OS'24 Project

MILESTONE 3: **CPU**

FAULT HANDLER II, SEMAPHORES & SCHEDULER

Agenda

- Logistics
- Part 0: Code Updates
- Part 1: Fault Handler II (Replacement)
- Part 2: User-Level Semaphores
- Part 3: Priority RR Scheduler
- OVERALL Testing
- BONUSES
- Summary & Quick Guide
- How to submit?

Logistics

Dependency:

- MS1: dynamic allocator (**`alloc_block_FF, free_block`**)
- MS2: kernel heap (**`kmalloc, kfree, k_virtial_address`** & **`k_physical_address`**)
- MS2: placement
- MS2: shared memory (**`smalloc, sget`**)

Delivery Method: GOOGLE FORM

- It's **FINAL** delivery
- **MUST** deliver the required tasks and **ENSURE** they're worked correctly

Delivery Dates:

- **SAT of Week #13 (21/12 @11:59 PM)**
- Upload your code EARLY as NO EXCEPTION will be accepted.

Support:

- The support for teams will be through their **MENTORS ONLY (+Lecturer)** during via:
  1. MAIN METHOD: weekly office hours.
  2. SECONDARY METHOD [OPTIONAL]: other contact method [**MUST declare your Team# first**]

Logistics

ADVICE#1: WORK AS A TEAM

Milestone 3: CPU

1. Fault Handler II: 1 functions
2. Semaphores: 4 functions
3. CPU Scheduling: 4 functions
4. OVERALL Testing

} **INDEPENDENT Modules**

≈ **~1-2 Functions/member + OVERALL Testing** on **3 Weeks**

L1 □ 4 FUNCTIONS - L2 □ 4 FUNCTIONS - L3 □ 1 FUNCTION

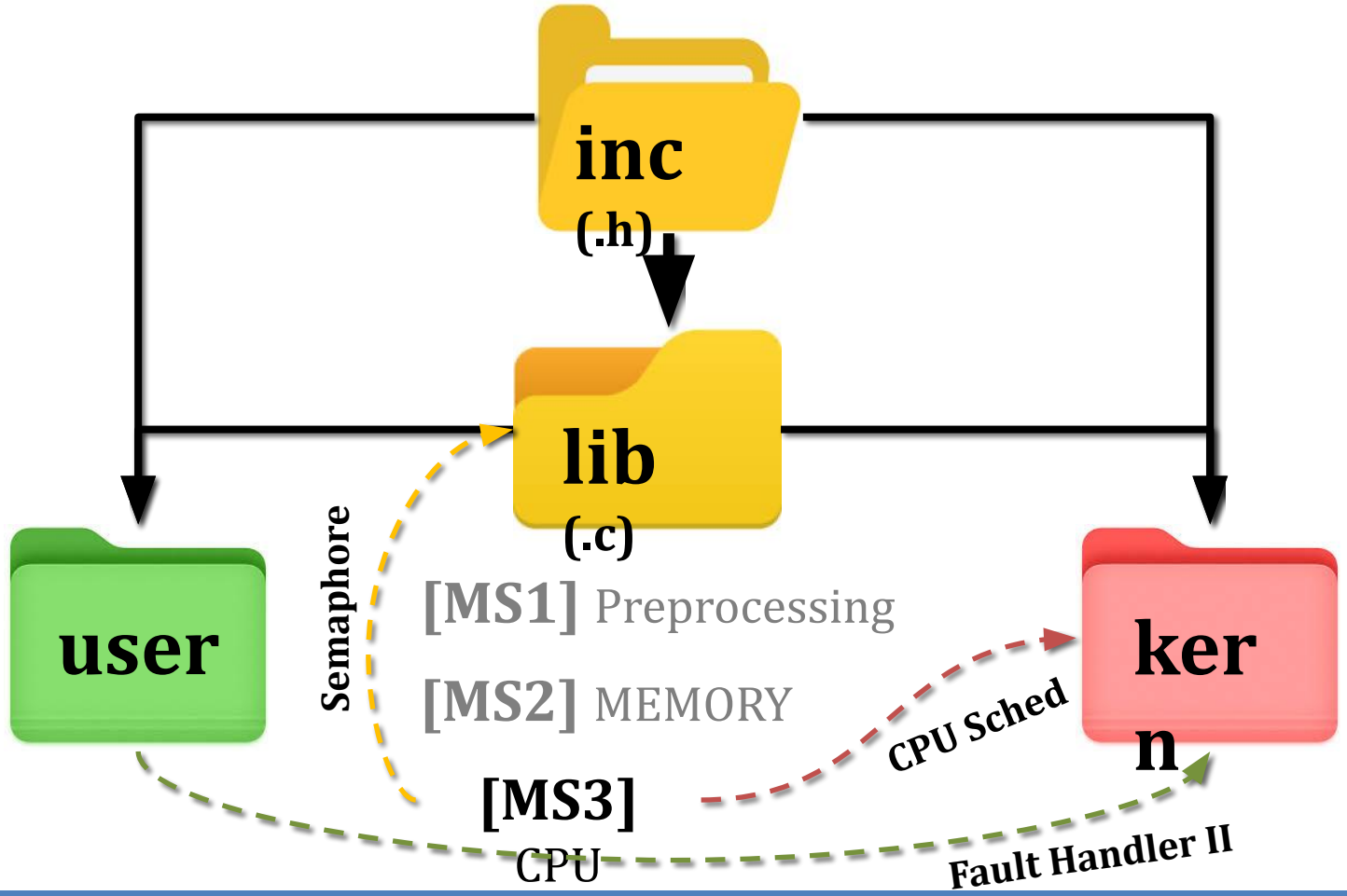
ADVICE#2: START immediately!

