OS'24 Project

MILESTONE 1: **PREPROCESSING**

COMMAND PROMPT, SYSTEM CALLS, DYNAMIC ALLOCATOR & LOCKS

Agenda

- Logistics

- Part 0: Prerequisites
  - Pointers
  - LISTs in FOS

- Part 1: Play with Code!

- Part 2: System Calls

- Part 3: Dynamic Allocator

- Part 4: Locks

- Summary & Quick Guide

- How to submit?

Logistics

Startup Code:

- FOS_PROJECT_2024_template.zip

- Follow these steps to import the project folder into the eclipse

Delivery Method: GOOGLE FORM

- It's **FINAL** delivery

- **MUST** deliver the required tasks and **ENSURE** they're worked correctly

Delivery Dates:

- **SAT of Week#5 (@11:59 PM isA)**

- Upload your code EARLY as NO EXCEPTION will be accepted.

Support:

- Each team will be supported via their **MENTOR (+Lecturer)** during the published **weekly office hours** and/or **contact method**. [check this link]

Logistics

ADVICE#1: WORK AS A TEAM

Milestone 1 Functions:

1. Command Prompt   □   1 function
2. System Calls   □   2 requirements
3. Dynamic Allocator   □   5 functions + 1 test
4. Locks   □   5 functions
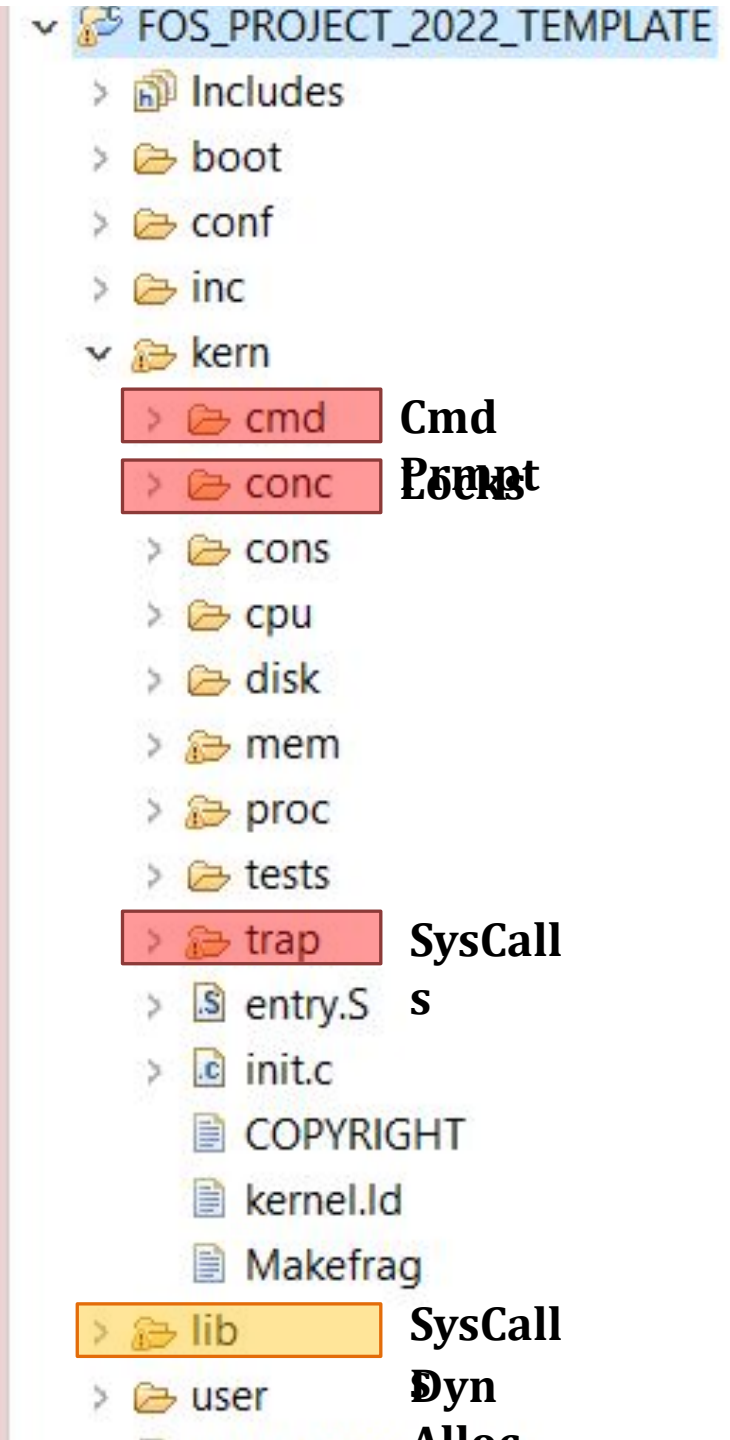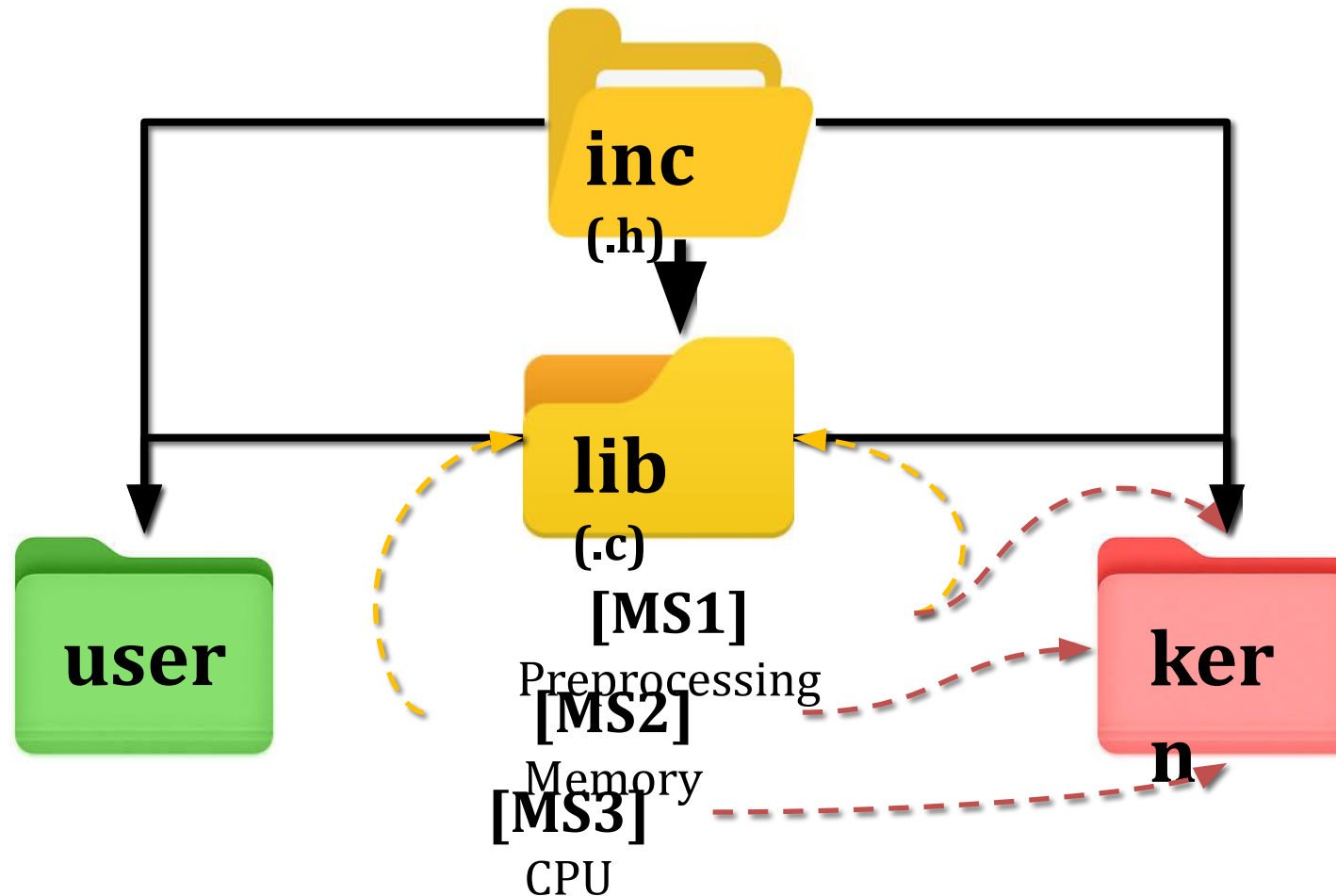
$\approx$ **2~3 Functions/member**
on **2 Weeks**

ADVICE#2: START immediately!

