Tutorial 1

CS2106: Introduction to Operating Systems

Outline

1. Module Overview
2. Recap
3. Tutorial Questions
    1. MIPS Assembly Revision
    2. Stack Frames
    3. Process Memory

Module Overview

- What will you learn in CS2106?
  - OS Structure and Architecture
  - Process Management
    - Process Abstraction
    - Process Scheduling
    - Threads
    - Inter Process Communication
    - Process Synchronization
  - Memory Management
  - File System Management
  - OS Protection Mechanism

First Half

Second Half

Mentioned throughout the module

Objectives

- To understand how to set up and tear down a stack frame during function invocation in an example using MIPS assembly.
- To understand how memory is allocated to a typical process using a C program as an example.

Recap

Lecture Contents

Reference: https://www.comp.nus.edu.sg/~cs2100/lect/MIPS_Reference_Data_page1.pdf

MIPS Registers

Program Counter (abbreviated as $PC or simply PC) is a special register that store the address of the instruction being executed in the processor. This register is updated automatically by the processor.

Name	Number	Use	Preserved Across A Call
`$zero`	`0`	The constant value 0	N.A.
`$at`	`1`	Assembler Temporary	No
`$v0 - $v1`	`2 – 3`	Values for Function Results and Evaluation Expressions	No
`$a0 - $a3`	`4 – 7`	Argument	No
`$t0 - $t7`	`8 – 15`	Temporaries	No
`$s0 - $s7`	`16 – 23`	Saved Temporaries	Yes
`$t8 - $t9`	`24 – 25`	Temporaries	No
`$k0 - $k1`	`26 – 27`	Reserved for OS Kernel	No
`$gp`	`28`	Global Pointer	Yes
`$sp`	`29`	Stack Pointer	Yes
`$fp`	`30`	Frame Pointer	Yes
`$ra`	`31`	Return Address	Yes

MIPS Instructions

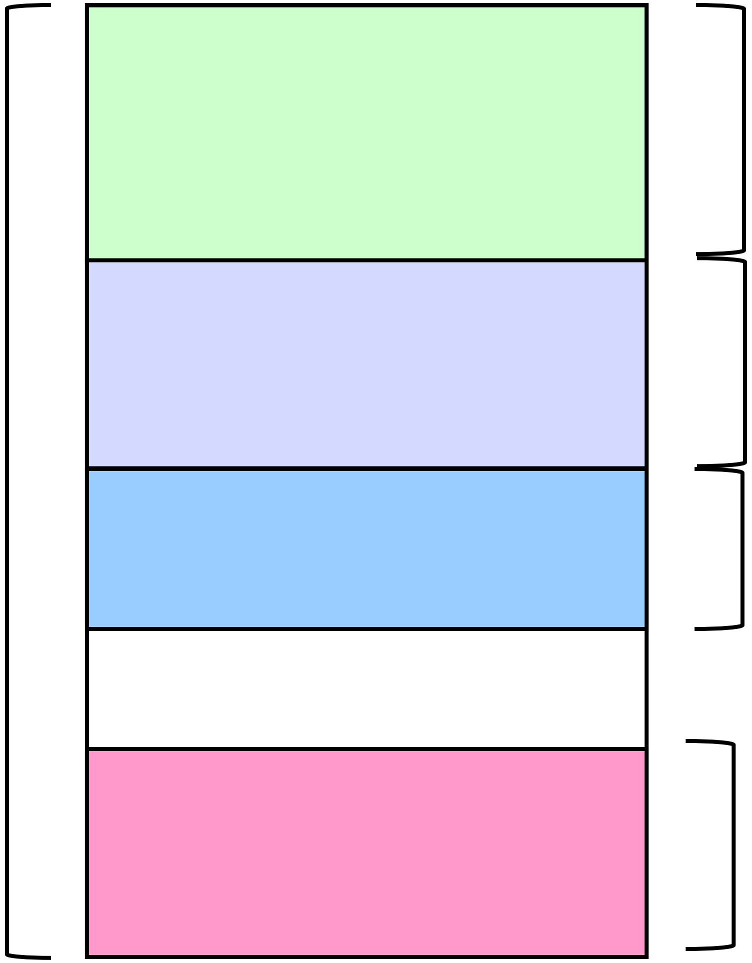
Sufficient for this tutorial

Note:
`0($t0)` = (content of register $t0) + 0

- `add $dst, $src1, $src2`
- `addi $dst, $src, immediate`
- `sll $dst, $src, immediate`
- `lw $destination, offset($source)`
  - `lw $t1, 0($t0)`
  - Load content in location `$t0 + 0` to register `$t1`
- `sw $source, offset($destination)`
  - `sw $t1, 0($t0)`
  - Store content in register `$t1` to location `$t0 + 0`
- `la $destination, label`
  - `la $t1, b`
  - Load address of label `b` into register `$t1`
- `li $R1, value`
  - `li $t1, 10`
  - Loads `10` to register `$t1`

References:
https://www.comp.nus.edu.sg/~cs2100/lect/MIPS_Reference_Data_page1.pdf
https://courses.cs.washington.edu/courses/cse378/09au/MIPS_Green_Sheet.pdf

Regions of Memory Space in a Process



The entire memory space

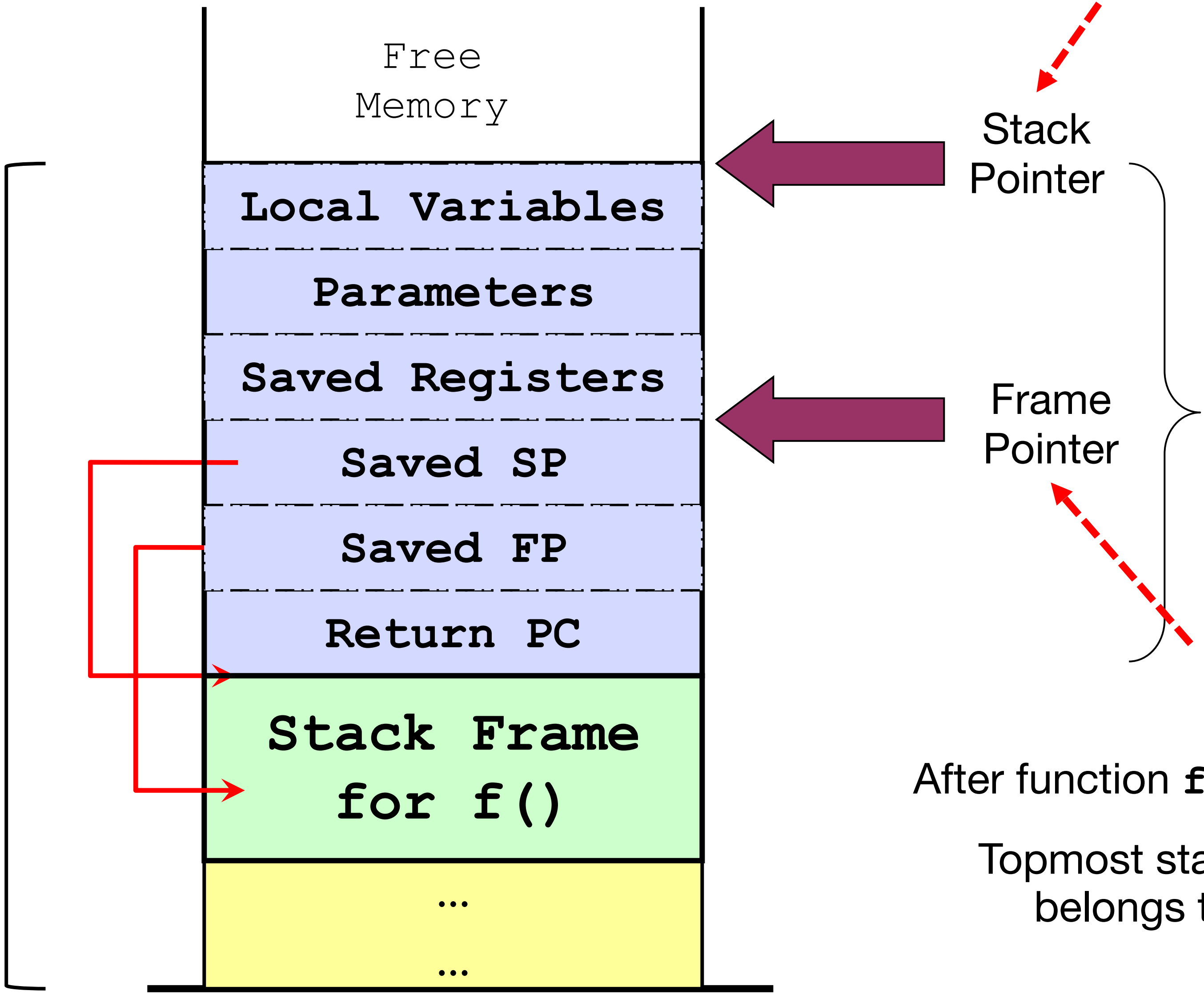
Text
(for instructions)

