CS/COE 0447

The Stack: The Untold Story

wilkie (with content borrowed from:
Jarrett Billingsley
Dr. Bruce Childers)

What is the Stack?

- In previous lectures, we introduced the stack.
- It is full of mysterious behavior:
  - “Allocating” on the stack (making room) has you *subtract* from its base address.
- Let’s visit this from a different direction.
- Let’s consider… the problem itself.
  - And how we might solve it.

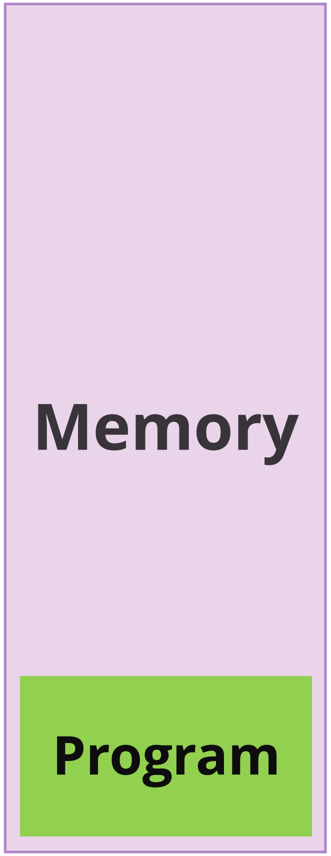
The Problem

- We have a program. It uses memory.
- We don’t know exactly how much memory we need.
  - It may depend on **how long the program runs**.
  - Or the **size of the data** it is working on (arbitrarily specified by a human being, perhaps)
  - Maybe our program **responds to the available memory** by choosing a different algorithm when it has more or less.

- Either way, a program does not have a static allocation of memory.
- How do we allow a program to ***allocate*** memory on-demand?

Our Example: Video Editor

- Let’s consider a video editing program.
  - But thankfully ignore all of the actual video details!
- Data is large, and the memory usage is relative to the size of our video.
- We want memory to be continuous.
  - Could you imagine if data were all broken up?
  - Your program would be difficult to code if an array was broken up.
    - Our array addressing math would no longer be general and would cease to work well. (You’d have multiple array base addresses)

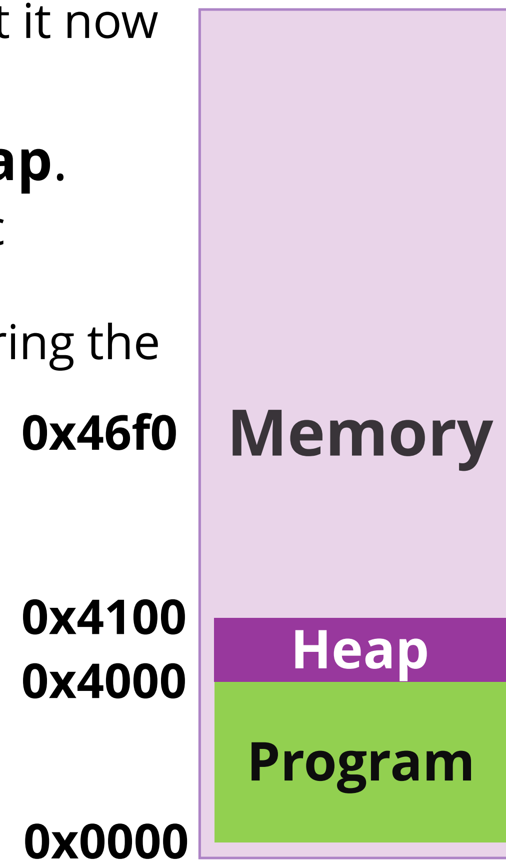


Allocating Memory

- You'll learn a lot more about this in CS 449
  - But it's worth sequence breaking and talking about it now

- We will maintain a section of memory: the **heap**.
  - The heap is a section of memory used for dynamic memory.
  - **Dynamic memory** is memory that is allocated during the runtime of a program and may be reclaimed later.
  - "heap" a very conflated term, unfortunately.