◦ To have the chance to ask and to understand errors in your code in whatever you want during your **mentor's support before the deadline**.

ADVICE#3: MUST read the **ppt & doc** CAREFULLY

◦ Detailed steps

◦ Helper ready made functions (*appendices*)

PROJECT BIG PICTURE

Pointers

PART0: PREREQUISITES

Memory and Pointers

a. Why we need pointers?!
b. Pointers vs. Variables...
   i.   Definition

   ii. Data type & size

   iii. Setting values

   iv. Incrementing…

   v.   Structure and accessing its members

Pointer vs Variables: Definition

Pointers	Variables
`char *ptr;`	`char x;`

Pointer vs Variables:
Data type & Size

Pointers	Variables

`char *ptr;`

Data type:
- Data type to point into it

Size of `ptr` (address):
- 4 Byte
- Size of address bus of CPU (protected ⬜ 32-bit)

`char x;`

Data type:
- Data type of variable itself

Size of x:
- 1 Byte ⬜ `sizeof(char)`

Pointer vs Variables:
Assigning value

Pointers

Variables

Assigning value:

```
char *c_ptr = 0x100;
//changes the pointed address
c_ptr = 0x50;
c_ptr = &x;


//changes  the value within
the pointed address
*c_ptr = 'A';
```

Assigning value:

```
char x = 10;
x = 50;
x = 'A';
```

Pointer vs Variables: Incrementing

Pointers	Variables

increment:

```
char *c_ptr = 0x100;
c_ptr++; // ptr=0x101


int *i_ptr = 0x100;
i_ptr++; // ptr=0x104
```

Increases by the size of its type

increment:

```
char x = 10;
x++; // x=11


int x = 10;
x++; // x=11
```

Increases by 1

Pointer vs Variables: Structure & its members

```c
struct MyStruct {
    int x, y;
    char c;
    char* c_ptr;
}
```

Pointers

Init. & Assign.:

```c
Struct MyStruct *my_struct;
my_struct->x = 5;
my_struct->c = 'A';
(*my_struct).y = 10
```

Variables

Init. & Assign.:

```c
Struct MyStruct my_struct;
my_struct.x = 5;
my_struct.c = 'A';
```

Pointer vs Variables: Structure & its members

```c
struct MyStruct {
    int x, y;
    char c;
    char* c_ptr;
};
```

Pointers

Init. & Assign.:

```c
Struct MyStruct *my_struct;
my_struct = 100;
my_struct++;   // ptr=113
increases by size of struct
```

Variables

Init. & Assign.:

```c
Struct MyStruct my_struct;
```

LISTs in FOS

PART0: PREREQUISITES

How to define LISTS in FOS?

To define a LIST that points to objects of type **struct my_struct**:

1. Create a list **head** that holds info about the list (size, head, tail).
   - **LIST_HEAD(**[LIST_TYPE_DEF], [STRUCT NAME THAT WILL POINTS TO]**);**
   - **Ex:**

   ```
   LIST_HEAD(MY_LIST_TYPE, my_struct);
   ```

2. Add **next** and **previous** pointers to the **struct**
   - **LIST_ENTRY(**[STRUCT NAME]**) prev_next_info;**
   - **Ex:**

   ```
   struct my_struct
   {
       int x, y;
       LIST_ENTRY(my_struct) prev_next_info;
   };
   ```

3. Define your list
   - **struct** [LIST_TYPE_DEF] my_list
   - **Ex:**

   ```
   struct MY_LIST_TYPE my_list;
   ```

How to use LISTS?

Set of helper ready made functions are available in <u>Appendix II here</u>

- LIST_INIT(…)

- LIST_INSERT_HEAD(…)

- LIST_SIZE(…)

- LIST_REMOVE(…)

- …

IMPORTANT: you should **pass** the list to any of these functions by **<u>reference</u>**

(i.e. Put **<u>&</u>** before the name of the list)

Agenda

- Logistics

- Part 0: Prerequisites
  - Pointers
  - LISTs in FOS

- Part 1: Play with Code!

- Part 2: System Calls

- Part 3: Dynamic Allocator

- Part 4: Locks

- Summary & Quick Guide

- How to submit?

Part1: Play with Code!

 The objective of this part is to **break the ice** with the code and start navigate in it easily.

Assess: using pointers & lists

Required:

#	Function	File(s)
1	`process_command()`	Essential declarations: **kern/cmd/command_prompt.h, kern/cmd/commands.h, kern/cmd/commands.c** Function definition TODO: **kern/cmd/command_prompt.c**

#1: process_command

Givens:

1. # of arguments per command

2. Linked list prev-next pointers

3. Initializations

```
//Structure for each command
struct Command
{
    char *name;
    char *description;
    // return -1 to force command prompt to exit
    int (*function_to_execute)(int number_of_arguments, char** arguments);
    int num_of_args ;
    Command_LIST_entry_t prev_next_info;    /* linked list links */
};
```

```
//********************************//
/* COMMANDS WITH TWO ARGUMENTS */
//********************************//
{ "wm", "writes one byte to specific physical location" ,command_writemem_k, 2},

//********************************//
/* COMMANDS WITH THREE ARGUMENTS */
//********************************//
{ "rub", "reads block of bytes " ,command_readuserblock, 3},

//**********************************//
/* COMMANDS WITH AT LEAST ONE ARGUMENT */
//**********************************//
{ "run", "runs a single user program", command_run_program, -1 },
```

#1: process_command

Givens:

3. List of found commands (if any)
   - Hold objects from **struct Command**

```
//List of found commands
struct Command_LIST foundCommands;
```

4. Command Status

```
enum{
CMD_INVALID = -3,
CMD_INV_NUM_ARGS,
CMD_MATCHED,
};
```

#1: process_command

Declaration:

```
int process_command(int number_of_arguments, char** arguments)
```

Description:

Status	Return	Content of List
1) invalid command (i.e. command chars are not exist/matched with any other command)	`CMD_INVALID`	Empty
2) command is found BUT with **invalid number of arguments**	`CMD_INV_NUM_ARGS`	Found command
3) command is **not found BUT** its chars are **subsequence-matched** with one or more commands	`CMD_MATCHED`	All matched commands
4) command is **found** with **correct** number of arguments	Index of the found command in "`commands`" array	Empty

#1: process_command

Testing:
- **[UNSEEN]** At your own…

▪**Examples:**
- **FOS>** kernel_info ⬚ should execute the kernel_info command
- **FOS>** clk ⬚ should print the commands that contains "clk" as subsequence
  - [1] nclock
  - [2] modifiedclock
  - [3] clock
- **FOS>** wm ⬚ should print invalid number of args
  - wm: invalid number of args.
- **FOS>** smm ⬚ invalid command
  - Unknown command "smm"

▪**Helper Functions:** refer to Appendix I

Agenda

- Logistics

- Part 0: Prerequisites
  - Pointers
  - LISTs in FOS

- Part 1: Play with Code!

- Part 2: System Calls

- Part 3: Dynamic Allocator

- Part 4: Locks

- Summary & Quick Guide

- How to submit?