Data
(for global variables)

Heap
(for dynamic allocation)

Stack
(for function invocations)

Stack Frame



Stack Frame for g()

Points to the first unused location of the stack region.

Points to a fixed location in a stack frame

After function `f()` calls `g()`

Topmost stack frame belongs to `g()`

MIPS Assembly Revision

Section 1

MIPS Assembly Programming Revision

You may assume the following:

- Execution always begins at the label “`main:`”
- When a program ends, it hands control back to the operating system using the following two instructions
  - `li $v0, 10`
  - `syscall`
- The availability of pseudo instructions like
  - load immediate (`li`)
  - load address (`la`)
  - move (`mov`).

MIPS Assembly Programming Revision

Example:

```
b = a + 10;
```

```
main: la $t0, a
      lw $t1, 0($t0)
      addi $t1, $t1, 10
      la $t0, b
      sw $t1, 0($t0)
      li $v0, 10
      syscall
```

Control handed back to OS

MIPS Assembly Programming Revision

- Function calls in MIPS are performed using `jal` and `jr`.

Labels	Address	Instruction	Comments
`main:`	`0x1000`	`addi $t0, $zero, 5`	
	`0x1004`	`jal func`	; Jump-and-link to func. Address 0x1008 put ; into $ra. ; Program Counter now pointing to 0x1010
	`0x1008`	`li $v0, 10`	; Exit to OS
	`0x100C`	`syscall`	
`func:`	`0x1010`	`addi $t0, $t0, 5`	
	`0x1014`	`jr $ra`	; Jump to 0x1008 to exit function

Question 1

- Write the following C program in MIPS assembly.
- We will explore passing parameters **<u>using registers instead of stack frames</u>**.
- Use `$a0` and `$a1` to pass parameters to the function f, and `$v0` to pass results back. You will need to use the MIPS `jal` and `jr` instructions.
- All variables are initially in memory, and you can use the `la` pseudo instruction to load the address of a variable into a register.

Question 1

Assume a, b and y are labels that have been defined in the .data section of a MIPS source file

C

```
int f(int x,y){
       return 2*(x+y);
}

int a = 3, b = 4, y;

int main() {
  y = f(a, b);
}
```

MIPS

```
f:      add $t1, $a0, $a1     ;  x + y
        sll $v0, $t1, 1       ;  2 * (x + y)
        jr  $ra               ;  Return to caller

main:   la  $t0, a            ;  Load a
        lw  $a0, 0($t0)       ;
        la  $t0, b            ;  Load b
        lw  $a1, 0($t0)       ;
        jal f                 ;  Call function f
        la  $t0, y            ;
        sw  $v0, 0($t0)       ;
        li  $v0, 10           ;  Exit to OS
        syscall               ;
```

Question 2

- In this question we explore how the stack and frame pointers on MIPS work.
- The MIPS stack pointer is called `$sp` (register `$29`) while the frame pointer is called `$fp` (register `$30`).
- Unlike many other processors like those made by ARM and Intel, the MIPS processor does not have “push” and “pop” instructions.
- Instead, we manipulate `$sp` directly:

Pushing a value in $r0 onto the stack.	**Popping a value from the stack to $s0:**
`sw $r0, 0($sp)` `addi $sp, $sp, 4`	`addi $sp, $sp, -4` `lw $s0, 0($sp)`

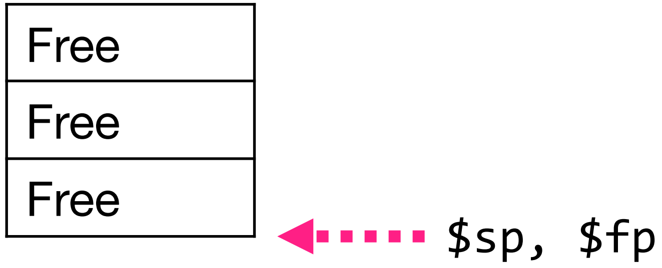
Question 2

Pushing a value in $r0 onto the stack.	Popping a value from the stack to $s0:
`sw $r0, 0($sp)` `addi $sp, $sp, 4`	`addi $sp, $sp, -4` `lw $s0, 0($sp)`

Repeat Question 1 above using the stack to pass arguments and results instead of $a0, $a1 and $v0.

```
int a=3, b=4, y;

int main(){
        y=f(a,b)
}
```

```
main:   addi $fp, $sp, 0     ; Save $sp. mov $fp, $sp also works
        addi $sp, $sp, 8     ; Reserve 2 integers for stack frame
        la   $t0, a          ; Load a
        lw   $t0, 0($t0)     ;
        sw   $t0, 0($fp)     ; Write a to stack frame
        la   $t0, b          ; Load b
        lw   $t0, 0($t0)     ;
        sw   $t0, 4($fp)     ; Write b to stack frame
        jal f                ; Call f

        lw    $t1, 0($fp)    ; Get the result from the stack frame
        la    $t0, y         ; Store into y
        sw    $t1, 0($t0)    ;
        addi $sp, $s0, -8    ; Pop off stack frame
        li    $v0, 10        ; Exit to OS
        syscall              ;
```



Stack

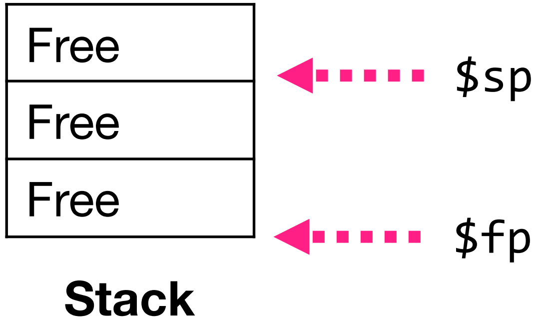
Question 2

Pushing a value in $r0 onto the stack.	Popping a value from the stack to $s0:
`sw $r0, 0($sp)` `addi $sp, $sp, 4`	`addi $sp, $sp, -4` `lw $s0, 0($sp)`