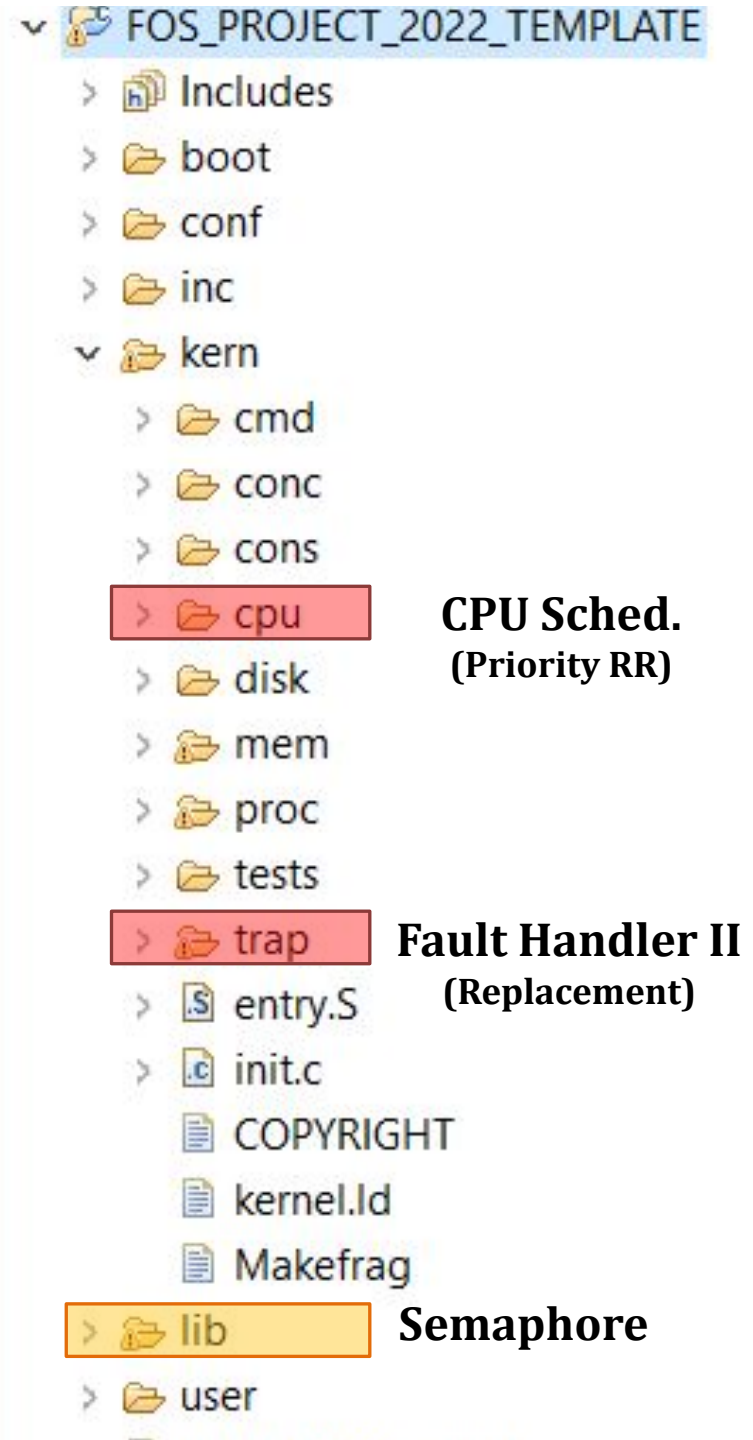
- To have the chance to ask and to understand errors in your code in whatever you want during your **mentor's support before the deadline**.

ADVICE#3: MUST read the ppt & doc CAREFULLY

- Detailed steps
- Helper ready made functions (*appendices*)

PROJECT BIG PICTURE





Code Updates

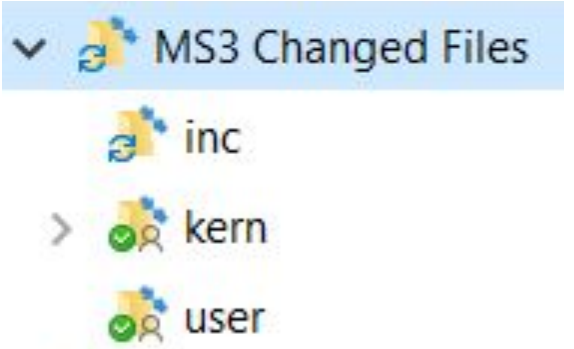
PART0: PREREQUISITES

New Files

1. **SELECT ALL** in the given "**Changed files**" folder,

2. **COPY** & **PASTE** (**REPLACE ALL**) in **FOS_CODES/FOS_PROJECT_2024_TEMPLATE/**

NOTE: If any of these files are already edited by you in MS1, make sure to apply the edits in the new files



Code Modification

Inside **`kern/trap/fault_handler.h`**, apply the following changes:

- **Modify** the data type of **`page_WS_max_sweeps`** variable:

```
31 /*2021*/ uint32 page_WS_max_sweeps;
31 /*2021*/ int page_WS_max_sweeps;
```

Code Modification

Inside **`inc/memlayout.h`**, apply the following changes:

- **ADD** the following line at any location **after line#155**:

```
#define PGFLTEMP (UTEMP - PAGE_SIZE)
```

Given Codes

APPENDICES:

1. ENTRY MANIPULATION in TABLES and DIRECTORY
2. PAGE FILE HELPER FUNCTIONS
3. WORKING SET STRUCTURE & HELPER FUNCTIONS
4. **SCHEDULER** STRUCTURE & HELPER FUNCTIONS
5. MEMORY MANAGEMENT FUNCTIONS
6. READY-MADE COMMANDS

CAUTION

During your solution, any SHARED data need to be PROTECTED by critical section via LOCKS

REMEMBER: Ensure CORRECTNESS by DESIGN

Protection Test (after MS2)