Idea of System Calls

It's OS procedure that executes privileged instructions (e.g., I/O); (API exported by kernel)

Causes a **trap**, which

1. Switch to the kernel mode
2. Look in Interrupt Descriptor Table (IDT)
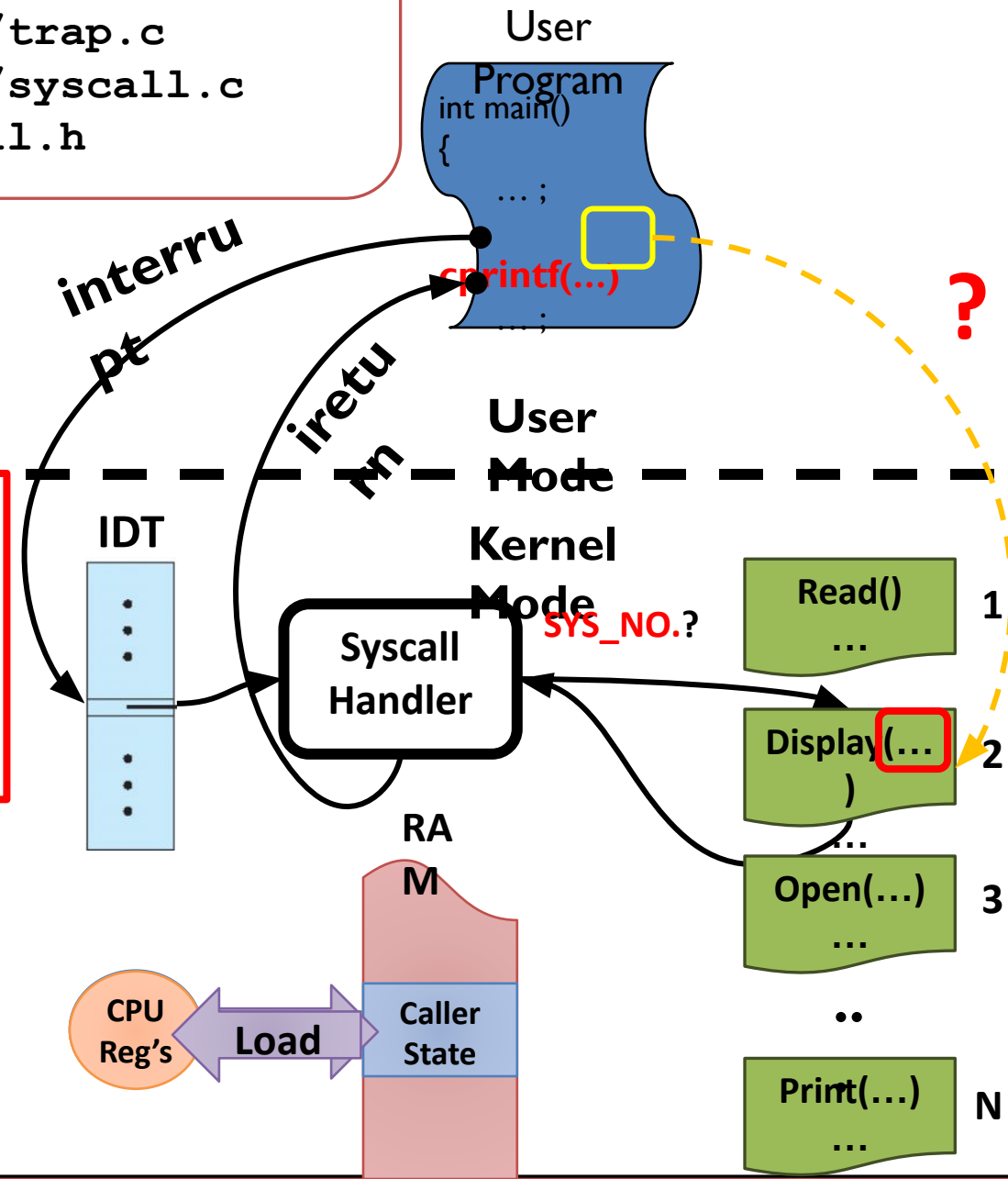3. Jumps to the **syscall handler** in the kernel.

How to pass params/return value from/to user to kernel or vice versa?

OS should verify the caller's parameters.

3. Call **associated function** that serve the given system call and pass to it the caller's **parameters**

4. After finish, **restore** caller's state (CPU Reg's)
5. use **iret** instruction to **return** to user mode.

Where in Code?
```
lib/syscall.c
kern/trap/trap.c
kern/trap/syscall.c
inc/syscall.h
```

User Program
```
int main()
{
  … ;
  printf(…)
  … ;
```

interrupt

ireturn

User Mode

?

IDT

Kernel Mode

Syscall Handler

SYS_NO.?

Read() … 1

Display(…) 2

Open(…) … 3

•••

Print(…) … N

RAM

CPU Reg's ⟷ **Load** ⟷ Caller State

#2: Syscalls

Description:
- **Implement/handle** THREE system calls.
- **Hint:** have a look on any of the existing system calls to get the idea

Syscall in User Side `(lib/syscall.c)`	Corresponding Fun in Kernel Side `(kern/trap/syscall.c)`	Used In
void* **sys_sbrk(int numOfPages)**	void* **sys_sbrk(int numOfPages)**	**MS#2**
void **sys_allocate_user_mem**(uint32 **virtual_address**, uint32 **size**)	void **sys_allocate_user_mem**(uint32 **virtual_address**, uint32 **size**)	**MS#2**
void **sys_free_user_mem**(uint32 **virtual_address**, uint32 **size**)	void **sys_free_user_mem**(uint32 **virtual_address**, uint32 **size**)	**MS#2**

Testing:
- **FOS> run tst_syscalls_1 10**

#3: Params Validation

Description:

◦ At **kernel side**: need to validate any **address (or range)** that is passed from user to kernel to ensure that:

1. **NOT null** pointers,
2. **NOT illegal** pointers (e.g. pointing to kernel memory) (i.e. outside User Heap [`USER_HEAP_START`, `USER_HEAP_MAX)`),
3. **NOT invalid** pointers (e.g. pointing to unmapped memory),

◦ Check only the **first TWO cases**. If violated, kernel should **EXIT** the **user program** by calling:

```
env_exit();
```

◦ Third case cause a "page fault" that you can handle by modifying the code for page_fault() in MS#2.

◦ This technique is normally faster because it takes advantage of the processor's MMU, so it tends to be used in real kernels (including Linux).

Testing:

◦ `FOS> run tst_syscalls_2 10`

Agenda

- Logistics

- Part 0: Prerequisites
  - Pointers
  - LISTs in FOS

- Part 1: Play with Code!