Repeat Question 1 above using the stack to pass arguments and results instead of $a0, $a1 and $v0.

```
int a=3, b=4, y;

int main(){
        y=f(a,b)
}
```

```
main:   addi $fp, $sp, 0    ; Save $sp. mov $fp, $sp also works
        addi $sp, $sp, 8    ; Reserve 2 integers for stack frame
        la   $t0, a         ; Load a
        lw   $t0, 0($t0)    ;
        sw   $t0, 0($fp)    ; Write a to stack frame
        la   $t0, b         ; Load b
        lw   $t0, 0($t0)    ;
        sw   $t0, 4($fp)    ; Write b to stack frame
        jal f               ; Call f

        lw    $t1, 0($fp)   ; Get the result from the stack frame
        la    $t0, y        ; Store into y
        sw    $t1, 0($t0)   ;
        addi $sp, $s0, -8   ; Pop off stack frame
        li    $v0, 10       ; Exit to OS
        syscall             ;
```



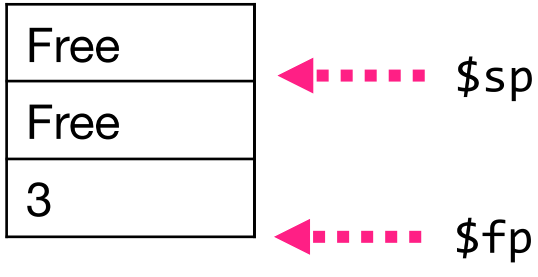
Question 2

Pushing a value in $r0 onto the stack.	Popping a value from the stack to $s0:
`sw $r0, 0($sp)` `addi $sp, $sp, 4`	`addi $sp, $sp, -4` `lw $s0, 0($sp)`

Repeat Question 1 above using the stack to pass arguments and results instead of $a0, $a1 and $v0.

```
int a=3, b=4, y;

int main(){
       y=f(a,b)
}
```

```
main:   addi $fp, $sp, 0   ; Save $sp. mov $fp, $sp also works
        addi $sp, $sp, 8   ; Reserve 2 integers for stack frame
        la  $t0, a         ; Load a
        lw  $t0, 0($t0)    ;
        sw  $t0, 0($fp)    ; Write a to stack frame
        la  $t0, b         ; Load b
        lw  $t0, 0($t0)    ;
        sw  $t0, 4($fp)    ; Write b to stack frame
        jal f              ; Call f

        lw   $t1, 0($fp)   ; Get the result from the stack frame
        la   $t0, y        ; Store into y
        sw   $t1, 0($t0)   ;
        addi $sp, $s0, -8  ; Pop off stack frame
        li   $v0, 10       ; Exit to OS
        syscall            ;
```



Stack

Question 2

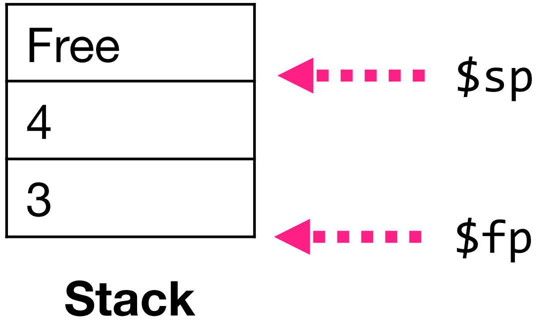
Pushing a value in $r0 onto the stack.	Popping a value from the stack to $s0:
`sw $r0, 0($sp)` `addi $sp, $sp, 4`	`addi $sp, $sp, -4` `lw $s0, 0($sp)`

Repeat Question 1 above using the stack to pass arguments and results instead of $a0, $a1 and $v0.

```
int a=3, b=4, y;

int main(){
       y=f(a,b)
}
```

```
main:   addi $fp, $sp, 0    ; Save $sp. mov $fp, $sp also works
        addi $sp, $sp, 8    ; Reserve 2 integers for stack frame
        la   $t0, a         ; Load a
        lw   $t0, 0($t0)    ;
        sw   $t0, 0($fp)    ; Write a to stack frame
        la   $t0, b         ; Load b
        lw   $t0, 0($t0)    ;
        sw   $t0, 4($fp)    ; Write b to stack frame
        jal f               ; Call f
```

```
        lw    $t1, 0($fp)   ; Get the result from the stack frame
        la    $t0, y        ; Store into y
        sw    $t1, 0($t0)   ;
        addi $sp, $s0, -8   ; Pop off stack frame
        li    $v0, 10       ; Exit to OS
        syscall             ;
```



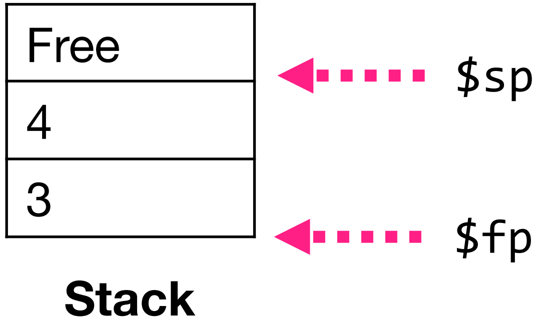
Question 2

Pushing a value in $r0 onto the stack.	Popping a value from the stack to $s0:
`sw $r0, 0($sp)` `addi $sp, $sp, 4`	`addi $sp, $sp, -4` `lw $s0, 0($sp)`

Repeat Question 1 above using the stack to pass arguments and results instead of $a0, $a1 and $v0.

```
int f(int x,y){
       return 2*(x+y);
}
```

```
f:      lw   $t0, 0($fp)          ; Get first parameter
        lw   $t1, 4($fp)          ; Get second parameter
        add $v0, $t0, $t1         ; $v0 = $t0 + $t1
        sll $v0, $v0, 1           ; $v0 = 2 * ($t0 + $t1)
        sw   $v0, 0($fp)          ; Store result
        jr   $ra                  ; Return to caller
```

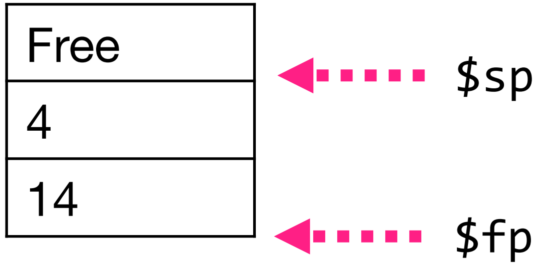

Free
4
3
$sp
$fp
Stack

Question 2

Pushing a value in $r0 onto the stack.	Popping a value from the stack to $s0:
`sw $r0, 0($sp)` `addi $sp, $sp, 4`	`addi $sp, $sp, -4` `lw $s0, 0($sp)`

Repeat Question 1 above using the stack to pass arguments and results instead of $a0, $a1 and $v0.

```
int f(int x,y){              f:      lw  $t0, 0($fp)          ; Get first parameter
       return 2*(x+y);               lw  $t1, 4($fp)          ; Get second parameter
}                                    add $v0, $t0, $t1        ; $v0 = $t0 + $t1
                                     sll $v0, $v0, 1          ; $v0 = 2 * ($t0 + $t1)
                                     sw  $v0, 0($fp)          ; Store result
                                     jr  $ra                  ; Return to caller
```



Free

4

14

Stack

Question 2

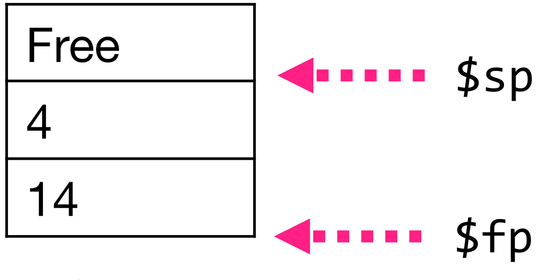
Pushing a value in $r0 onto the stack.	Popping a value from the stack to $s0:
`sw $r0, 0($sp)` `addi $sp, $sp, 4`	`addi $sp, $sp, -4` `lw $s0, 0($sp)`