- When we allocate memory, we add it to **the end** of the heap.
  - It's like appending to an array.
  - Look at it go!



Revisiting Functions: A Problem Arises

- Now, consider functions.

- When we call a function, we need to remember where we were.
  - This is stored in the $**ra** register.
  - But if we call a function twice, what happens to $**ra**?
    - It is overwritten, and our first value in $*ra* is lost.
    - This means after our second function is called, the first function will now be lost, and it will return *to itself*. (Refer to the previous slides)

- What are our strategies for remembering **ra**?

Remembering RA

- Bad Idea #1: **Place it in another register**

```
myFunction:
    move t0, ra

    # overwrites ra!
    jal myOtherFunction

    # it's ok though:
    move ra, t0
    jr ra
```

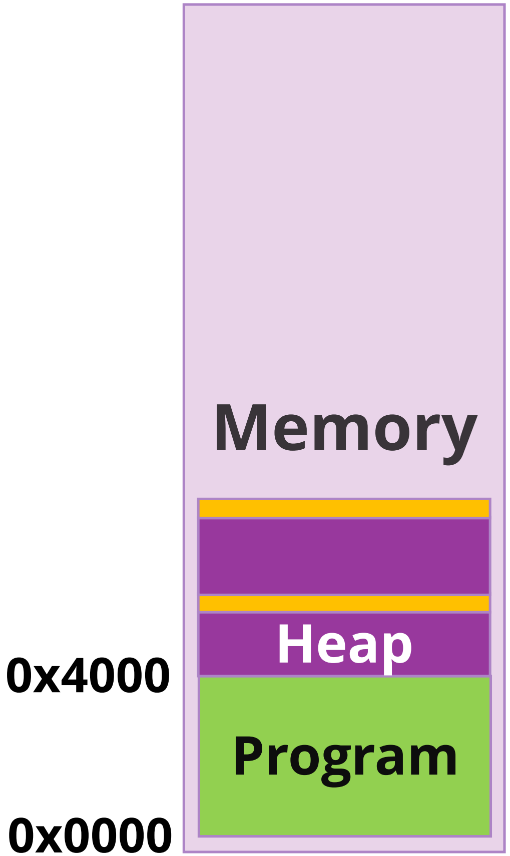
However:

- What if myOtherFunction uses **t0**?
- Ok, t0 isn't preserved, so let's use **s0**.
- Wait… ***we*** need to preserve **s0**…
- Where do we put that?? **s1???**
- **Wait… we need to preserve s1!!**

- We will run out of saved registers and we cannot trust unsaved registers. (other functions may overwrite them)
- Therefore, we need memory.

Remembering RA

- We need memory. We have that *heap thing*.
- So can't we just **allocate some on the heap**?
- Sure can. But it is *Bad Idea #2*.
- What happens if that function allocates memory?
- And then calls another function.
- And then we return…
- And return from the first function…
- Leaving gaps in our memory!



Let's Design a Memory Layout (kinda)

- Our video editing application wants to use large, continuous memory regions.
  - Videos are big things! (Continuous memory makes things easier/faster… future courses will convince you.)

- We have very few registers, and need to remember $**ra**
  - So, we need to place **ra** in memory to recall it before we **jr ra**
- However, placing it with other program memory creates gaps
  - This is very very trash!!