- Part 2: System Calls

- Part 3: Dynamic Allocator

- Part 4: Locks
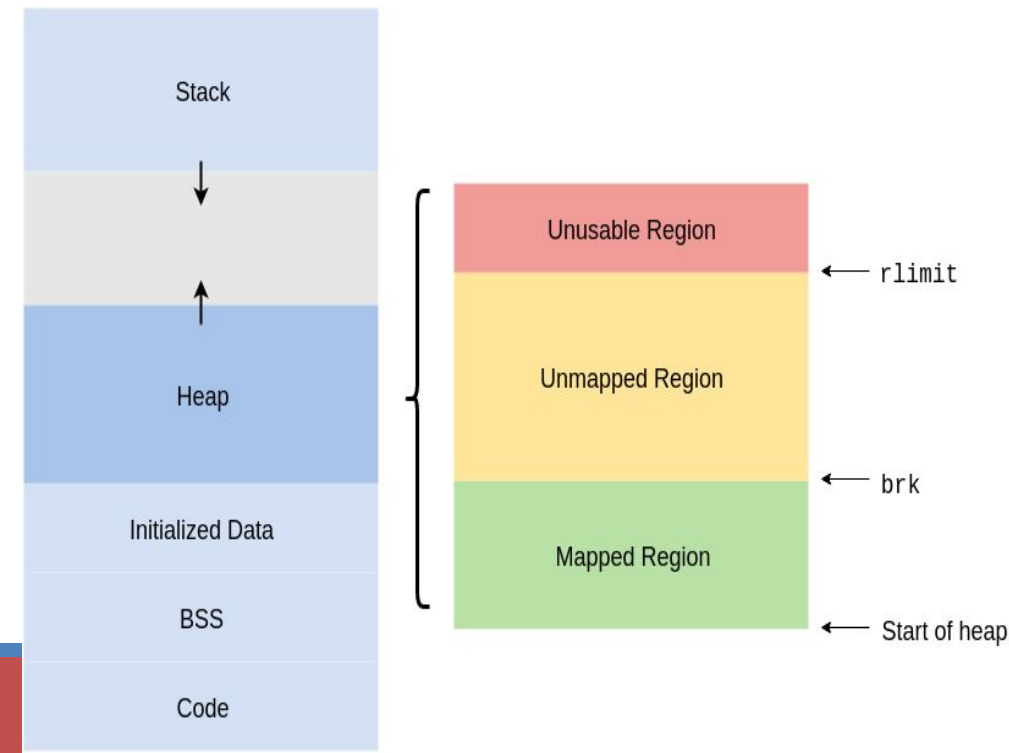
- Summary & Quick Guide

- How to submit?

Objective

Handle memory blocks **dynamically** by using **LISTS**

to **allocate or free any required space in heap by**

either OS or any user program

Idea: Heap

a space of memory, continuous in the virtual address space, with three bounds:

1. The **bottom** of the heap.

2. The top of the heap, known as the **break**. The break can be changed using sbrk.

3. The **hard limit** of the heap, which the break can't surpass.

Stack

Heap

Initialized Data

BSS

Code

Unusable Region

Unmapped Region

Mapped Region

← rlimit

← brk

← Start of heap

Idea: sbrk

```
void* sbrk(int numOfPages);
```

manipulate the **position** of the break

If it moves the segment break to **increase** the size of the heap, pages should be allocated and mapped as necessary.

More details about sbrk and its implementation will be in **MS#2** isA
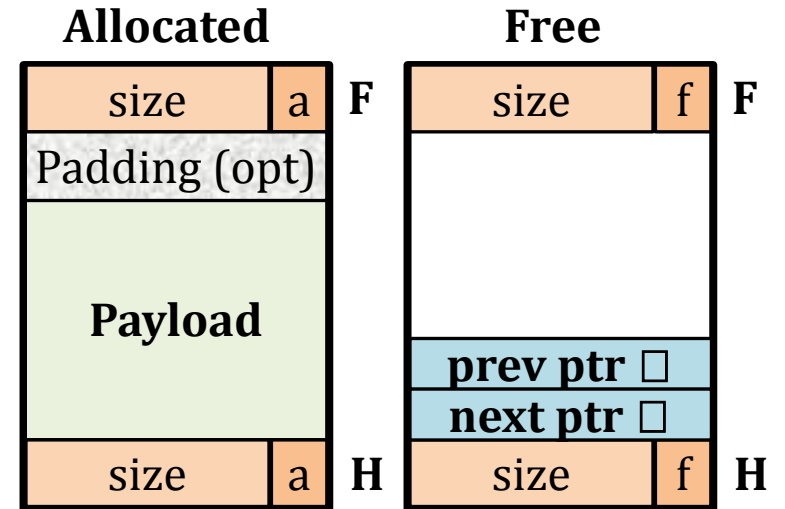
Idea: Explicit Free List

A simple memory allocator for the heap can be implemented using a

doubly linked list on the **free blocks**.

Each block has the following **metadata**:

1. **Header & Footer:** two integers, each contains

   1. **Size:** of the entire block (including the size of its meta data)

   2. **a/f:** LSB indicate whether this block is allocated (1) or free (0)

Free block has the following **extra metadata**:

1. **prev, next:** Pointers to the **adjacent FREE** blocks

Allocated		Free	
size	a **F**	size	f **F**
Padding (opt)			
Payload		**prev ptr □**	
		next ptr □	
size	a **H**	size	f **H**

- Size MUST be **even**… **WHY?**

- Min block size is **16 B**…

WHY?

Idea: Explicit Free List

The heap dynamic allocator has the following **special blocks**:

1. **BEG Block:** an integer with **size** 0 and **"a"**llocate flag of 1

2. **END Block:** an integer with **size** 0 and **"a"**llocate flag of 1

Each free block has **prev** & **next** pointers to maintain the free list

These pointers **point** to the first **location after** the **block header**

The free list need to be **sorted** by **addresses** (to facilitate the first fit allocation strategy… *see next*)
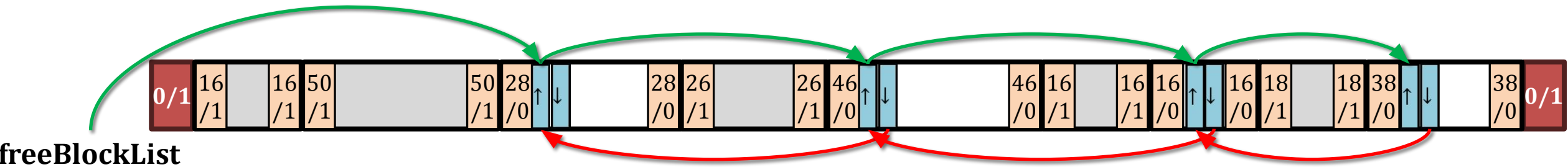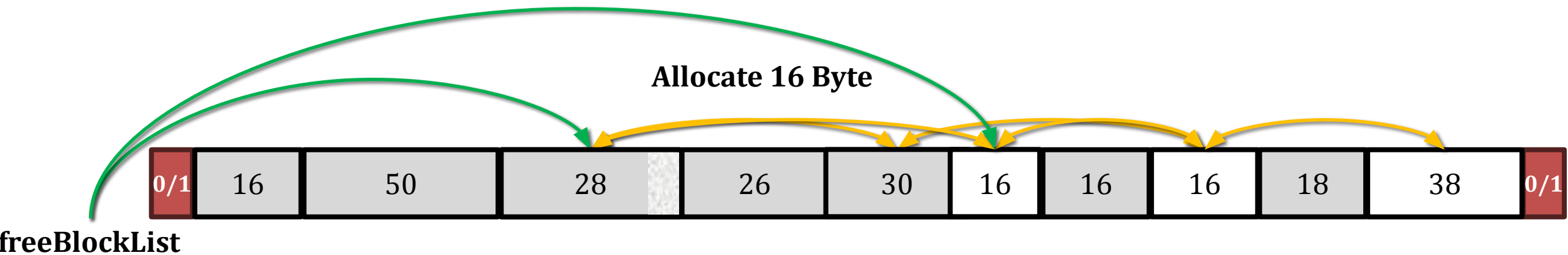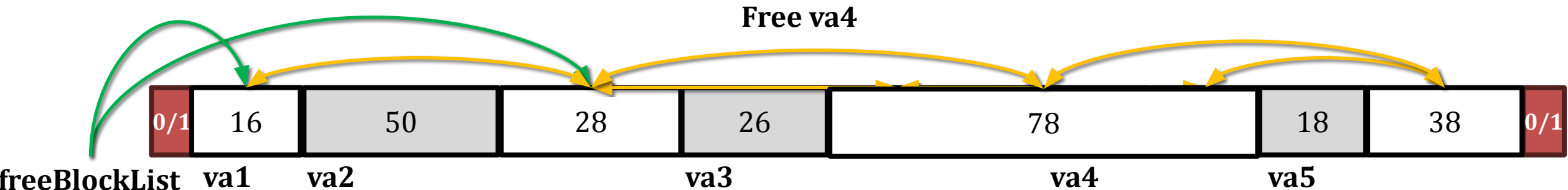

freeBlockList

Idea: Allocate Block

◦ Find the first block the fit the **required size** in addition to the **size of its metadata** (First Fit strategy)

◦ If the found block is **so large**: **split** it into **two**; one to hold the required size, other to for residual free block**.**

◦ If the found block is a **bit larger**, **BUT** not large enough for a new block

   so, **don't split!** there'll be some unused space at the end of the allocated block (**internal fragmentation**).