Repeat Question 1 above using the stack to pass arguments and results instead of $a0, $a1 and $v0.

```
int a=3, b=4, y;

int main(){
       y=f(a,b)
}
```

```
main:   addi $fp, $sp, 0    ; Save $sp. mov $fp, $sp also works
        addi $sp, $sp, 8    ; Reserve 2 integers for stack frame
        la   $t0, a         ; Load a
        lw   $t0, 0($t0)    ;
        sw   $t0, 0($fp)    ; Write a to stack frame
        la   $t0, b         ; Load b
        lw   $t0, 0($t0)    ;
        sw   $t0, 4($fp)    ; Write b to stack frame
        jal f               ; Call f
```

```
        lw    $t1, 0($fp)   ; Get the result from the stack frame
        la    $t0, y        ; Store into y
        sw    $t1, 0($t0)   ;
        addi $sp, $s0, -8   ; Pop off stack frame
        li    $v0, 10       ; Exit to OS
        syscall             ;
```



Stack

Question 2

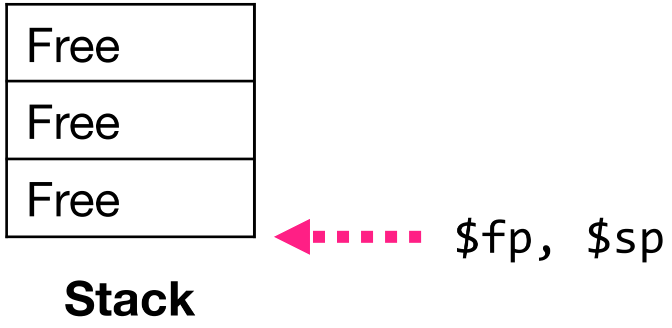
Pushing a value in $r0 onto the stack.	Popping a value from the stack to $s0:
`sw $r0, 0($sp)` `addi $sp, $sp, 4`	`addi $sp, $sp, -4` `lw $s0, 0($sp)`

Repeat Question 1 above using the stack to pass arguments and results instead of $a0, $a1 and $v0.

```
int a=3, b=4, y;

int main(){
        y=f(a,b)
}
```

```
main:   addi $fp, $sp, 0    ; Save $sp. mov $fp, $sp also works
        addi $sp, $sp, 8    ; Reserve 2 integers for stack frame
        la  $t0, a          ; Load a
        lw  $t0, 0($t0)     ;
        sw  $t0, 0($fp)     ; Write a to stack frame
        la  $t0, b          ; Load b
        lw  $t0, 0($t0)     ;
        sw  $t0, 4($fp)     ; Write b to stack frame
        jal f               ; Call f
```

```
        lw   $t1, 0($fp)    ; Get the result from the stack frame
        la   $t0, y         ; Store into y
        sw   $t1, 0($t0)    ;
        addi $sp, $s0, -8   ; Pop off stack frame
        li   $v0, 10        ; Exit to OS
        syscall             ;
```



Free

Free

Free

$fp, $sp

Stack

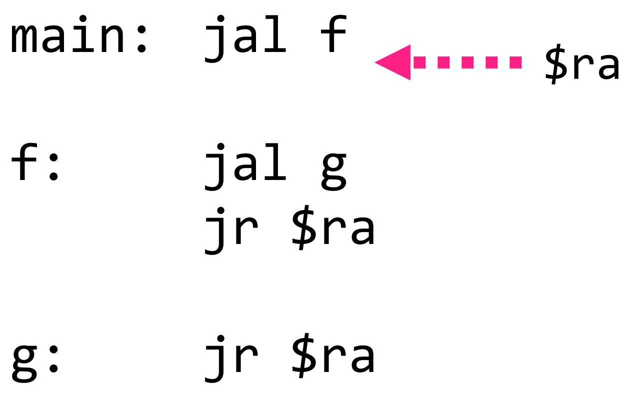
Stack Frame

Section 2

Question 3

Can your approach in Questions 1 and 2 above work for recursive or even nested function calls? Explain why or why not.

- The solution as provided does not support nesting as it does not save `$ra` in the stack and retrieve it before `jr $ra`.



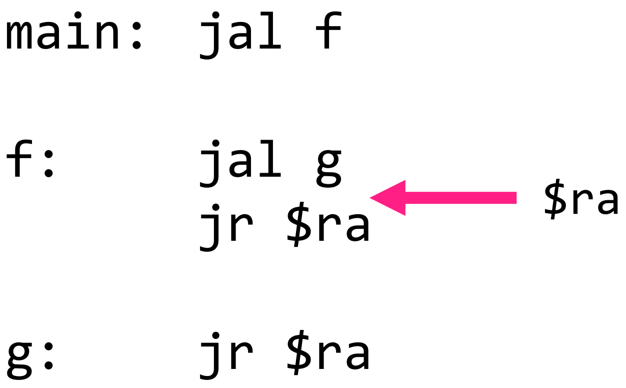
```
main:  jal f

f:     jal g
       jr $ra

g:     jr $ra
```

- `$ra` contains the return address for a function call.
- Suppose we have a function `main()` which calls function `f()`.
- When function `f()` calls function `g()`, `$ra` is updated to the instruction after “`jal g`”
- When we execute the instruction “`jr $ra`” in `f()`, the PC does not point back to the instruction to resume in `main`.

Question 3

Can your approach in Questions 1 and 2 above work for recursive or even nested function calls? Explain why or why not.

- The solution as provided does not support nesting as it does not save `$ra` in the stack and retrieve it before `jr $ra`.



```
main:  jal f

f:     jal g
       jr $ra

g:     jr $ra
```

- $ra contains the return address for a function call.
- Suppose we have a function `main()` which calls function `f()`.
- When function `f()` calls function `g()`, `$ra` is updated to the instruction after “`jal g`”
- When we execute the instruction “`jr $ra`” in `f()`, the PC does not point back to the instruction to resume in `main`.

Question 4

- We now explore making use of a proper stack frame to implement our function call from Question 1.
- Our stack frame looks like this when calling a function.

Saved registers
$ra
Parameter n
…
Parameter 2
Paremeter 1
Saved `$sp`
Saved `$fp`

Notice that the difference here from Slide 36 of Lecture 2 is that over here, the callee saves the return PC on the stack.

Question 4

We follow this convention (callee = function being called). Assume that initially `$sp` is pointing to the bottom of the stack.