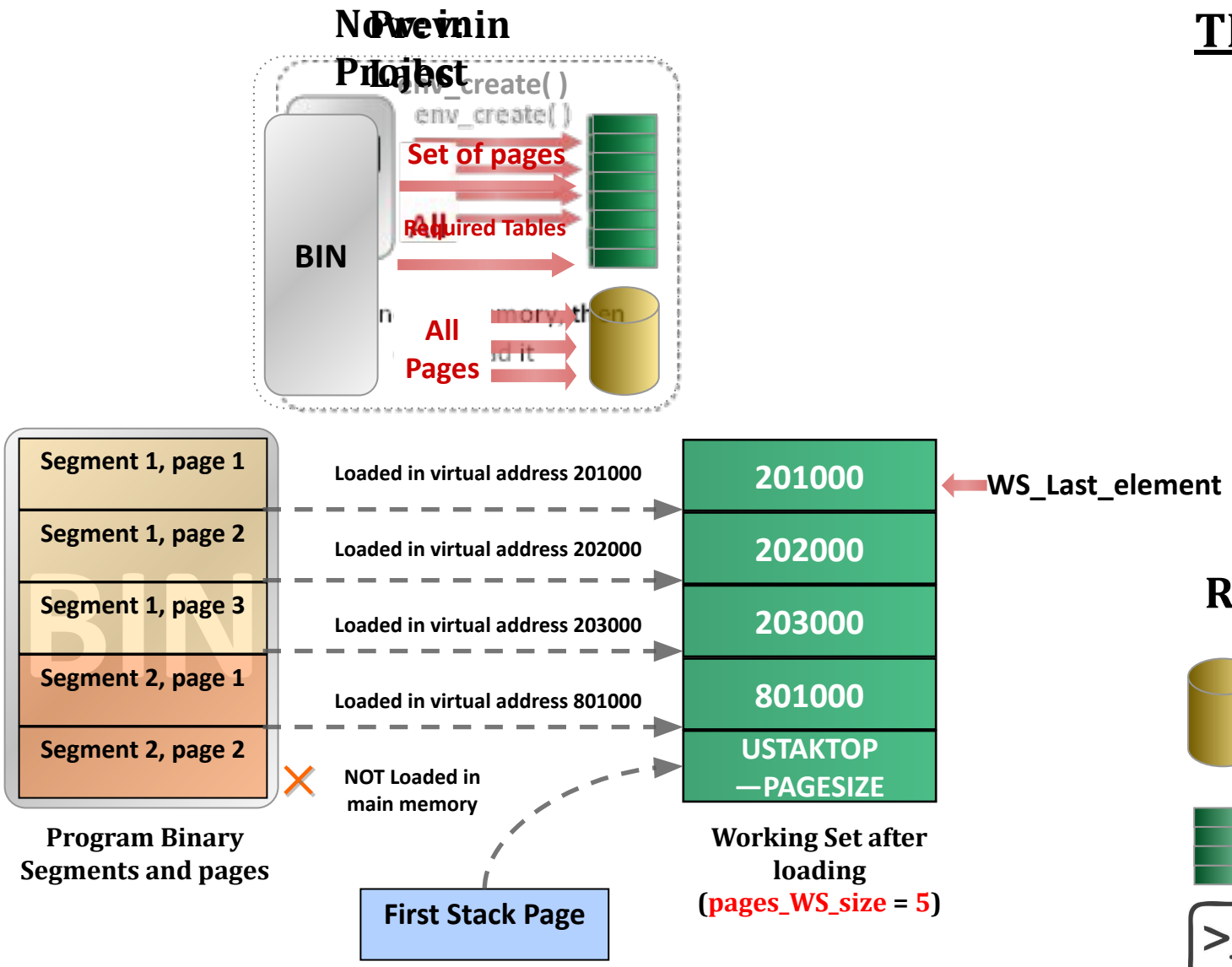
- **FOS>** `run tst_protection 5000`

- Successful message should occur…

Agenda

- Logistics
- Part 0: Code Updates
- **Part 1: Fault Handler II (Replacement)**
- Part 2: User-Level Semaphores
- Part 3: Priority RR Scheduler
- OVERALL Testing
- BONUSES
- Summary & Quick Guide
- How to submit?

Load Program [env_create] [already DONE]



THREE kernel dynamic allocations:

1. **`create_page_table()`:** create new page table and link it to directory.
2. **`create_user_directory()`:** create new user directory.
3. **`create_user_kern_stack(...)`:** create new user kernel stack.

Refer to APPENDICES for:

- Page File Helper Functions
- Working Set Structure & Helper Functions
- Ready-Made Commands

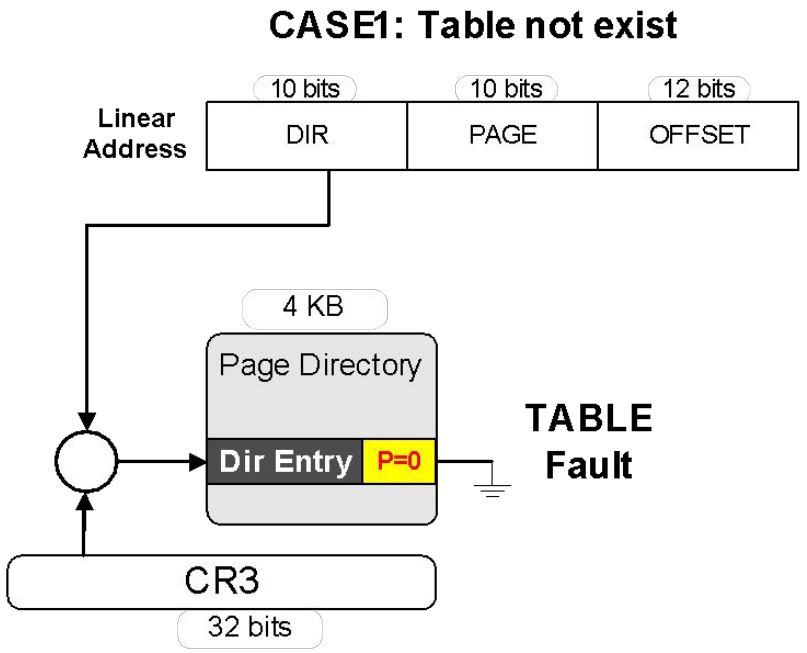
Fault Handler II: **Replacement**

The main functions required by MS3 to handle "**Page Fault**" are:

#	Function	File
1	`page_fault_handler`	**Functions definitions TO DO in: kern/trap/fault_handler.c**

Fault Handler II: **Introduction**

- **Fault:** is an exception thrown by the processor (MMU) to indicate that:
  - **A page table** is not exist in the main memory (i.e. new table). (see the following figure) **OR**
  - **A page** can't be accessed due to either it's not present in the main memory



Replacement: N^{th} Chance Clock – Idea

N^{th} chance algorithm: Give page N chances

- OS keeps counter per page: # sweeps
- Start searching from the WS element after the last placed one (i.e. FIFO)
- On page fault, OS checks **use bit**:
  - 0⇒increment counter; if count=N, replace page
  - 1⇒clear use and also clear counter (used in last sweep)
- Means that clock hand has to sweep by N times without page being used before page is replaced
- What about modified pages?
  - Takes extra overhead to replace it, so give it an **extra chance** before replacing?
  - Common approach: * **Clean** pages: use **N** * **Modified** pages: use **N+1**

Replacement: $\mathbf{N^{th}}$ Chance Clock – Idea

How do we pick N?

