CS2106
Introduction to OS

Tutorial 1

Question 1

- Write the following C program in MIPS assembly.

- Pass parameters using registers instead of stack frames.

Question 1

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

MIPS
```
f:    add $t1, $a0, $a1    ;
      sll $v0, $t1, 1       ;
      jr  $ra               ;
```

x+y

2*(x+y)

Question 1

C

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

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

MIPS

```
main: la  $t0, a      ;
      lw  $a0, 0($t0) ;
      la  $t0, b      ;
      lw  $a1, 0($t0) ;
      jal f           ;
      la  $t0, y      ;
      sw  $v0, 0($t0) ;
```

Load a

Load b

Call f

Question 2

- Write the following C program in MIPS assembly.

- Use the stack to pass arguments and results instead of $a0, $a1, and $v0.

Question 2

- Pushing $r0 to stack:
  ```
  sw $r0, 0($sp)
  addi $sp, $sp, 4
  ```

- Popping to $s0 from from stack
  ```
  addi $sp, $sp, -4
  lw $s0, 0($sp)
  ```

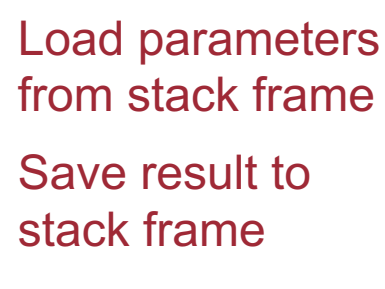# Question 2

C

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

MIPS

```
f:   lw  $t0, 0($fp)     ;
     lw  $t1, 4($fp)     ;
     add $v0, $t0, $t1   ;
     sll $v0, $v0, 1     ;
     sw  $v0, 0($fp)     ;
     jr  $ra             ;
```

Load parameters from stack frame

Save result to stack frame

Question 2

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

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

MIPS
```
main: addi $fp, $sp, 0;
      addi $sp, $sp, 8;
```

Save $sp

Reserve 2 integers on stack for stack frame
Note that each integer is 4 bytes

Question 2

C

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

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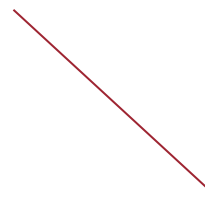
MIPS

```
main: la  $t0, a      ;
      lw  $t0, 0($t0) ;
      sw  $t0, 0($fp) ;
      la  $t0, b      ;
      lw  $t0, 0($t0) ;
      sw  $t0, 4($fp) ;
      jal f           ;
```

Write a to stack frame →

Write b to stack frame →

Question 2

C

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

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

MIPS

```
main: lw    $t1, 0($fp)  ;
      la    $t0, y        ;
      sw    $t1, 0($t0)   ;
      addi  $sp, $s0, -8   ;
      li    $v0, 10        ;
      syscall             ;
```
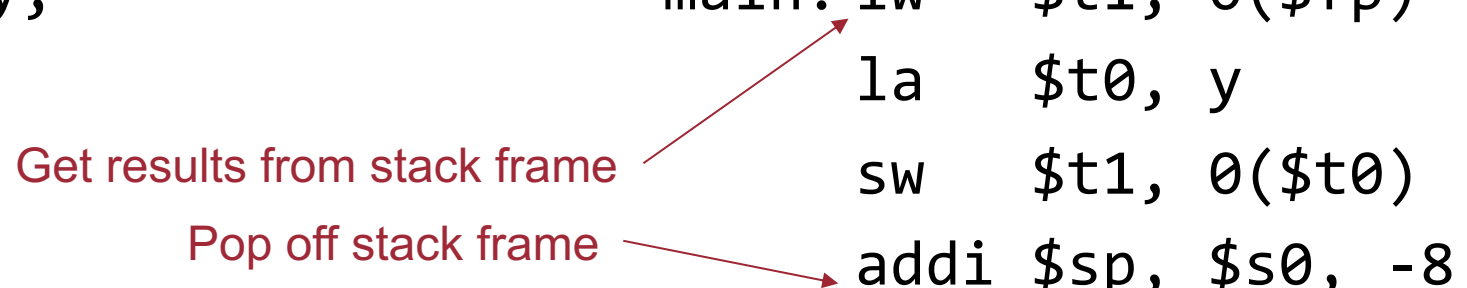
Get results from stack frame

Pop off stack frame

Question 3

- Can your approach in Questions 1 and 2 above work for recursive or even nested function calls?

- Explain why or why not.

Question 3

MIPS

```
main: jal f

f:    jal g
      jr $ra

g:    jr $ra
```

- $ra is not saved
- $ra is overwritten if f calls another function

$ra back to main is overwritten with return address of f

Where does $ra lead to now?