Caller:	1. Push `$fp` and `$sp` to stack
	2. Copy `$sp` to `$fp`
	3. Reserve sufficient space on stack for parameters by adding to `$sp`
	4. Write parameters to stack using offsets from `$fp`
	5. `jal` to callee
Callee:	1. Push `$ra` to stack
	2. Push registers we intend to use onto the stack
	3. Use `$fp` to access parameters
	4. Compute result
	5. Write result to stack
	6. Restore registers we saved from the stack
	7. Get `$ra` from the stack
	8. Return to caller by doing `jr $ra`
Caller	1. Get result from stack
	2. Restore `$sp` and `$fp`

Question 4

- Caller needs to reserve 20 bytes:
  - 8 bytes for `$sp` and `$fp`
  - 8 bytes to pass `a` and `b`
  - 4 bytes for callee to save `$ra`

Offset from `$fp`	Contents
0	`$fp`
4	`$sp`
8	`a`
12	`b`
16	`$ra`

Question 4 - Main

Note:
You can also follow the offset table from the previous slide to help you better visualize the what is stored in the stack frame

C

```
int a=3, b=4, y;

int main(){
        y=f(a,b)
}
```

Save $fp
Copy $sp to $fp
Save $sp to stack frame
Reserve 20 bytes on stack frame

MIPS

```
main: sw $fp, 0($sp)          ;
      mov $fp, $sp            ;
      sw $sp, 4($sp)          ;
      addi $sp, $sp, 20       ;
```

20		
16		
12		
8		
4		
0	$fp	← $sp

Question 4 - Main

C

```
int a=3, b=4, y;

int main(){
      y=f(a,b)
}
```

MIPS

```
main: sw $fp, 0($sp)          ;
      mov $fp, $sp            ;
      sw $sp, 4($sp)          ;
      addi $sp, $sp, 20       ;
```

Save $fp

Copy $sp to $fp

Save $sp to stack frame

Reserve 20 bytes on stack frame

20	
16	
12	
8	
4	
0	$fp

$sp, $fp

Question 4 - Main

C

```
int a=3, b=4, y;

int main(){
        y=f(a,b)
}
```

MIPS

```
main: sw $fp, 0($sp)          ;
      mov $fp, $sp            ;
      sw $sp, 4($sp)          ;
      addi $sp, $sp, 20       ;
```

Save `$fp`

Copy $sp to `$fp`

Save `$sp` to stack frame

Reserve 20 bytes on stack frame

20	
16	
12	
8	
4	$sp
0	$fp

`$sp, $fp`

Question 4 - Main

C

```
int a=3, b=4, y;

int main(){
      y=f(a,b)
}
```

MIPS

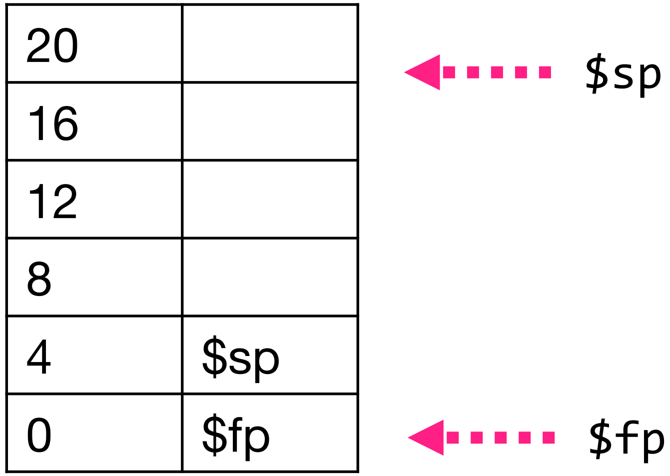
```
main: sw $fp, 0($sp)          ;
      mov $fp, $sp            ;
      sw $sp, 4($sp)          ;
      addi $sp, $sp, 20       ;
```

Save $fp

Copy $sp to $fp

Save $sp to stack frame

Reserve 20 bytes on stack frame



20	
16	
12	
8	
4	$sp
0	$fp

Question 4 - Main

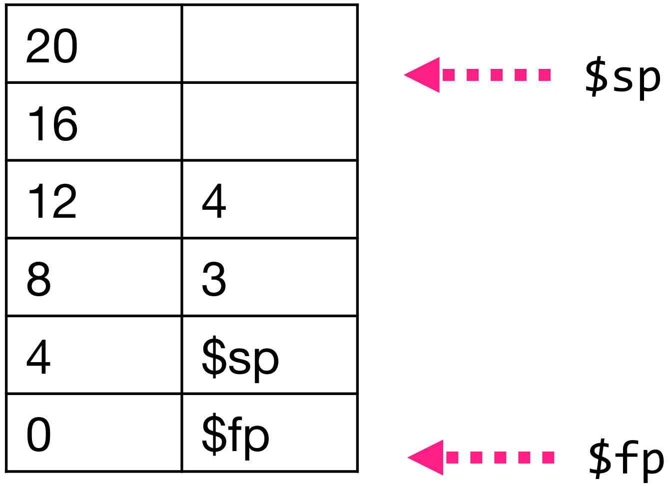
C

```
int a=3, b=4, y;

int main(){
      y=f(a,b)
}
```

MIPS

```
la   $t0, a       ;
lw   $t0, 0($t0)  ;
sw   $t0, 8($fp)  ;
la   $t0, b       ;
lw   $t0, 0($t0)  ;
sw   $t0, 12($fp) ;
jal f             ;
```

Same logic as before (Question 2)



20	
16	
12	4
8	3
4	$sp
0	$fp

Question 4 - f

C

```
int f(int x,y){
      return 2*(x+y);
}
```

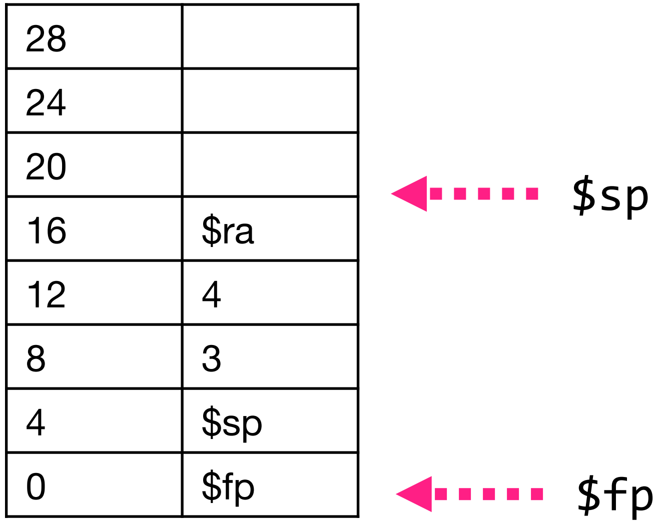
MIPS

```
f:    sw   $ra, 16($fp)      ;
      addi $sp, $sp, 8       ;
      sw   $t0, 20($fp)      ;
      sw   $t1, 24($fp)      ;
```

Save `$ra`

Reserve 8 bytes on stack to store registers we want to use for f

Save `$t0` and `$t1` which we will use



28	
24	
20	
16	$ra
12	4
8	3
4	$sp
0	$fp

Question 4 - f

C

```
int f(int x,y){
        return 2*(x+y);
}
```

MIPS

```
f:    sw   $ra, 16($fp)      ;
      addi $sp, $sp, 8       ;
      sw   $t0, 20($fp)      ;
      sw   $t1, 24($fp)      ;
```

Save `$ra`

Reserve 8 bytes on stack to store registers we want to use for f

Save `$t0` and `$t1` which we will use

28		`$sp`
24	$t1	
20	$t0	
16	$ra	
12	4	
8	3	
4	$sp	
0	$fp	`$fp`

Question 4 - f

C

```
int f(int x,y){
     return 2*(x+y);
}
```

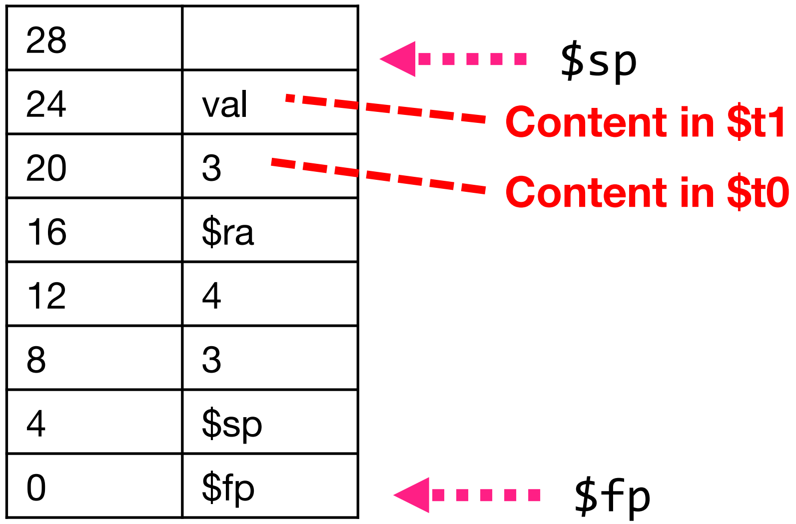
MIPS

```
f:    sw   $ra, 16($fp)     ;
      addi $sp, $sp, 8      ;
      sw   $t0, 20($fp)     ;
      sw   $t1, 24($fp)     ;
```