- Why pick large N? Better approximation to LRU
  - If N ~ 1K, really good approximation
- Why pick small N? More efficient
  - Otherwise might have to look a long way to find free page

IMP. NOTE: you should maintain `page_last_WS_element` & correct FIFO order of the List in rest of code (`page_fault_handler`, `free_user_mem` …)

```
//================
/*WORKING SET*/
//================
//page working set management
struct WS_List page_WS_list; FIFO, CLK & Nth CLK   //List of WS elements
struct WorkingSetElement* page_last_WS_element;    //ptr to last inserted WS element
unsigned int page_WS_max_size;                     //Max allowed size of WS
```

Each Element

Proc Limit

- Each process has a **working set LIST** that is initialized in **`env_create`**()
- Its **max size** is set in "**`page_WS_max_size`**" during the **`env_create`**()
- "**`page_last_WS_element`**" will point to either:
  - the **next location** in the WS after the last set one If **list is full**.
  - **Null** if the list is **NOT full**.

inc/environment_definit

```
struct WorkingSetElement {
    unsigned int virtual_address;
    unsigned int time_stamp ;
    unsigned int sweeps_counter;
    LIST_ENTRY(WorkingSetElement) prev_next_info;
```

- This list hold pointers to **`struct`** containing info about the currently loaded pages in memory.
- Each struct holds two important values about each page:
  1. User virtual address of the page
  2. Number of sweeps (**+ve**: NORMAL Ver, **-ve**: MODIFIED Ver.)

Working Set: **Structure**

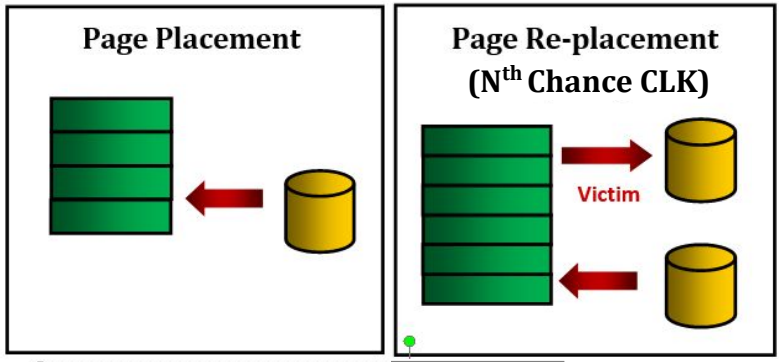
kern/trap/fault_hand

```
31 /*2021*/ int page_WS_max_sweeps;
```

- MAX number of sweeps for the N^{th} chance clock replacement algorithm:
  - If +ve: NORMAL algorithm.
  - If -ve: MODIFIED algorithm.

Refer to APPENDICES for: Page File Helper Functions

Working Set Structure & Helper Functions



#1: Nth CLK Re/placement

```
page_fault_handler(struct Env * curenv, uint32 fault_va)
```

```
if(isPageReplacmentAlgorithmNchanceCLOCK())
{
    if the size of page_WS_list < its max size, then do
    {
        Scenario 1: Placement

        //[DONE in MS2]: [PROJECT'24.MS2] PAGE FAULT HANDLER - Placement

    }
    else
    {
        Scenario 2: Replacement

        //TODO: [PROJECT'24.MS3 - #1] [1] PAGE FAULT HANDLER - Nth CLK Replac.

    }
}
```

Page Fault Handler: **Switching…**

- To switch the replacement from the FOS prompt:
  - **`FOS> nclock <N> 1`** → switch the replacement to Nth Chance Clock (**NORMAL** Version)
  - **`FOS> nclock <N> 2`** → switch the replacement to Nth Chance Clock (**MODIFIED** Version)
  - To switch the replacement from the code:
    - Inside the **`FOS_initialize()`** in "**`kern/init.c`**":
      - **`setPageReplacmentAlgorithmNchanceCLOCK(int N);`** → set Nth Chance Clock replacement
        - If **+ve**: **NORMAL** algorithm.
        - If **-ve**: **MODIFIED** algorithm.

"Congratulations!! test [TEST NAME] completed successfully."
To ensure the test success, a congratulations message like this **MUST appear without any ERROR messages or PANICs**.

Page Fault Handler: SEEN Testing

- Test each function in MS3 independently in a **FRESH SEPARATE RUN**
- The time limit of each individual test: **max of 15 sec / each**
- All below tests are used to validate the **page_fault_handler with Nth Chance Clock** function

#	Test Functionality	Test
1	***tst_page_replacement_alloc.c (tpr1):*** tests allocation in memory and page file after page replacement.	**FOS>** run tpr1 11
2	***tst_page_replacement_stack.c (tpr2):*** tests page replacement of **stack** (i.e. new pages) (creating, modifying and reading them)	**FOS>** run tpr2 6
3	***tst_page_replacement_nthclock_1.c (tnclock1):*** tests page replacement by **Nth Chance CLK** algorithm [**NORMAL** version] (it checks working set before & after replacements)	**FOS>** nclock 5 1 **FOS>** run tnclock1 11
4	***tst_page_replacement_nthclock_2.c (tnclock2):*** tests page replacement by **Nth Chance CLK** algorithm [**MODIFIED** version] (it checks working set before & after replacements)	**FOS>** nclock 5 2 **FOS>** run tnclock2 11

Page Fault Handler: **UNSEEN** Testing

#	Test Functionality
1	tests page replacement by **Nth Chance CLK** algorithm (check **maintaining of** `page_last_WS_element` **& correct FIFO order** of the List after `free_user_mem`)
2	tests page replacement by **Nth Chance CLK** algorithm (check **effect of large N vs normal clock (N = 1)**)
3	tests page replacement by **Nth Chance CLK** algorithm (check **# disk writes in NORMAL vs. MODIFIED**)

Agenda

- Logistics
- Part 0: Code Updates
- Part 1: Fault Handler II (Replacement)
- **Part 2: User-Level Semaphores**
- Part 3: Priority RR Scheduler
- OVERALL Testing
- BONUSES
- Summary & Quick Guide
- How to submit?

User-Level Semaphores

The main functions required by MS3 to handle "**Semaphores**" are:

#	Function	File
1	`create_semaphore()`	All essential declarations in: **inc/semaphore.h** **Functions definitions TO DO in: lib/semaphore.c**
2	`get_semaphore()`	
3	`wait_semaphore()`	
4	`signal_semaphore()`	

User-Level Semaphores – **Overview**

Operations:

1. **Initialize** by **non-negative** value
2. **semWait:**
   1. decrements the value
   2. If value < 0 □ process is blocked
   3. Else □ process continues execution
3. **semSignal:**
   1. increments the value.
   2. If value ≤ 0 □ unblock a process (make it Ready)

3 Operations are **atomic**

User-Level Semaphores – **Overview**

```
void semWait(semaphore s)
{  int keyw = 1;
   do xchg(&keyw, &s.lock) while (keyw != 0);
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue */;
      /* block this process (must also set s.lock = 0) */;
   }
   s.lock = 0;     }
```

acquire()

release()

```
void semSignal(semaphore s)
{  int keys = 1;
   do xchg(&keys, &s.lock) while (keys != 0);
   s.count++;
   if (s.count ≤ 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;}
   s.lock = 0;     }
```

acquire()

release()

```
struct semaphore {
   int count;
   queueType queue;};
```

lock = 0

WHERE?