Allocate 16 Byte

| 0/1 | 16 | 50 | 28 | 26 | 30 | 16 | 16 | 16 | 18 | 38 | 0/1 |

freeBlockList

Idea: Free Block

◦ Free the previously allocated block at the given **virtual address**

◦ Free **doesn't release** the memory back to OS; just **indicates** that block is available for future use.

◦ Must **coalesce (i.e. merge) consecutive free blocks** upon freeing a block that is adjacent to other free block(s).

Free va4

| 0/1 | 16 | 50 | 28 | 26 | 78 | 18 | 38 | 0/1 |

freeBlockList   va1   va2   va3   va4   va5

Idea: Pros/Cons

PROS:

1. **Allocate** is a **LINEAR time** in the **number of free blocks** instead of all blocks
   - Much faster when the dynamic allocator space is almost full

2. **Coalesce** (merge) is **O(1)**

3. **Free** has a **BEST case** of **O(1)** and a **WORST case LINEAR** time in the **number of free blocks**

CONS:

1. **Extra space** is required for the list links (2 extra words for each block)
   - Increase min block size ☐ more internal fragmentation

2. After running for a while, the space becomes **highly fragmented**, creating many small free regions. This results in low utilization.

Code: Given Data Structures

1. Struct containing the **prev-next pointer** (to be used as list element)

```
struct BlockElement
{
    LIST_ENTRY(BlockElement) prev_next_info;    /* Linked List links */
};// __attribute__((packed))
```

2. List of Free Blocks that holds elements of type **BlockElement**

```
LIST_HEAD(MemBlock_LIST, BlockElement);
struct MemBlock_LIST freeBlocksList ;
```

Code: Given Functions

1. Get size of the block at the given address

   ```
   uint32 get_block_size(void* va);
   ```

2. Check the status of the block at the given address

   ```
   int8 is_free_block(void* va);
   ```

3. Print the elements of a given block list

   ```
   void print_blocks_list(struct MemBlock_LIST list);
   ```

#4: Initialize

```
void initialize_dynamic_allocator
(uint32 daStart, uint32 initSizeOfAllocatedSpace);
```

Description:

◦ Initialize the dynamic allocator starting from the given address "**daStart**" with the given allocated space "**initSizeOfAllocatedSpace**"

◦ The following items should be initialized here:

   ◦ **Free block list, BEG Block , END Block** and the **first free block**
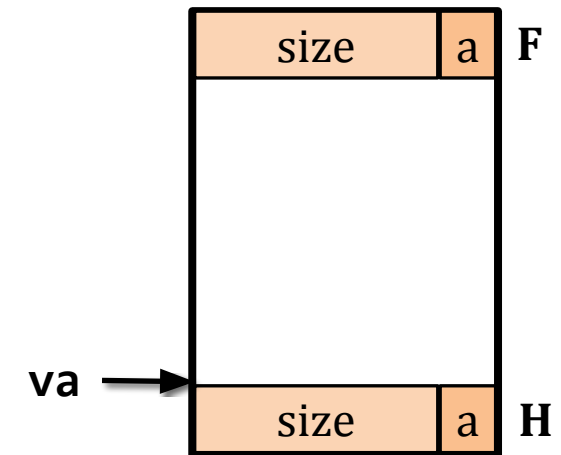
Testing: `FOS> tst dynalloc init`

#5: Set Block Data

`void set_block_data(void* va, uint32 totalSize, bool isAllocated)`

Description:

- Set the **header** & **footer** of the block at the given **va** with the given info (**totalSize & isAllocated**)

Testing: at your own...

- **Hint:** can use the given functions to check that the data is correctly set

#6: Allocate by First Fit

```
void *alloc_block_FF(uint32 required_size)
```

Description:

◦ Find the first block the fit the **required size** in addition to the **size of its metadata** (First Fit)

◦ If no sufficiently large free block is found, call **sbrk** to create more space on the heap.

◦ If the found block is **so large**: **split** it into **two**; one to hold the required size, other to for residual free block.

◦ If the found block is a **bit larger**, **BUT** not large enough for a new block

so, **don't split!** there'll be some unused space at the end of the allocated block (**internal fragmentation**).

#6: Allocate by First Fit

void *alloc_block_FF(uint32 required_size)

Return:

◦ Make sure the returned pointer points to the **beginning of the allocated space**, **not** the header.

◦ NULL if cannot allocate the new requested size (after **sbrk**)

◦ NULL if the requested size is 0.

Testing: FOS> tst dynalloc allocff

#7: Free

```
void free_block(void* va)
```

Description:

◦ Free the previously allocated block at the given address "**va**"

◦ Free **doesn't release** the memory back to OS; just **indicates** that block is available for future use.

◦ Must **coalesce (i.e. merge) consecutive free blocks** upon freeing a block that is adjacent to other free block(s).

◦ **If va is NULL, do nothing**

Testing: `FOS> tst dynalloc freeff`

#8: Reallocate by First Fit

```
void *realloc_block_FF(void* va, uint32 new_size)
```

Description:

- Should resize the allocated block at `va` to `new_size`

- If **there's sufficient** free block in front of the allocated block ☐ **resize** it in the same place

- **Else**, **relocate** it to a suitable free block (if any) using First Fit strategy

- If no sufficiently large free block is found, use `sbrk` to create more space on the heap.

- Make sure you **handle** the case where **size is less** than the **original size**.

#8: Reallocate by First Fit

```
void *realloc_block_FF(void* va, uint32 new_size)
```

Return:

◦ Address of the reallocated block.

◦ **realloc_block_FF**(va, 0) is equivalent to calling **free**(va) and returning NULL.

◦ **realloc_block_FF**(NULL, n) is equivalent to calling **alloc_FF**(n) and return the allocated address.

◦ **realloc_block_FF**(NULL, 0) is equivalent to calling **alloc_FF**(0), which should just return NULL.

Testing: FOS> tst dynalloc reallocff   [PARTIAL TEST]

#9: Complete Test of Realloc

`void **test_realloc_block_FF()**`

Description:

◦ The given test of realloc doesn't handle all possible cases

◦ It's required to **pick-up the missing cases** and **test them** to ensure that your implementation is 100% correct.

BONUS: Allocate by Best Fit

```
void *alloc_block_BF(uint32 size)
```

Description:

◦ Find the smallest block that fit the **required size** in addition to the **size of its metadata** (Best Fit strategy)

◦ If no sufficiently large free block is found, use **sbrk** to create more space on the heap.

◦ If the found block is **so large**: **split** it into **two**; one to hold the required size, other to for residual free block.

◦ If the found block is a **bit larger**, **BUT** not large enough for a new block

so, **don't split!** there'll be some unused space at the end of the allocated block (**internal fragmentation**).

BONUS: Allocate by Best Fit

void *alloc_block_BF(uint32 size)

Return:

◦ Make sure the returned pointer points to the beginning of the allocated space, not metadata header.

◦ NULL if cannot allocate the new requested size

◦ NULL if the requested size is 0.

Testing:

◦ **FOS> tst dynalloc allocbf**

◦ **FOS> tst dynalloc freebf**

Agenda

- Logistics

- Part 0: Prerequisites
  - Pointers
  - LISTs in FOS

- Part 1: Play with Code!

- Part 2: System Calls

- Part 3: Dynamic Allocator

- Part 4: Locks

- Summary & Quick Guide

- How to submit?