Save `$ra`

Reserve 8 bytes on stack to store registers we want to use for f

Save `$t0` and `$t1` which we will use



28	
24	val
20	3
16	$ra
12	4
8	3
4	$sp
0	$fp

Question 4 - f

C

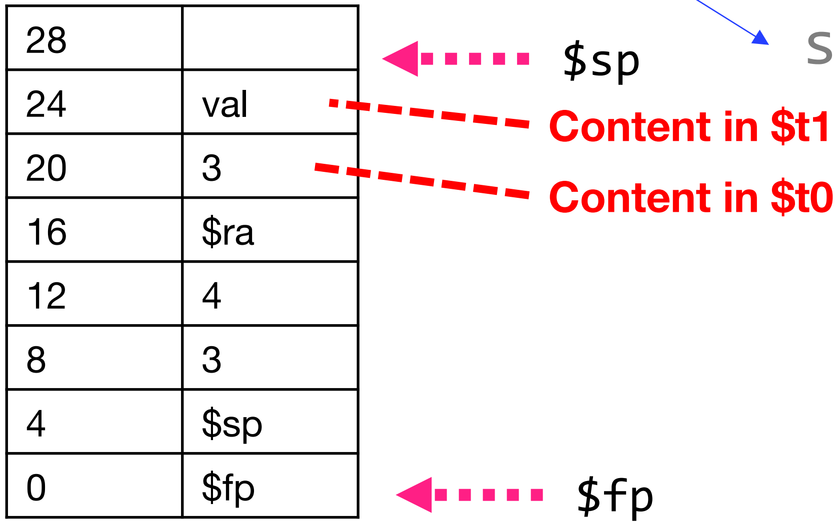
```
int f(int x,y){
    return 2*(x+y);
}
```

MIPS

```
f:    lw   $t0, 8($fp)       ;
      lw   $t1, 12($fp)      ;
      add  $t1, $t0, $t1     ;
      sll  $t1, $t1, 1       ;
      sw   $t1, 8($fp)       ;
```

Same logic as before

Store result to stack frame



28	
24	val
20	3
16	$ra
12	4
8	3
4	$sp
0	$fp

Question 4 - f

C

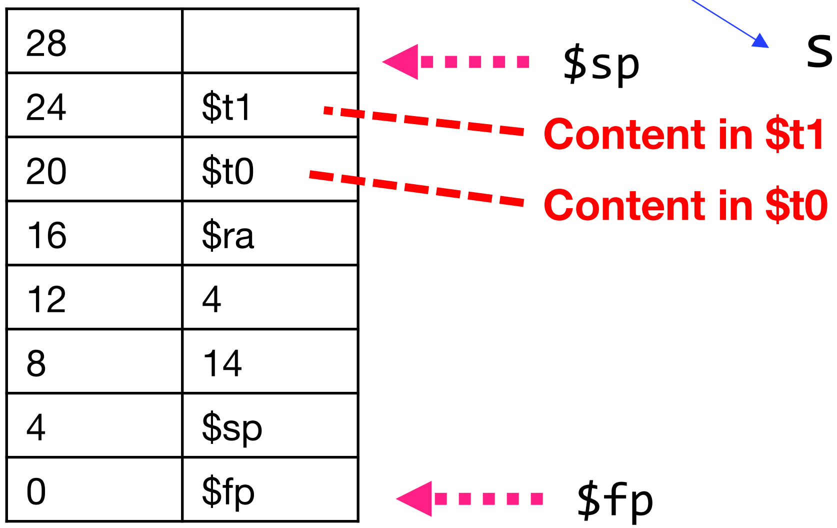
```
int f(int x,y){
        return 2*(x+y);
}
```

MIPS

```
f:    lw  $t0, 8($fp)        ;
      lw  $t1, 12($fp)       ;
      add $t1, $t0, $t1      ;
      sll $t1, $t1, 1        ;
      sw  $t1, 8($fp)        ;
```

Same logic as before

Store result to stack frame



28	
24	$t1
20	$t0
16	$ra
12	4
8	14
4	$sp
0	$fp

Question 4 - f

C

```
int f(int x,y){
     return 2*(x+y);
}
```

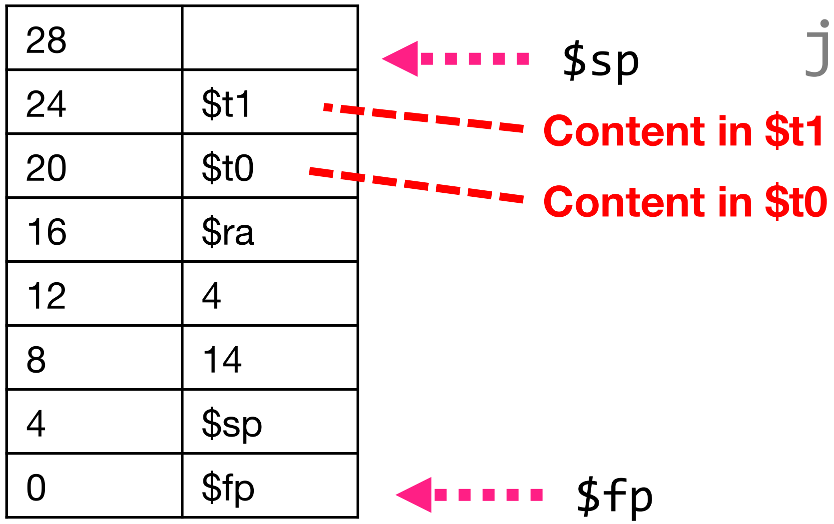
MIPS

```
f:   lw   $t0, 20($fp)          ;
     lw   $t1, 24($fp)          ;
     addi $sp, $sp, -8          ;
     lw    $ra, 16($fp)         ;
     jr   $ra                   ;
```

Restore $t0 and $t1

Deallocate space on stack

Restore $ra



28	
24	$t1
20	$t0
16	$ra
12	4
8	14
4	$sp
0	$fp

Question 4 - f

C

```
int f(int x,y){
    return 2*(x+y);
}
```

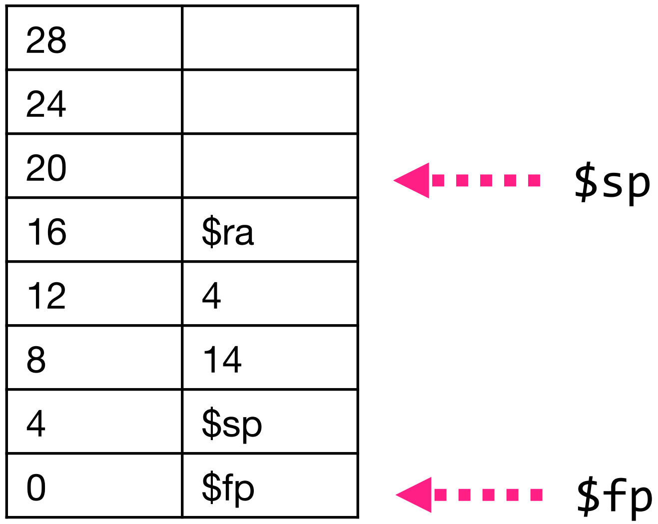
MIPS

```
f:    lw   $t0, 20($fp)        ;
      lw   $t1, 24($fp)        ;
      addi $sp, $sp, -8        ;
      lw    $ra, 16($fp)       ;
      jr   $ra                 ;
```