User-Level Semaphores – **Overview**

Usage:

1. Critical Section

```
//Semaphore for critical section

Semaphore S = 1 ; //only 1 process can enter critical section
```

Function1()	Function2()
{	{
...	S.Wait()
S.Wait()	**<critical section>**
<critical section>	S.Signal()
S.Signal()	...
...	}
}	

User-Level Semaphores – **Overview**

Usage:

2. Synchronization

```
//Semaphore for dependency (Function1 depends on Function2)

Semaphore D1 = 0 ; //block first until released

Semaphore D2 = 20 ; //start first until blocked
```

Function1()	Function2()
{	{
...	...
D1.Wait()	**Required Code**
Dependent code	D1.Signal()
...	}
}	

User-Level Semaphores – **Overview**

Pros:

1. **No busy-waiting**
2. Applicable to **any number of processes**
3. Applicable on **Uni or Multi processors**
4. Support **multiple critical sections**
5. **No starvation**
   - selection of a waiting process is **FIFO**
6. **No deadlock** in **Priority Inversion** (if code is correctly written!)
   - when a **low-priority** process interrupted inside CS,
   - High-priority can't enter CS, will be BLOCKED state
   - Low-priority can continue

User-Level Semaphores – **Overview**

Cons:

1. Semaphores are **dual** purpose, slight change in order of wait's deadlock!!
2. **Deadlock** in **Priority Inversion** of its **lock**
   - when a **low-priority** process acquires the lock then interrupted,
   - High-priority executes busy-waiting on the lock
   - Low-priority can't resume! Deadlock!

Solutions:

1. priority **promotion**
2. Priority **donation**

User-Level Semaphores – **Givens**

Data Structures: inc/semaphore.h

```
struct semaphore
{
    struct __semdata* semdata ;
};
```

Wrapper

For data protection

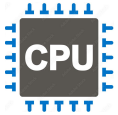
```
struct __semdata
{
    //queue of all blocked envs on this Semaphore
    struct Env_Queue queue;

    //semaphore value
    int count;

    //lock variable protecting this count
    uint32 lock;

    // For debugging: Name of semaphore.
    char name[64];
};
```

Refer to APPENDICES for:



Scheduler Functions

User-Level Semaphores – **Givens**

Queuing Functions:

kern/cpu/sched_helpers.h

```
void init_queue(struct Env_Queue* queue);

int queue_size(struct Env_Queue* queue);

void enqueue(struct Env_Queue* queue, struct Env* env);

struct Env* dequeue(struct Env_Queue* queue);

struct Env* find_env_in_queue(struct Env_Queue* queue, uint32 envID);

void remove_from_queue(struct Env_Queue* queue, struct Env* e);
```

#2: Create Semaphore

lib/semaphore.c

```
struct semaphore create_semaphore(char *semaphoreName,
                                  uint32 value)
```

1. Dynamically allocates the semaphore as a shared object (to allow other processes to share it)
2. Initializes its members

Return: object from the wrapper semaphore

#3: Get Semaphore

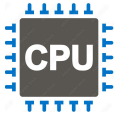
lib/semaphore.c

```
struct semaphore get_semaphore(int32 ownerEnvID, char*
                               semaphoreName)
```

- Get the shared semaphore object

Return: object from the wrapper semaphore

Refer to APPENDICES for:



Scheduler Functions

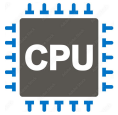
#4: Wait

lib/semaphore.c

```
void wait_semaphore(struct semaphore sem)
```

- Implement the logic of the "**wait**" function
- Refer to the previous pseudocode for details

Refer to APPENDICES for:



Scheduler Functions

#5: Signal

lib/semaphore.c

```
void signal_semaphore(struct semaphore sem)
```

- Implement the logic of the "**signal**" function
- Refer to the previous pseudocode for details

"Congratulations!! test [TEST NAME] completed successfully."
To ensure the test success, a congratulations message like this **MUST appear without any ERROR messages or PANICs**.

User-Level Semaphores: SEEN Test

- Test each function in MS3 independently in a **FRESH SEPARATE RUN**
- The time limit of each individual test: **max of 30 sec / each**
- All below tests are used to validate the **4 functions TOGETHER**

#	Test Functionality	Test
1	***tst_semaphore_1master.c (tsem1):*** tests the implementation of wait & signal by using the Semaphores for handling the critical section and the dependency.	**FOS>** run tsem1 100
2	***tst_semaphore_2master.c (tsem2):*** tests the implementation of wait & signal by using the Semaphores for allowing certain number of processes to enter the critical section at the same time.	**FOS>** run tsem2 100 Enter total # cust's: **100** Enter shop capacity: **30**

3	***MidTermEx_Master.c (midterm):*** tests the effect of race condition in simple concurrent execution of two processes that attempt to edit same shared variables (X & Y). Try it with & without using semaphores (observe the final value of X, is it same each time?)
	`ProcA { Y = X * 2; X = Y; Signal(T); }`
	`ProcB { Wait(T) Z = X + 1; X = Y; }`
		FOS> run midterm 100