Objective

Implement a **Sleep Lock** inside the kernel to use it for

protecting shared resources and/or **blocking** on I/O

Locks: **Definition**

Locks provide two **atomic** operations:

- Lock.acquire() – **wait** until lock is **free**; then **mark** it as **busy**
  - After this returns, we say the calling thread *holds* the lock
- Lock.release() – **mark** lock as **free**
  - Should only be called by a thread that currently holds the lock
  - After this returns, the calling thread no longer holds the lock

Negatives of interrupt-based implementation:
- **Can't** give lock implementation to **users**
- **Doesn't** work well on **multiprocessor**

Locks: **Implementation**

Exchange Instruction

- ◦ Exchanges the contents of a register with a memory location.

- ◦ Available in Intel IA-32 (Pentium) and IA-64 (Itanium)

- ◦ The entire function is carried out **atomically**;

  - ◦ not subject to interruption (i.e. *indivisible*).

```
void exchange (int register, int memory)
{
    int temp;
    temp = memory;
    memory = register;
    register = temp;
}
```

Locks: **Implementation (SpinLock)**

Simple lock that doesn't require entry into the kernel:

```
// (Free) Can access this memory location from user space!
int lock = 0;  // Interface: acquire(&lock);
               //            release(&lock);

acquire(int *thelock) {

    int mykey = 1;
    while (xchg(thelock, mykey));    // Atomic operation!
}

release(int *thelock) {
    *thelock = 0;               // Atomic operation!
}
```

Busy-Waiting: thread consumes cycles while waiting

Locks: **Implementation (SpinLock)**

Positives

◦ Machine can receive interrupts

◦ User code can use this lock

◦ Works on a multiprocessor

◦ No System Calls at all

Usually used for
short-time **critical section**

Negatives

◦ This is very **inefficient** as thread will consume cycles waiting (**busy-waiting**)

◦ **Cache coherence** issue in multi-cores

  ◦ always read-write while waiting ☐ high traffic on cache buses to ensure data consistency

◦ **Priority Inversion**: If busy-waiting thread has higher priority than thread holding lock ⇒ no progress!

  ◦ **Solution**: **disable** the interrupt in acquire and **enable** it in release

Locks: **Implementation (SpinLock)**

```c
struct spinlock {
    uint32 locked;          // Is the lock held?
```

```c
// Acquire the lock.
// Loops (spins) until the lock is acquired.
// Holding a lock for a long time may cause
// other CPUs to waste time spinning to acquire it.
void acquire_spinlock(struct spinlock *lk)
{
    if(holding_spinlock(lk))
        panic("acquire_spinlock: lock \"%s\" is already

    pushcli(); // disable interrupts to avoid deadlock

    // The xchg is atomic.
    while(xchg(&lk->locked, 1) != 0) ;
```

```c
// Release the lock.
void release_spinlock(struct spinlock *lk)
{
    if(!holding_spinlock(lk))
        panic("release: lock %s is either not held or held

    // Release the lock, equivalent to lk->locked = 0.
    // This code can't use a C assignment, since it might
    // not be atomic. A real OS would use C atomics here.
    asm volatile("movl $0, %0" : "+m" (lk->locked) : );

    popcli();    // enable the interrupts
```

Locks: **Implementation (SleepLock)**

Idea: only busy-wait to atomically check lock value

```
SpinLock guard = 0;
int mylock = FREE;

acquire(int *thelock) {                 release(int *thelock) {
    acquire_spinlock(&guard)                 acquire_spinlock(&guard)
    while (*thelock == BUSY) {                if anyone on wait queue {
        put thread on wait queue;                 wake-up ALL blocked
        go to sleep(& release guard        processes
        // guard == 1 on wakeup!              }
    }                                         *thelock = FREE;
    *thelock = BUSY;                          release_spinlock(&guard);
    release_spinlock(&guard);           }
    }
}
```

Note: unlike previous solution, the critical section is **very short**

Note: sleep has to be sure to **reset the guard variable**

◦ Why can't we do it just before or just after the sleep?

Locks: **Implementation (SleepLock)**

What about **release guard** when going to sleep?

Release Position →
Release Position →
Release Position →

Before Putting thread on the wait queue?
◦ Release can check the queue and not wake up thread ☐ missing wake-up

After putting the thread on the wait queue
◦ Release puts the thread on the ready queue, but the thread still thinks it needs to go to sleep
◦ Misses wakeup and still holds lock (deadlock!)

Want to put it after `sleep()`. But – **how**?
◦ In Sleep: Protect the process queue(s) then release the guard. This ensures no missing wake-up even if release is called.

```
SpinLock guard = 0;
int mylock = FREE;


acquire(int *thelock) {
    acquire_spinlock(&guard)
    while (*thelock == BUSY) {
        put thread on wait queue;
        go to sleep(&)release guard
        // guard == 1 on wakeup!
    }
    *thelock = BUSY;
    release_spinlock(&guard);
}
}
```
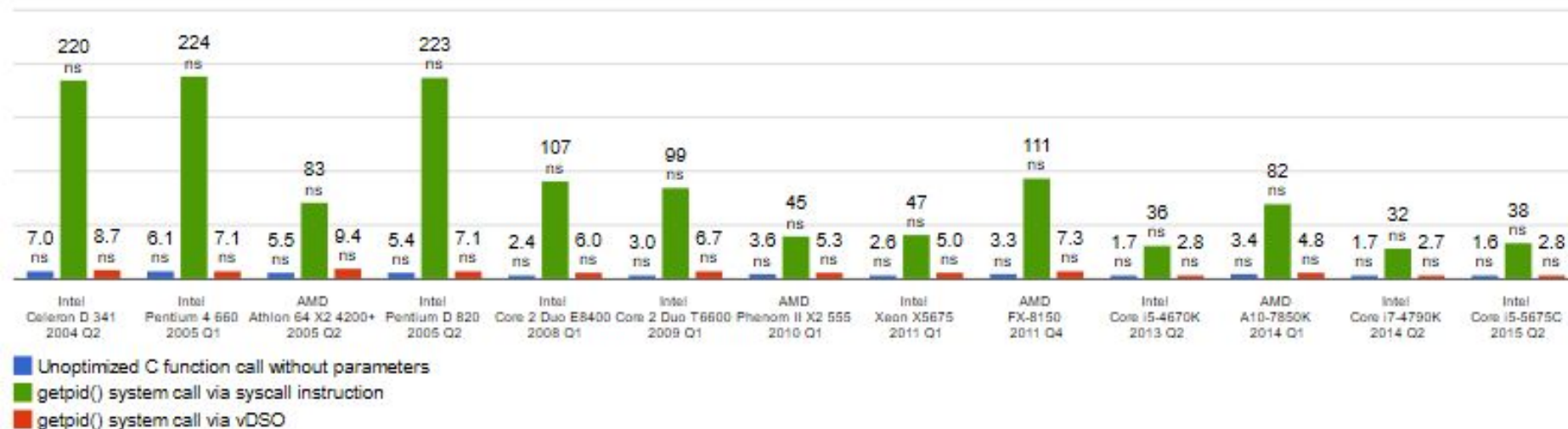
Locks: **Implementation (SleepLock)**

Positives

◦ Machine can receive interrupts

◦ User code can use this lock

◦ Works on a multiprocessor

◦ Solve Busy-Waiting

Negatives

◦ Need system call in sleep() & wakeup()

  ◦ Min System call ~ 25x cost of function call

Usually used for
long-time critical section

| | 220 ns | 224 ns | 83 ns | 223 ns | 107 ns | 99 ns | 45 ns | 47 ns | 111 ns | 36 ns | 82 ns | 32 ns | 38 ns |

| | 7.0 ns / 8.7 ns | 6.1 ns / 7.1 ns | 5.5 ns / 9.4 ns | 5.4 ns / 7.1 ns | 2.4 ns / 6.0 ns | 3.0 ns / 6.7 ns | 3.6 ns / 5.3 ns | 2.6 ns / 5.0 ns | 3.3 ns / 7.3 ns | 1.7 ns / 2.8 ns | 3.4 ns / 4.8 ns | 1.7 ns / 2.7 ns | 1.6 ns / 2.8 ns |

