memory protection, notifying an operating system that the software has **attempted to access a restricted area of memory**.

25

## Segmentation fault



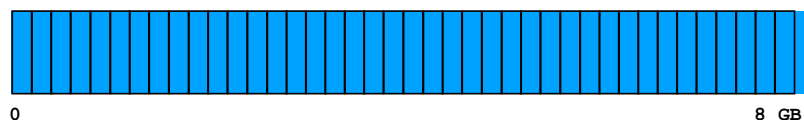
Reading here will **fail**. Also...

26

## Address zero

Address zero in virtual memory is **special**.

Attempting to read or write to address 0 **segfaults**.



0

```
char *buf1 = malloc(MAX_INT);  
char *buf2 = "something";  
strncpy(buf1, buf2, strlen(buf2));
```

This is why NULL equals 0. Protects against alloc fail.  
Segmentation faults are a **safety feature**.

27

## Does this segfault?

A 4k page of memory (vaddr 4096-8191):



You ask malloc for 3 bytes and get

You write "eph" into this buffer.

Does the write segfault? **No!**

Write did not **cross page boundary**.

28

## Makefiles

29

## Makefiles

A **Makefile** is a **specification** used by the **make** tool to **automate** the compilation of programs.

30

## Rationale

Programmers build software **frequently**.



**Lazy**  
(don't want to retype)



**Impatient**  
(don't want to wait for gcc)

31

## Insight

An entire project does not need to be rebuilt every time.

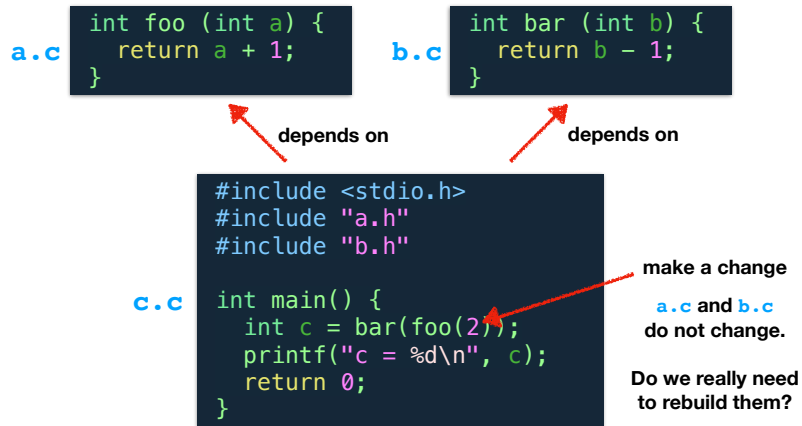


32



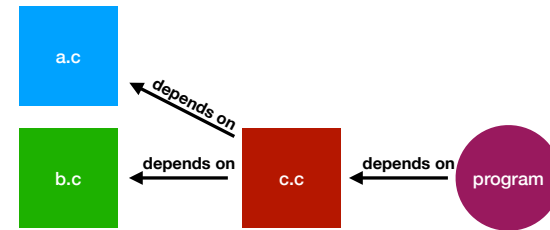
## Insight

An entire project does not need to rebuilt every time.



33

## A Makefile encodes dependencies



```
$ gcc a.c b.c c.c -o program
```

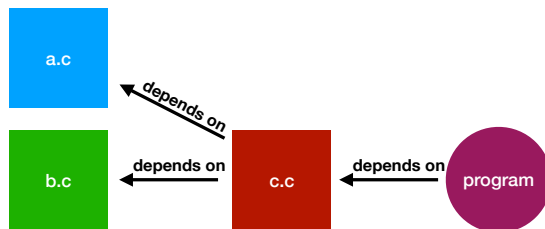
Small catch: **make** can only avoid rebuilding if there is a **produced thing** that it can avoid rebuilding.

There is only one **produced thing** here: **program**

(produced things are circles; source files are squares)

34

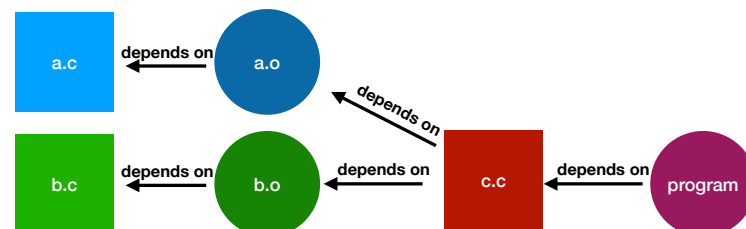
## A Makefile encodes dependencies



Fix: make more **produced things**.

35

## A Makefile encodes dependencies



Fix: make more **produced things**.