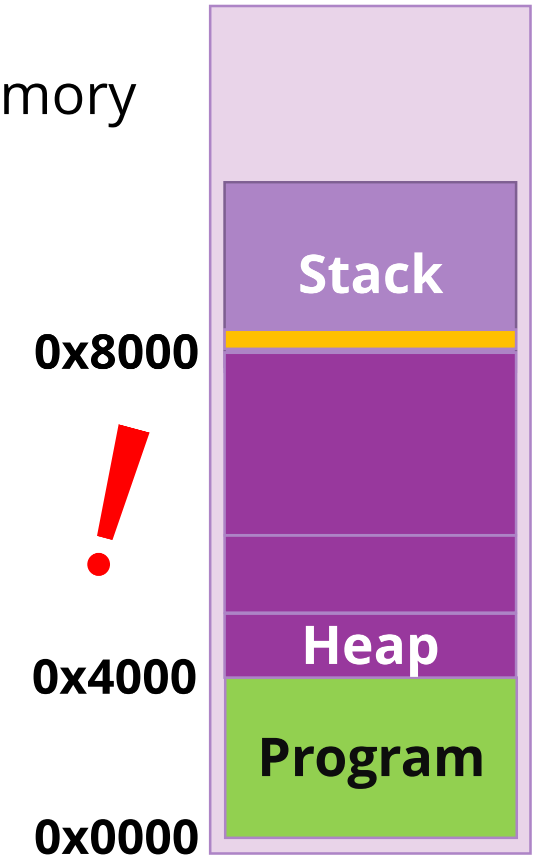
- How do we solve this.
  - Occam's Razor to the rescue… and it will create a very weird situation.
  - One that involves subtracting to allocate…

Solving our Problem: Step 1

- How can we use memory, but not create gaps?

- Good [rational] Idea: Maintain two dynamic memory sections.
- We call our function.
- What happens if that function allocates memory?
- And then calls another function.
- And then we return…
- And return from the first function… WHEW! No gaps.

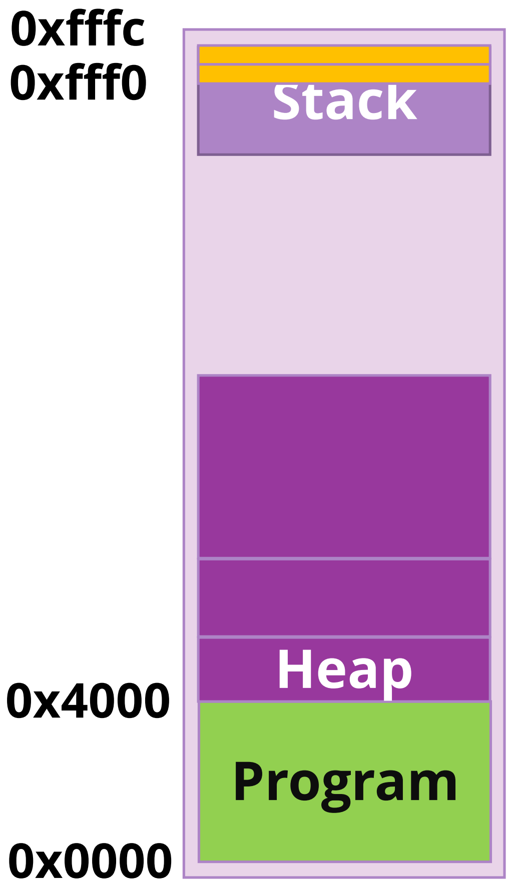
- (Ok, but now we start editing a LARGE video…)
  - Uh oh! We've lost our $**ra**



Solving our Problem: Step 2

- Good [weird] Idea: Maintain two dynamic memory sections. One of which starts at the **highest** memory address. Allocate via **subtraction** (append to bottom)
- We call our function.
- What happens if that function allocates memory?
- And then calls another function.
- And then we return…
- And return from the first function… No gaps.

- As for our large memory case…
- It's fine! (only problem: running out of memory)
  - But, my goodness, you have a bigger problem, then.



Solving our Problem: Step 2

- Good [weird] Idea: Maintain two dynamic memory sections. One of which starts at the **highest** memory address. Allocate via **subtraction** (append to bottom)
- We call our function. **(*subtract* $sp, store)**
- What happens if that function allocates memory?
- And then calls another function. (**sub, store)**
- And then we return… **(load, *add* to $sp)**
- And return from the first function… **(load, add)**
- Refer to the previous slides on the Stack with this knowledge in your mind.