Intel Celeron D 341 2004 Q2 — Intel Pentium 4 660 2005 Q1 — AMD Athlon 64 X2 4200+ 2005 Q2 — Intel Pentium D 820 2005 Q2 — Intel Core 2 Duo E8400 2008 Q1 — Intel Core 2 Duo T6600 2009 Q1 — AMD Phenom II X2 555 2010 Q1 — Intel Xeon X5675 2011 Q1 — AMD FX-8150 2011 Q4 — Intel Core i5-4670K 2013 Q2 — AMD A10-7850K 2014 Q1 — Intel Core i7-4790K 2014 Q2 — Intel Core i5-5675C 2015 Q2

- ■ Unoptimized C function call without parameters
- ■ getpid() system call via syscall instruction
- ■ getpid() system call via vDSO

Code: Given Data Structures

1. Complete implementation of **SpinLock** (kern/conc/spinlock.h & .c)

2. **SpinLock** to protect ANY process queue (kern/cpu/sched.h)

ProcessQueues.qlock

3. Struct declaration of the **SleepLock** (kern/conc/sleeplock.h)

```
struct sleeplock
{
    bool locked;            // Is the lock held?
    struct spinlock lk;     // spinlock protecting this sleep lock
    struct Channel chan;    // channel to hold all blocked processes on this lock
    // For debugging:
    char name[NAMELEN];     // Name of lock.
    int pid;                // Process holding lock
};
```

4. Struct declaration of the **Channel** (kern/conc/channel.h)

```
struct Channel
{
    struct Env_Queue queue; //queue of blocked processes waiting on this channel
    char name[NAMELEN];     //channel name
};
```

5. Process Status (inc/environment_definitions.h)

```
// Values of env_status in
#define ENV_FREE        0
#define ENV_READY       1
#define ENV_RUNNING     2
#define ENV_BLOCKED     3
#define ENV_NEW         4
#define ENV_EXIT        5
#define ENV_UNKNOWN     6
```

Code: Given Functions

1. Initialize the `SleepLock` (kern/conc/sleeplock.c)

```c
void init_sleeplock(struct sleeplock *lk, char *name)
{
    init_channel(&(lk->chan), "sleep lock channel");
    init_spinlock(&(lk->lk), "lock of sleep lock");
    strcpy(lk->name, name);
    lk->locked = 0;
    lk->pid = 0;
}
```

2. Check whether the lock is held or not? (kern/conc/sleeplock.c)

```c
int holding_sleeplock(struct sleeplock *lk)
{
    int r;
    acquire_spinlock(&(lk->lk));
    r = lk->locked && (lk->pid == get_cpu_proc()->env_id);
    release_spinlock(&(lk->lk));
    return r;
}
```

Code: Given Functions

3. Get the current running process (kern/proc/user_environment..c)

```
struct Env* get_cpu_proc();
```

4. Insert a process into the ready queue (kern/cpu/sched_helpers.c)

```
void sched_insert_ready0(struct Env* p);
```

5. Queues functions: (kern/cpu/sched_helpers.c)

```
int queue_size(struct Env_Queue* queue);
void enqueue(struct Env_Queue* queue, struct Env* env);
struct Env* dequeue(struct Env_Queue* queue);
```

6. Invoke the scheduler to context switch into the next ready queue (if any) (kern/proc/user_environment..c)

```
void sched();
```

#10: Sleep on Channel

```
void sleep(struct Channel *chan, struct spinlock* lk);
```

Description:

◦ Should **block** the current running process on the given **chan** and **schedule** a next ready one

◦ It should **release** the given **lk** before being blocked so that other process(es) can use it

◦ It should **reacquire** the given **lk** again when awakened.

GENERAL NOTE: make sure to protect any process queue using the suitable lock

#11: Wake-up ONE in Channel

`void` **`wakeup_one(`**`struct` **`Channel`** `*`**`chan`**`)`

Description:

◦ Should **wake-up ONE blocked** process in the given **chan** and change it to **ready**

GENERAL NOTE: make sure to protect any process queue using the suitable lock

#12: Wake-up ALL in Channel

```
void wakeup_all(struct Channel *chan)
```

Description:

◦ Should **wake-up ALL blocked** process(es) in the given **chan** and change them to **ready**

GENERAL NOTE: make sure to protect any process queue using the suitable lock

#13: Acquire Sleep Lock

```
void acquire_sleeplock(struct sleeplock *lk)
```

Description:

◦ Should **acquire** the given sleep lock **lk**

◦ If successfully acquired, continue

◦ If failed, block the process on the corresponding channel

◦ Refer to the previously explained pseudocode

> **GENERAL NOTE:** make sure to protect any process queue using the suitable lock

#14: Release Sleep Lock

```
void release_sleeplock(struct sleeplock *lk)
```

Description:

◦ Should **release** the given sleep lock `lk` by **waken-up ALL blocked** processes on it

◦ Refer to the previously explained pseudocode

GENERAL NOTE: make sure to protect any process queue using the suitable lock

Agenda

- Logistics

- Part 0: Prerequisites
  - Pointers
  - LISTs in FOS

- Part 1: Play with Code!

- Part 2: System Calls

- Part 3: Dynamic Allocator

- Part 4: Locks

- Summary & Quick Guide

- How to submit?

Module	Function	Diff.	Testing	Files
Play w/ Code	process_command	**L2**	**UNSEEN** – Test at your own	kern/cmd/command_prompt.c
System Calls	3 system calls	**L2**	**FOS>** run tst_syscalls_1 10	lib/syscall.c, kern/trap/trap.c, kern/trap/syscall.c, inc/syscall.h
	params validation	**L1**	**FOS>** run tst_syscalls_2 10	
Dynamic Allocator	initialize	**L1**	**FOS>** tst dynalloc init	
	set_block_data	**L1**	**UNSEEN** – Test at your own	
	alloc_block_FF	**L2**	**FOS>** tst dynalloc allocff	lib/dynamic_allocator.c
	free_block	**L3**	**FOS>** tst dynalloc freeff	
	realloc_block_FF	**L3**	**FOS>** tst dynalloc reallocff (**Partial**)	
	test_realloc_block_FF	**L2**	**UNSEEN** – Test at your own	kern/tests/test_dynamic_allocator.c
	(+) alloc_block_BF	**L2**	**FOS>** tst dynalloc allocbf **FOS>** tst dynalloc freebf	lib/dynamic_allocator.c
Sleep Lock	Sleep	**L2**	TBD	kern/conc/channel.c
	Wakeup One	**L1**	TBD	kern/conc/channel.c
	Wakeup ALL	**L1**	TBD	kern/conc/channel.c
	Acquire	**L1**	TBD	kern/conc/sleeplock.c
	Release	**L1**	TBD	kern/conc/sleeplock.c