User-Level Semaphores: **UNSEEN** Test

#	Test Functionality
1	Tests a complete concurrent program using semaphores (e.g. barber shop, prod cons, …etc)

Agenda

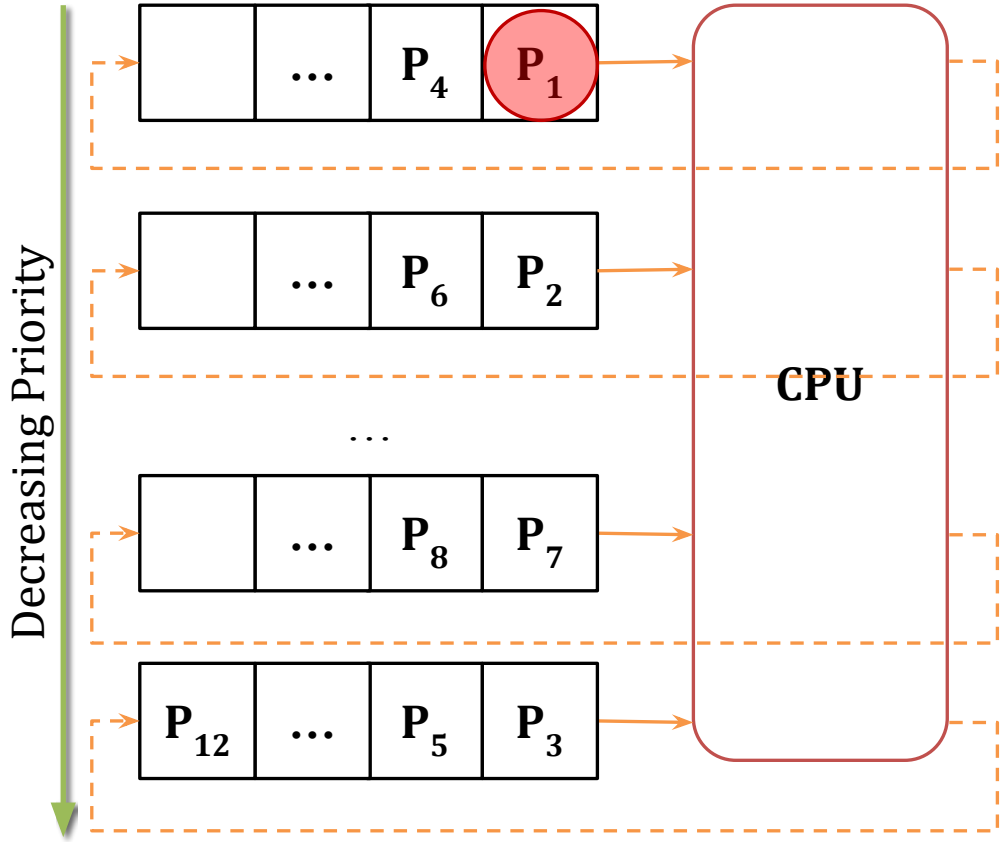
- Logistics
- Part 0: Code Updates
- Part 1: Fault Handler II (Replacement)
- Part 2: User-Level Semaphores
- **Part 3: Priority RR Scheduler**
- OVERALL Testing
- BONUSES
- Summary & Quick Guide
- How to submit?

Refer to APPENDICES for:

Scheduler Functions

Priority RR Scheduler: **Overview**

- **WHAT?**
  - multiple **ready queues** to represent each **level** of **priority**
  - Preemptive on clock
  - At any given time:
    - the scheduler chooses a process from the **highest-priority non-empty** queue.
    - If the highest-priority queue contains multiple processes, then they run in "**round robin**" order.

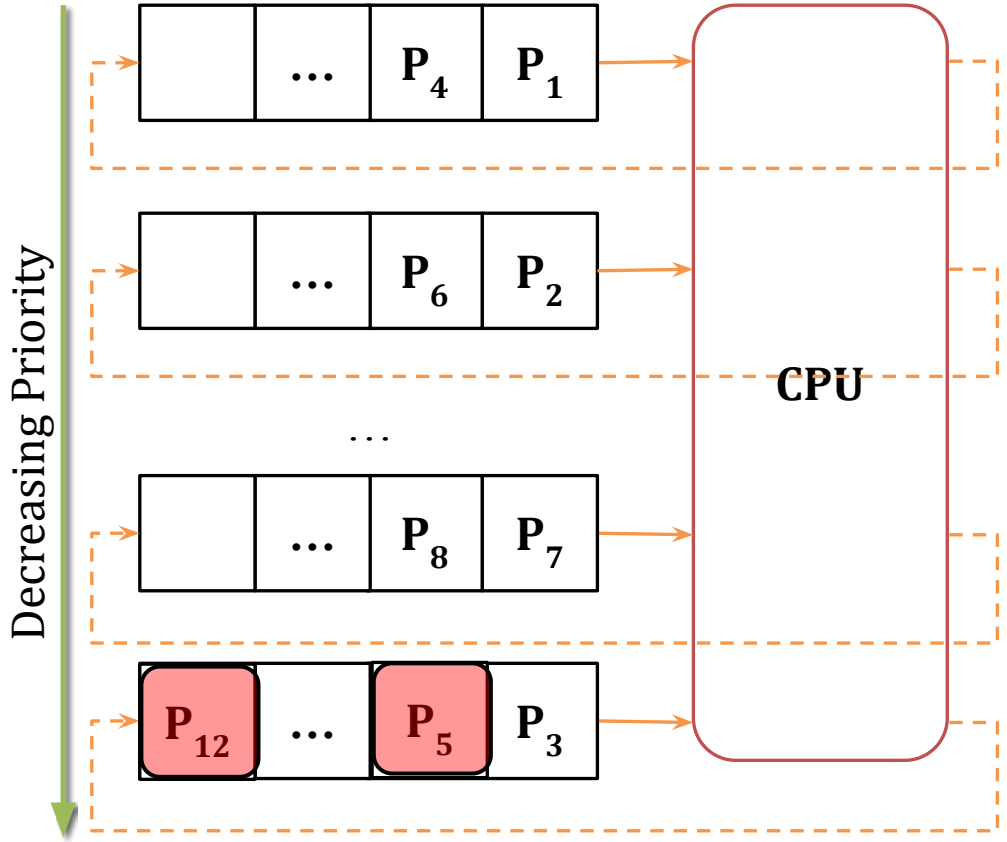


Refer to APPENDICES for:

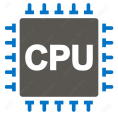
Scheduler Functions

Priority RR Scheduler: **Overview**

- **PROBLEM**
  - Lower-priority may suffer **starvation** if there is a steady supply of high priority processes.
- **SOLUTION**
  - Allow a process to **change its priority** based on **its age** (IF #TICKS EXCEEDS CERTAIN THRESHOLD)



Refer to <u>APPENDICES</u> for:



Scheduler Functions

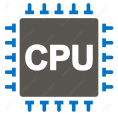
Priority RR Scheduler: **Overview**

REMEMBER

During your solution, any SHARED data need to be PROTECTED by critical section via LOCKS

Given Data Structures & Functions

Refer to **APPENDICES** for:



Scheduler Functions

kern/cpu/sched.h

Queues

```
struct
{
    struct spinlock qlock;                  //SpinLock to protect all process queues
    struct Env_Queue env_new_queue;         // queue of all new envs
    struct Env_Queue env_exit_queue;        // queue of all exited envs
    struct Env_Queue *env_ready_queues; // Ready queue(s) for the MLFQ or RR
}ProcessQueues;
```

CPU Scheduler

```
uint8 *quantums ;                       // Quantum(s) in ms
uint8 num_of_ready_queues ;             // Number of ready queue(s)
```

Queues Function

kern/cpu/sched_helpe

```
void sched_insert_ready0(struct Env* env);
void sched_insert_ready(struct Env* env);
void sched_remove_ready(struct Env* env);
void sched_insert_new(struct Env* env);
void sched_remove_new(struct Env* env);
void sched_insert_exit(struct Env* env);
void sched_remove_exit(struct Env* env);
```

- **Insert process in the corresponding ready queue (based on priority)**
- **MUST be called in the entire project instead of `sched_insert_ready0`**

Given Commands

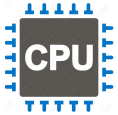
Refer to APPENDICES for:

Ready-Made Commands

1. **FOS>** run <prog_name> <page_WS_size> [<priority>]
2. **FOS>** load <prog_name> <page_WS_size> [<priority>]
3. **FOS>** setPri <envID> <priority>
4. **FOS>** setStarvThr <starvationThreshold>
5. **FOS>** runall
6. **FOS>** printall
7. **FOS>** sched?