Restore $t0 and $t1

Deallocate space on stack

Restore $ra



28	
24	
20	
16	$ra
12	4
8	14
4	$sp
0	$fp

Question 4 - Main

C

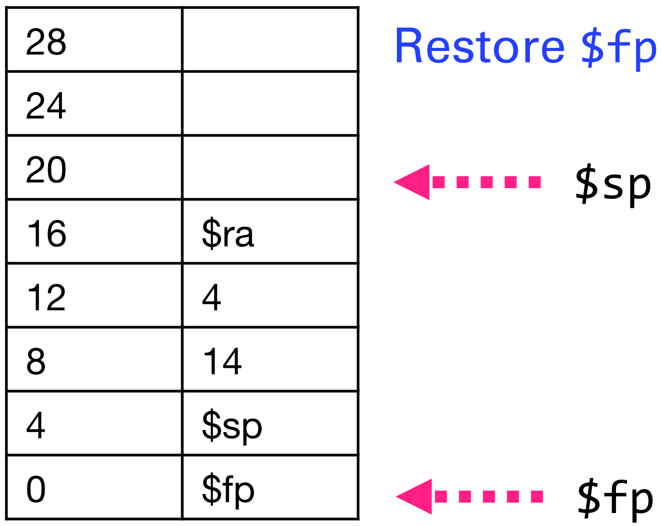
```
int a=3, b=4, y;

int main(){
     y=f(a,b)
}
```

MIPS

```
lw    $t0, 8($fp)         ;
la    $t1, y              ;
sw    $t0, 0($t1)         ;
lw    $sp, 4($fp)         ;
lw    $fp, 0($fp)         ;
li    $v0, 10             ;
syscall                   ;
```

Restore $sp

Restore $fp



28	
24	
20	
16	$ra
12	4
8	14
4	$sp
0	$fp

Question 4 - Main

C

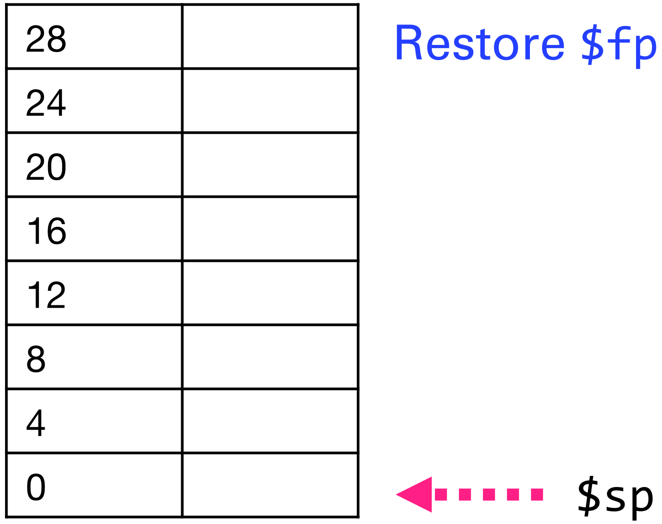
```
int a=3, b=4, y;

int main(){
     y=f(a,b)
}
```

MIPS

```
lw    $t0, 8($fp)         ;
la    $t1, y              ;
sw    $t0, 0($t1)         ;
lw    $sp, 4($fp)         ;
lw    $fp, 0($fp)         ;
li    $v0, 10             ;
syscall                   ;
```

Restore $sp

Restore $fp



28	
24	
20	
16	
12	
8	
4	
0	

Question 5

- In Question 4, the callee saved registers it intends to use onto the stack and restores them after that.
- What would happen if the callee does not do that?

Question 5

From Question 4

Caller:	1. Push `$fp` and `$sp` to stack
	2. Copy `$sp` to `$fp`
	3. Reserve sufficient space on stack for parameters by adding to `$sp`
	4. Write parameters to stack using offsets from `$fp`
	5. `jal` to callee
Callee:	1. Push `$ra` to stack
	2. Push registers we intend to use onto the stack
	3. Use `$fp` to access parameters
	4. Compute result
	5. Write result to stack
	6. Restore registers we saved from the stack
	7. Get `$ra` from the stack
	8. Return to caller by doing `jr $ra`
Caller	1. Get result from stack
	2. Restore `$sp` and `$fp`

Why?

Question 5

- In Question 4, the callee saved registers it intends to use onto the stack and restores them after that.
- What would happen if the callee does not do that?
- The callee does not know what registers the caller is using, and thus may accidentally overwrite the contents of a register that the caller was using. By saving and restoring the registers it intends to use, it prevents errors from happening.

Question 5

- In Question 4, the callee saved registers it intends to use onto the stack and restores them after that.
- Why don't we do the same thing for main?
- Main is likely to be invoked by the OS.
- The OS would have saved the registers needed during context switching.

Question 6

- Explain why, in step 7 of the callee, we retrieve `$ra` from the stack before doing `jr $ra`. Why can't we just do `jr $ra` directly?

Question 6

From Question 4

Caller:	1. Push `$fp` and `$sp` to stack
	2. Copy `$sp` to `$fp`
	3. Reserve sufficient space on stack for parameters by adding to `$sp`
	4. Write parameters to stack using offsets from `$fp`
	5. `jal` to callee
Callee:	1. Push `$ra` to stack
	2. Push registers we intend to use onto the stack
	3. Use `$fp` to access parameters
	4. Compute result
	5. Write result to stack
	6. Restore registers we saved from the stack
	7. Get `$ra` from the stack
	8. Return to caller by doing `jr $ra`
Caller	1. Get result from stack
	2. Restore `$sp` and `$fp`

Why?

Question 6

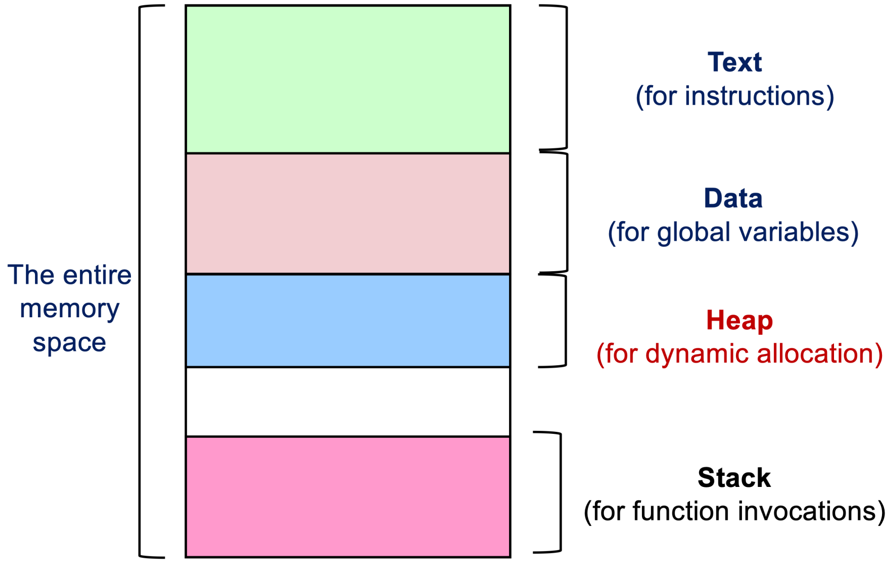
- Explain why, in step 7 of the callee, we retrieve `$ra` from the stack before doing `jr $ra`. Why can't we just do `jr $ra` directly?
- Calling another function would overwrite $ra.
- Saving and restoring $ra lets us support nesting and recursion.
- Similar to Question 3