Startup Code

FOS_PROJECT_2024_Template.Zip

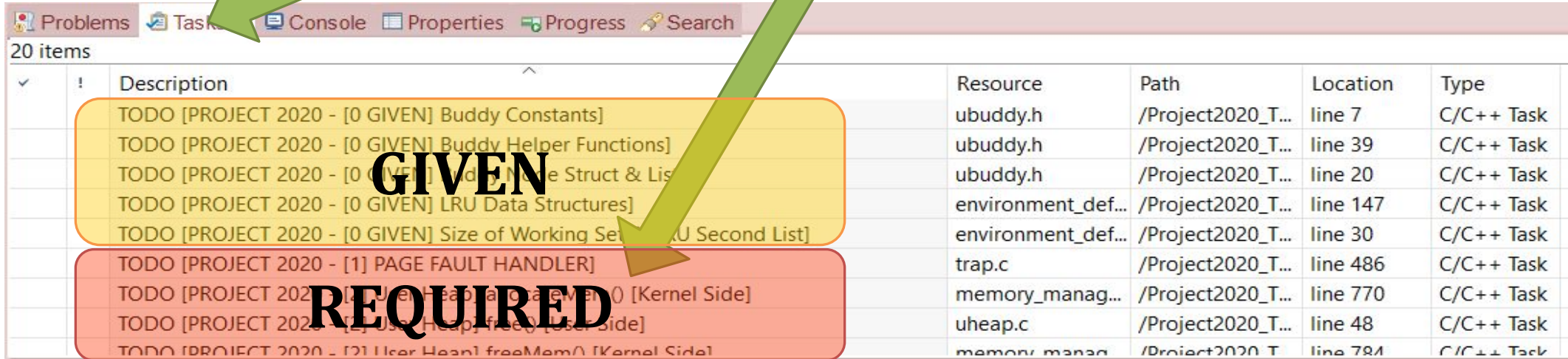-Follow [these steps](#) to import the project folder into the eclipse
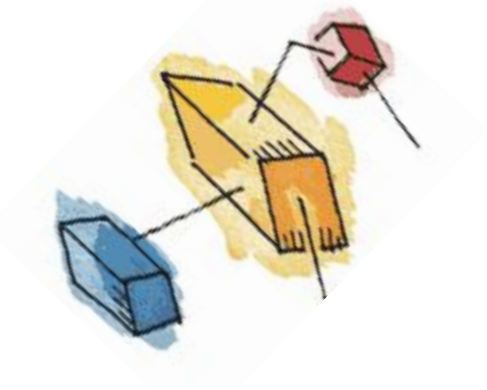
- **Helper Videos**
  - How to import a new project into eclipse?   [link]
  - Debugging:
    1. Debug via breakpoints (ECLIPSE)   [link]
    2. Debug via printing   [link: 1st minute]
    3. Locate the line causing exception via disassembly [link]

Where should I write the Code?

There're shortcut links that direct you to the function definition
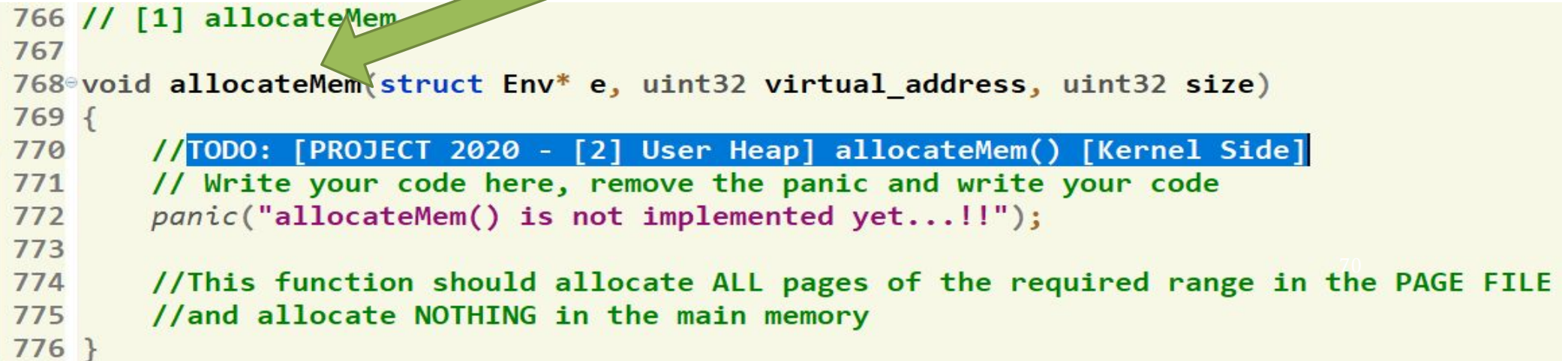
[1] Click on "Tasks" Tab   [2] Double Click on the required function

	!	Description		Resource	Path	Location	Type
		TODO [PROJECT 2020 - [0 GIVEN] Buddy Constants]		ubuddy.h	/Project2020_T...	line 7	C/C++ Task
		TODO [PROJECT 2020 - [0 GIVEN] Buddy Helper Functions]		ubuddy.h	/Project2020_T...	line 39	C/C++ Task
		TODO [PROJECT 2020 - [0 GIVEN] ...e Struct & Lis		ubuddy.h	/Project2020_T...	line 20	C/C++ Task
		TODO [PROJECT 2020 - [0 GIVEN] LRU Data Structures]		environment_def...	/Project2020_T...	line 147	C/C++ Task
		TODO [PROJECT 2020 - [0 GIVEN] Size of Working Se... LRU Second List]		environment_def...	/Project2020_T...	line 30	C/C++ Task
		TODO [PROJECT 2020 - [1] PAGE FAULT HANDLER]		trap.c	/Project2020_T...	line 486	C/C++ Task
		TODO [PROJECT 2020... [Kernel Side]		memory_manag...	/Project2020_T...	line 770	C/C++ Task
		TODO [PROJECT 2020... Heap] free() [User Side]		uheap.c	/Project2020_T...	line 48	C/C++ Task
		TODO [PROJECT 2020 - [2] User Heap] freeMem() [Kernel Side]		memory_manag...	/Project2020_T...	line 784	C/C++ Task

GIVEN

REQUIRED

20 items

[3] Function body, at which you should write the code

```
766  // [1] allocateMem
767
768  void allocateMem(struct Env* e, uint32 virtual_address, uint32 size)
769  {
770      //TODO: [PROJECT 2020 - [2] User Heap] allocateMem() [Kernel Side]
771      // Write your code here, remove the panic and write your code
772      panic("allocateMem() is not implemented yet...!!");
773
774      //This function should allocate ALL pages of the required range in the PAGE FILE
775      //and allocate NOTHING in the main memory
776  }
```

Helper Ready Made Functions

 Set of **ready-made LIST and string functions** are available to help you when writing your solution.

 Detailed description can be found in this document.

Agenda

- Logistics

- Part 0: Prerequisites
  - Pointers
  - LISTs in FOS

- Part 1: Play with Code!

- Part 2: System Calls

- Part 3: Dynamic Allocator

- Part 4: Locks

- Summary & Quick Guide

- How to submit?

Submission Rules

First ensure the following that (READ CAREFULLY):

- You tested each function in a **FRESH RUN** and a congratulations message have been appeared.

- **NO CODE with errors WILL BE CORRECTED**. So, CLEAN & RUN your project several times before your submission.

- You submitted **BEFORE** the deadline by several hours to **AVOID** any internet problems.

- **DEADLINE: SAT of Week #5 (26/10 @11:59 PM)**

- **NO DELAYED submissions WILL BE ACCEPTED**.

- **ONLY ONE person** from the team shall submit the code.

- The **TEAM # MUST BE CORRECT**.

- **DON'T take the FORM LINK FROM ANYONE**. OPEN the form from its LINK ONLY. **Otherwise, your submission is AUTOMATICALLY CANCELLED by GOOGLE**.

- You **MUST RECEIVE A MAIL FROM GOOGLE with your submission after clicking submit**. If **nothing received, re-submit again to consider your submission**.

Submission Steps

STEPS to SUBMIT:

☐   Step 1: Clean & run your code the last time to ensure that there are any errors.

☐   Step 2: Create a new folder and name it by your team number **ONLY**. Example `1` or `95`. [**ANY extra chars will lead to 0**].

☐   Step 3: **DELETE** the "**obj**" folder from the "FOS_PROJECT_2024_Template"

☐   Step 4: PASTE the "FOS_PROJECT_2024_Template" in the folder created in step #2.

☐   Step 5: Zip the created new folder. Its name shall be like **[num of your team.zip]**. [**ANY extra chars will lead to ZERO**].

☐   Step 6: Open the form from **HERE.**

☐   Step 7: Fill your team's info .. Any wrong information will cancel your submission, revise them well.

☐   Step 8: Upload the zipped folder in step 5 to the form in its field.

☐   Step 9: MUST RECEIVE A MAIL from GOOGLE with your submission, otherwise re-submit again.

Thank you for your care…
Enjoy **making** your **own FOS** ☺