Question 4

- Use 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

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. Allocate enough space for local variables.
	3. Push registers we intend to use onto the stack
	4. Use $fp to access parameters
	5. Compute result
	6. Write result to stack
	7. Restore registers we saved from the stack
	8. Get $ra from the stack
	9. Return to caller by doing jr $ra
Caller	1. Get result from stack
	2. Restore $sp and $fp

Question 4

C

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

Save $ra

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

Save $t0 and $t1 which we will use

MIPS

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

Question 4

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

Question 4

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

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

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

Question 4

C

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

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

MIPS

Same logic as before (Question 2)

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

Question 4

C

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

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

MIPS

```
main: 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 →

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 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 `$ra` to stack
	2. Allocate enough space for local variables.
	3. Push registers we intend to use onto the stack
	4. Use `$fp` to access parameters
	5. Compute result
	6. Write result to stack
	7. Restore registers we saved from the stack
	8. Get `$ra` from the stack
	9. 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?

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 8 of the callee, we retrieve `$ra` from the stack before doing `jr $ra`. Why can't we just do `jr $ra` directly?

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 `$ra` to stack
	2. Allocate enough space for local variables.
	3. Push registers we intend to use onto the stack
	4. Use `$fp` to access parameters
	5. Compute result
	6. Write result to stack
	7. Restore registers we saved from the stack
	8. Get `$ra` from the stack
	9. 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 8 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.

- (See Question 3)

Question 7

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

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
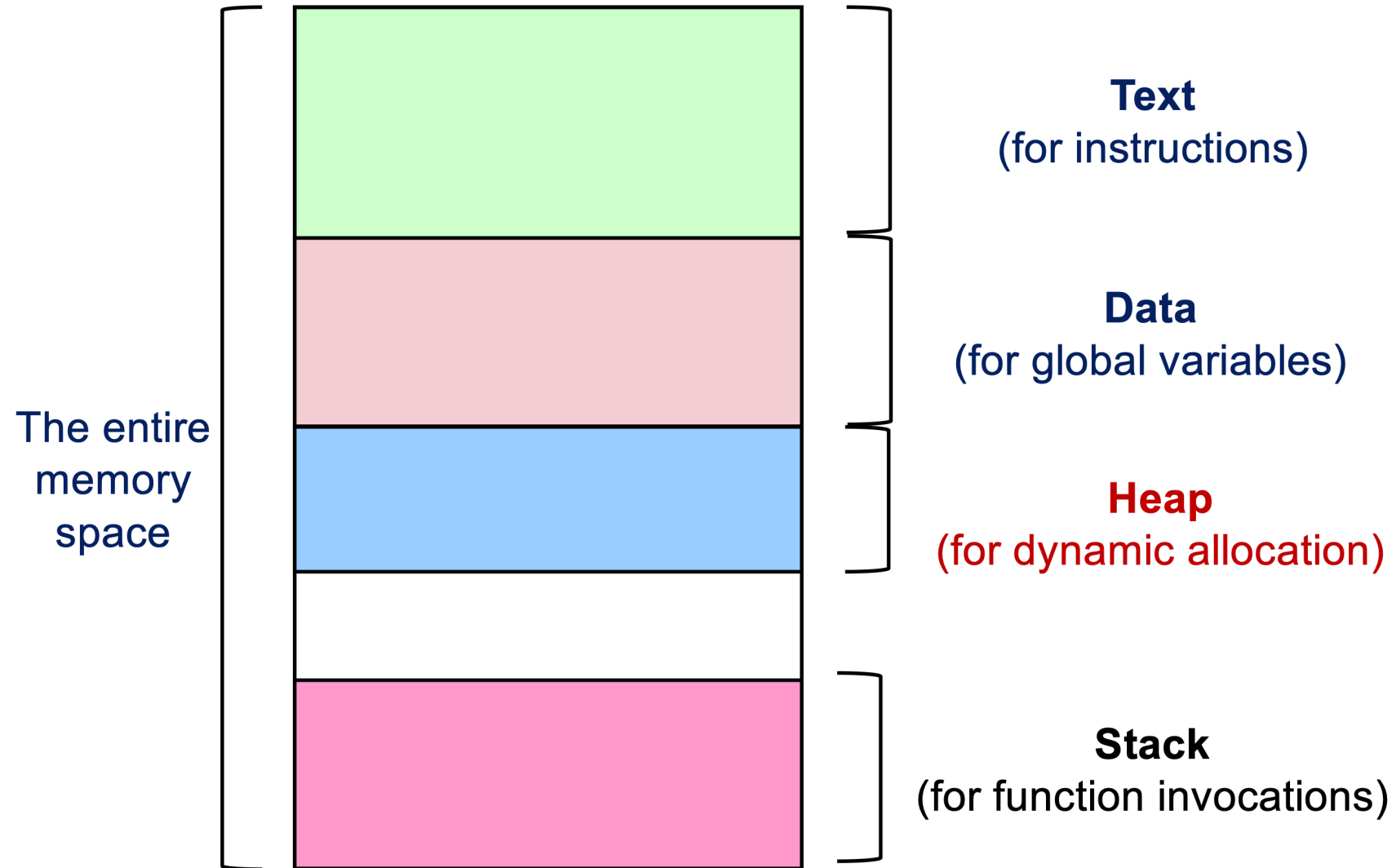
Question 7
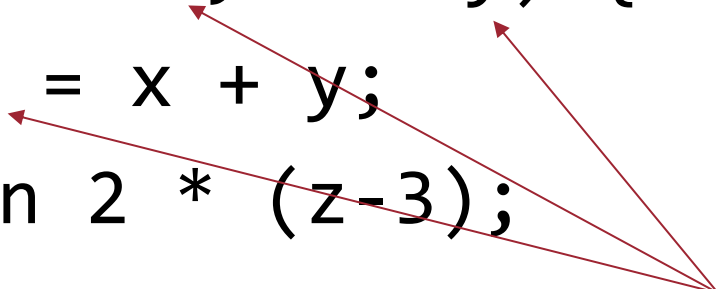
- Code is stored in Text

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

```
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

Question 7

- Information required for function invocation is stored in Stack memory

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

The End

```
li $v0, 10;
syscall
```