Process Memory

Section 3

Question 7

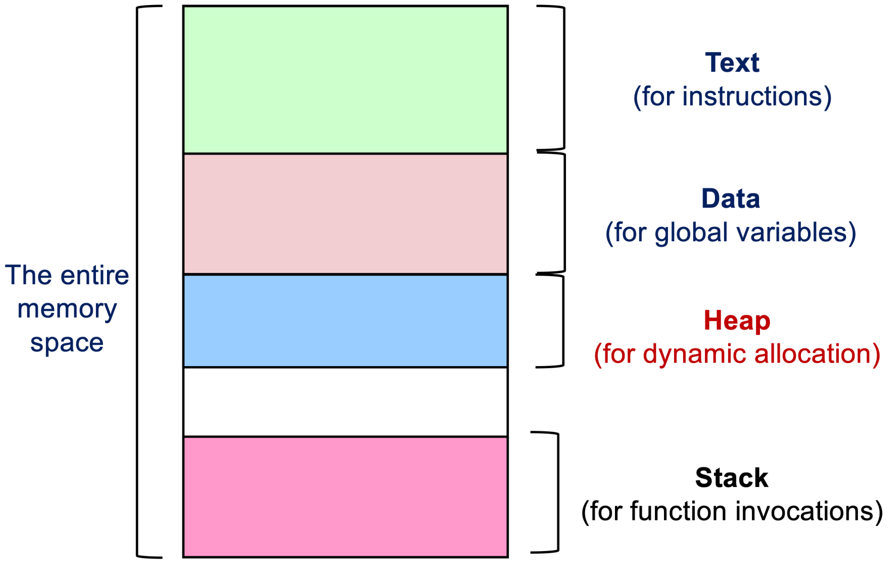
The diagram below shows the memory allocated to a typical process:



Question 7

We have the following program:

```
int fun1(int x, int y) {
  int z = x + y;
  return 2 * (z - 3);
}

int c;

int main() {
  int *a = NULL, b = 5;
  a = (int *) malloc(sizeof(int));
  *a = 3;
  c = fun1(*a, b);
}
```



Indicate in which part of a process is each of the following stored or created (Text, Data, Heap or Stack)

Question 7

- Indicate in which part of a process is each of the following stored or created.

```
int fun1(int x, int y) {
  int z = x + y;
  return 2 * (z – 3);
}

int c;

int main() {
  int *a = NULL, b = 5;
  a = (int *) malloc(sizeof(int));
  *a = 3;
  c = fun1(*a, b);
}
```

Item	**Where it is stored / created**
a	
*a	
b	
c	
x	
y	
z	
fun1's return result	
main's code	
Code for f	

Question 7

Note: Code is stored in Text

- Indicate in which part of a process is each of the following stored or created.

```
int fun1(int x, int y) {
  int z = x + y;
  return 2 * (z - 3);
}

int c;

int main() {
  int *a = NULL, b = 5;
  a = (int *) malloc(sizeof(int));
  *a = 3;
  c = fun1(*a, b);
}
```

Item	**Where it is stored / created**
a	
*a	
b	
c	
x	
y	
z	
fun1's return result	
main's code	Text
Code for f	Text

Question 7

Note: Information required for function invocation is stored in Stack memory

- Indicate in which part of a process is each of the following stored or created.

```
int fun1(int x, int y) {
  int z = x + y;
  return 2 * (z – 3);
}

int c;

int main() {
  int *a = NULL, b = 5;
  a = (int *) malloc(sizeof(int));
  *a = 3;
  c = fun1(*a, b);
}
```

Item	Where it is stored / created
a	Stack
*a	
b	Stack
c	
x	Stack
y	Stack
z	Stack
fun1's return result	Stack
main's code	Text
Code for f	Text

Question 7

Note: Global variables are stored in Data memory

- Indicate in which part of a process is each of the following stored or created.

```
int fun1(int x, int y) {
  int z = x + y;
  return 2 * (z - 3);
}

int c;

int main() {
  int *a = NULL, b = 5;
  a = (int *) malloc(sizeof(int));
  *a = 3;
  c = fun1(*a, b);
}
```

Item	Where it is stored / created
a	Stack
*a	
b	Stack
c	Data memory
x	Stack
y	Stack
z	Stack
fun1's return result	Stack
main's code	Text
Code for f	Text

Question 7

Note: Dynamically allocated memory is stored in Heap

- Indicate in which part of a process is each of the following stored or created.

```
int fun1(int x, int y) {
  int z = x + y;
  return 2 * (z - 3);
}

int c;

int main() {
  int *a = NULL, b = 5;
  a = (int *) malloc(sizeof(int));
  *a = 3;
  c = fun1(*a, b);
}
```

Item	**Where it is stored / created**
a	Stack
*a	Heap
b	Stack
c	Data memory
x	Stack
y	Stack
z	Stack
fun1's return result	Stack
main's code	Text
Code for f	Text

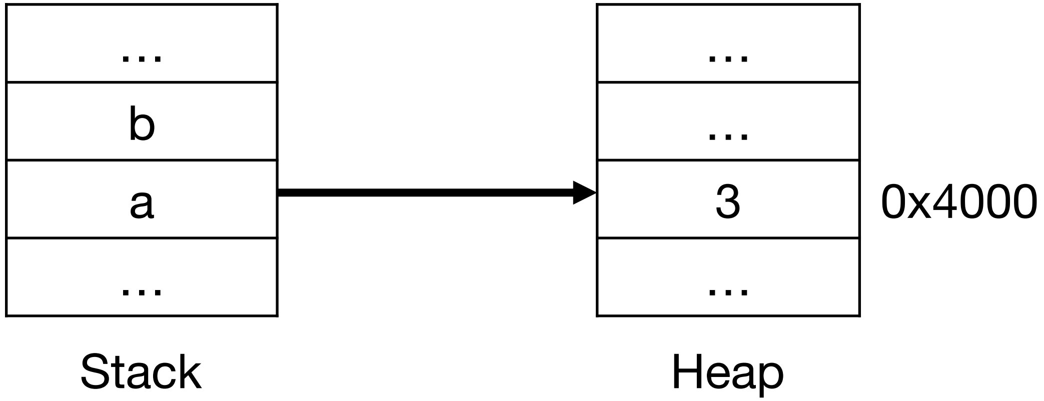
Question 7

```
int fun1(int x, int y) {
    int z = x + y;
    return 2 * (z-3);
}

int c;
```

`x, y, z:`
Information for functional invocations are stored in Stack memory

`c:`
Global variables are stored in Data

Question 7



```
int main() {
    int *a=NULL, b=5;
    a = (int*)malloc(sizeof(int));
    *a=3;
    c=fun1(*a, b);
}
```

c:
Information for functional invocations are stored in Stack memory

*a:
- Dynamically allocated memory is stored in Heap memory.
- When we dereference a, it is pointing to the heap