#6: Set Process Priority & Thresh

Refer to APPENDICES for:



Scheduler Functions

```
void env_set_priority(int envID, int priority)
```

kern/cpu/sched_helpers.c

Description:

1. **Set** the priority of the given process by the given priority value
2. If it's in **READY state,** update its location in the ready queues

```
void sched_set_starv_thresh(uint32 starvThresh)
```

Description:

- **Set** the starvation threshold by the given value

New System Call

```
void sys_env_set_priority(int32 envID, int priority)
```

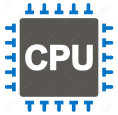
Description:

- **Implement** and handle a new system call to set the **priority** of the user process from user level
- Should call the `env_set_priority(...)` from the kernel
- Should be named as `sys_env_set_priority(...)`
- **Refer** to MS#1 for steps

Testing: UNSEEN: test at your own

#7: Initialize Priority RR Scheduler

Refer to APPENDICES for:



Scheduler Functions

```
void sched_init_PRIRR(uint8 numOfPriorities, uint8 quantum, uint32 starvThresh)
```

kern/cpu/sched.c

Description:

- **Initialize** the Priority RR scheduler by the given number of priorities, CPU quantum (in millisecond) and starvation threshold
- Do other initializations (if any)
- Should use the following **global variables** for initialization (declared in **`kern/cpu/sched.h`**)

```
struct Env_Queue *env_ready_queues; // Ready queue(s)
uint8 *quantums ;                   // Quantum(s) in ms
uint8 num_of_ready_queues ;         // Number of ready queue(s)
```

New Command

```
FOS> schedPRIRR <numOfPriority> <quantum> <starvThresh>
```

kern/cmd/commands.c

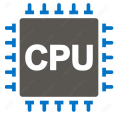
Description:

- **Switch & initialize** the scheduler to PRIORITY RR with given #priorities, quantum and starvation threshold
- Should call the **`sched_init_PRIRR(...)`**
- Should be named as **`schedPRIRR`**

Testing:

- **UNSEEN: test at your own**

Refer to <u>APPENDICES</u> for:



Scheduler Functions

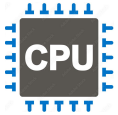
#8: Schedule Next Process

```
struct Env* fos_scheduler_PRIRR()
```

Description:

- **If there's** a current **process on the CPU**, place it in the corresponding ready queue (do any required initializations)
- **Select** the next environment to be run on the CPU and return it
- **REMEMBER** to se the CPU quantum

Refer to APPENDICES for:



Scheduler Functions

#9: Timer Tick Handler

```
void clock_interrupt_handler()
```

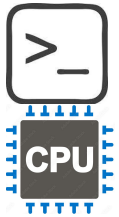
Description:

- This handler is automatically **called every "quantum"** period
- Should be used to **promote** any process that exceeds the **starvation threshold**
  - IF #TICKS IT EXCEEDS THE STARVATION THRESHOLD

Priority RR Scheduler: **Switching…**

- To switch the scheduler from the FOS prompt:
  - **`FOS> schedPRIRR <#priorities> <quant> <starvThresh>`**
    - switch the scheduler to Priority RR with the given #priorities, quantum and starvation threshold
  - **`FOS> schedRR <quantum>`**
    - switch the scheduler to RR with the given quantum

Refer to APPENDICES for:



Ready-Made Commands

Scheduler Functions

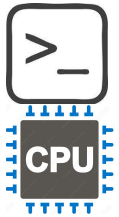
Priority RR Scheduler: **Testing**

- **UNSEEN Tests…**
- As a guide, you can use the **given** set of **ready-made programs** to test the following aspects:

1. Load & run multiple instances from the **same program** with **different "priority"** values.
2. Load & run **different** set of **programs** with **same or different "priority"** values and compare!
3. Write a master program that create & run a **large number** of different processes with different needs (i.e. cpu, I/O, memory).
   - Use `sys_create_env()` and `sys_run_env()`

…

Refer to APPENDICES for:



Ready-Made Commands

Scheduler Functions

Priority RR Scheduler: **Testing**

- **UNSEEN Tests…**
- **Example**: Run 2 instance of Fibonacci with different priorities
  - LARGE starvation threshold
    1. `FOS> schedPRIRR 5 10 1000`
    2. `FOS> load fib 100 0` //priority 0
    3. `FOS> load fib 100 4` //priority 4
    4. `FOS> runall`
  - SMALL starvation threshold
    1. `FOS> schedPRIRR 5 10 10`
    2. `FOS> load fib 100 0` //priority 0
    3. `FOS> load fib 100 4` //priority 4
    4. `FOS> runall`