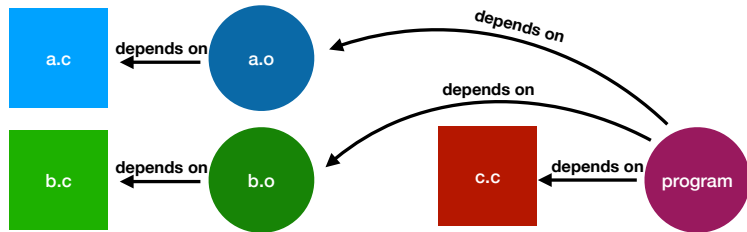
This still has a problem.

**c.c** is not a **produced thing**.

Only **produced things** can depend on other things.

36

## A Makefile encodes dependencies

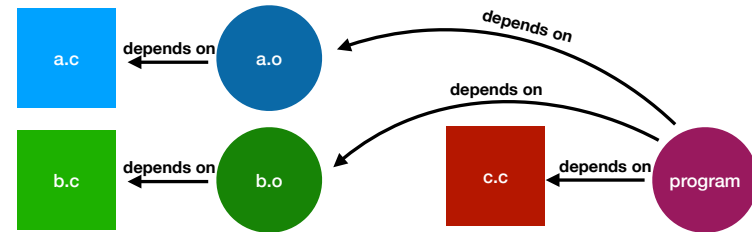


Fix: make `program` depend on `a.o` and `b.o`.

Observe: The same amount of work is being done. But the **things** are **smaller**.

37

## A Makefile encodes dependencies



Suppose we update `c.c`.

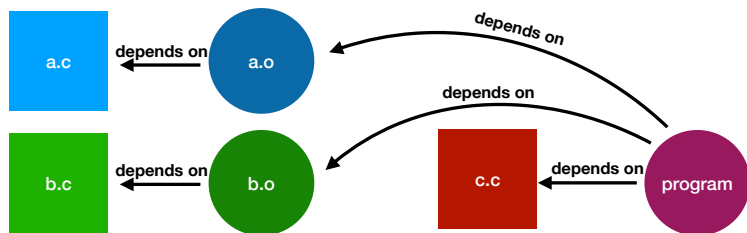
What needs to be rebuilt?

Just `program`.

We don't need to rebuild `a.o` or `b.o` at all.

38

## A Makefile encodes dependencies



Let's write a `Makefile` for this, starting with `program`.

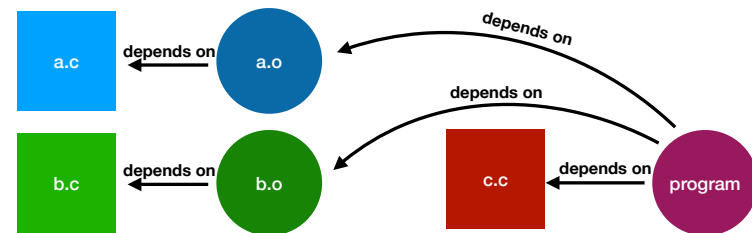
```

program: c.c b.o a.o
tab → gcc -o program c.c b.o a.o
  
```

3 things, 3 rules.

39

## A Makefile encodes dependencies



Let's write a `Makefile` for this, starting with `program`.

```

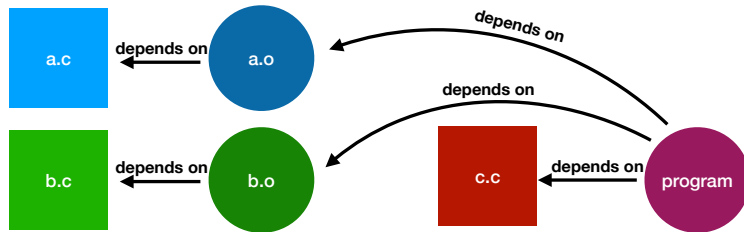
program: c.c b.o a.o
tab → gcc -o program c.c b.o a.o

b.o: b.c
tab → gcc -c b.c
  
```

3 things, 3 rules.

40

## A Makefile encodes dependencies



Let's write a Makefile for this, starting with `program`.

```
program: c.c b.o a.o
tab → gcc -o program c.c b.o a.o

b.o: b.c
tab → gcc -c b.c

a.o: a.c
tab → gcc -c a.c
```

3 things, 3 rules.

41

## Makefile syntax

```
program: c.c b.o a.o
tab → gcc -o program c.c b.o a.o
```

```
target: dep1 ... depn
tab → command
```

`command` should produce `target`.

42

## Recap & Next Class

### Today we learned:

- Stack layouts
- Makefiles
- Static vs. shared libraries

### Next class:

- Pseudoterminals
- Password security

43