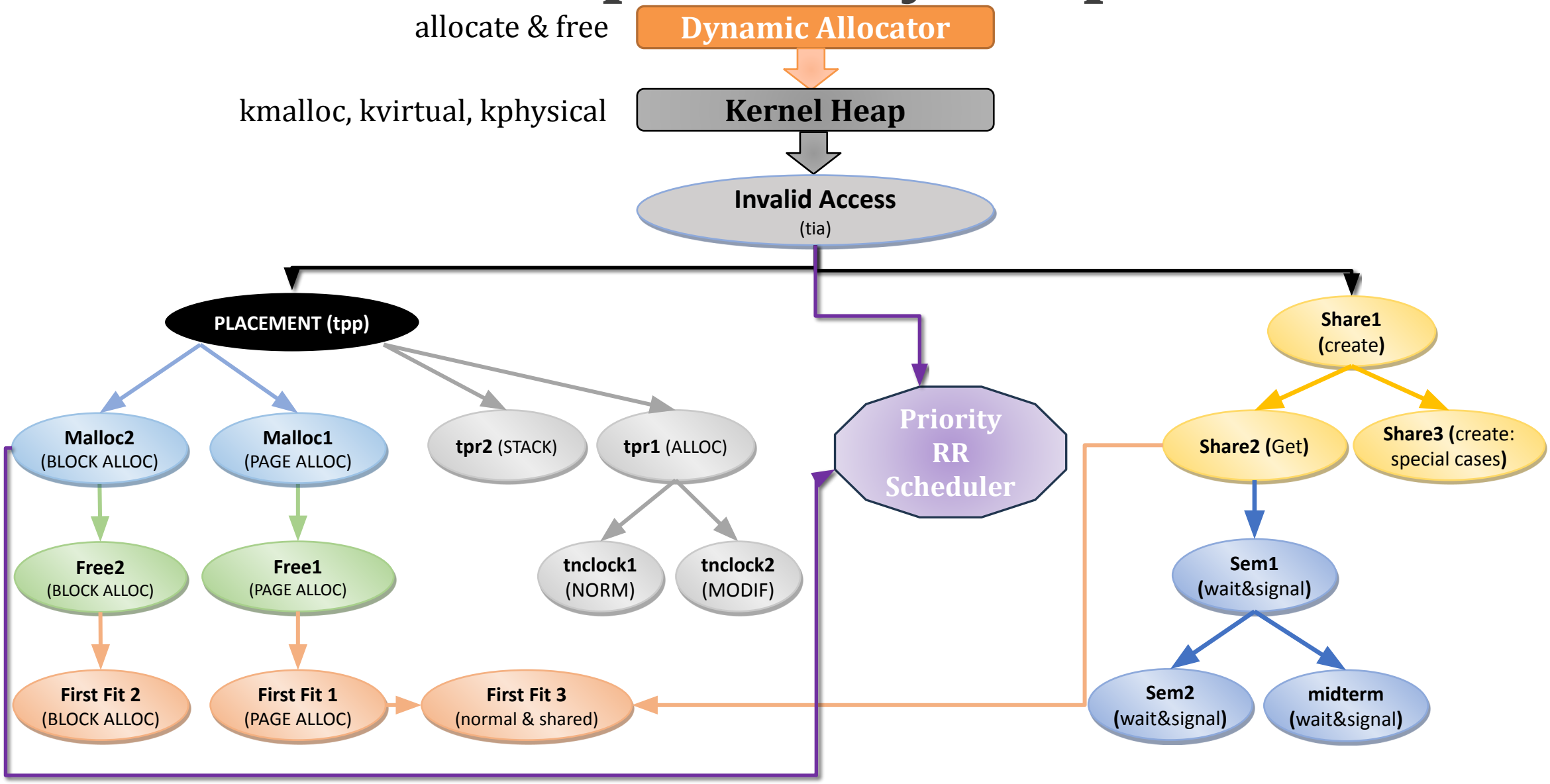
LARGE: observe that the 2nd program will not start until the 1st one finishes

SMALL: observe that the 2nd program will start during the execution of the 1st one (due to priority promotion)

Agenda

- Logistics
- Part 0: Code Updates
- Part 1: Fault Handler II (Replacement)
- Part 2: User-Level Semaphores
- Part 3: Priority RR Scheduler
- **OVERALL Testing**
- **BONUSES**
- Summary & Quick Guide
- How to submit?

Modular Tests: **Dependency Graph**



Project ENTIRE Tests: **Intro**

GIVEN

- Set of **ready-made C programs** to test the entire project in different scenarios

REQUIRED

- Use these programs to **test & validate** that the entire project will run successfully

EVALUATION

- **FIVE UNSEEN Scenarios** (1 mark/each). The time limit of each one: **max of 1 min / each**

Project ENTIRE Tests: **Programs**

To run each program: `FOS> run <prog name> <WS Size> [<priority>]`

To load multiple programs: `FOS> load <prog name> <WS Size> [<priority>]`

...

`FOS> runall`

I	Program	Params to Play With!
1	***fos_factorial.c (fact):*** calculate the factorial of the given integer (recursive code)	1. Input integer 2. Working set size 3. Priority
2	***fos_fibonacci.c (fib):*** calculate the Fibonacci value of the given index (recursive code)	1. Fibonacci index 2. Working set size 3. Priority
3	***arrayOperations_Master.c (arrop):*** test the shared memory & semaphore modules by creating & initializing a shared array, then run 3 programs that apply different operations on this array (quicksort, mergesort and statistics). The four processes use semaphores for sync.	1. Array size 2. Initializ. (Asc, Ident., Random) 3. Working set size 4. Priority

Project ENTIRE Tests: **Programs**

To run each program: `FOS> run <prog name> <WS Size> [<priority>]`

To load multiple programs: `FOS> load <prog name> <WS Size> [<priority>]`

...

`FOS> runall`

I	Program	Params to Play With!
4	***quicksort_noleakage.c (qs1):*** apply the quick-sort recursive algorithm to sort a pre-initialized array of the given size (using malloc()). The created array will be deleted every time (using free())	1. Array size 2. Initializ. (Asc, Desc, Random) 3. Working set size 4. Priority
5	***quicksort_leakage.c (qs2):*** apply the quick-sort recursive algorithm to sort a pre-initialized array of the given size (using malloc()). The created array will NOT be deleted, leading to memory leakage.	1. Array size 2. Initializ. (Asc, Desc, Random) 3. Working set size 4. Priority

Project ENTIRE Tests: **Programs**

To run each program: `FOS> run <prog name> <WS Size> [<priority>]`

To load multiple programs: `FOS> load <prog name> <WS Size> [<priority>]`

...

`FOS> runall`

I	Program	Params to Play With!
6	***mergesort_noleakage.c (ms1):*** apply the merge-sort recursive algorithm to sort a pre-initialized array of the given size (using malloc()). The auxiliary arrays in the "Merge" function are deleted every time (using free())	1. Array size 2. Initializ. (Asc, Desc, Random) 3. Working set size 4. Priority
7	***mergesort_leakage.c (ms2):*** apply the merge-sort recursive algorithm to sort a pre-initialized array of the given size (using malloc()). The auxiliary arrays in the "Merge" function are **NOT** deleted (i.e. causing memory leakage)	1. Array size 2. Initializ. (Asc, Desc, Random) 3. Working set size 4. Priority

Project ENTIRE Tests: **Scenarios**

Possible Scenarios:

1. Run Single Program with Different Params
    1. Can be used to test **ALL** Modules **except** CPU scheduling
    2. To Run any program:

        FOS> run <prog name> <WS Size> [<priority>]

    3. Examples for Some Scenarios
        1. Test the same program with **different WS sizes** (very small, medium, large and very large sizes)
        2. Compare **NORMAL vs. MODIFIED versions** of N^{th} Chance Clock algorithm on different WS sizes
        3. Compare **mem-leakage** program **vs. non-leaky** ones

Project ENTIRE Tests: **Scenarios**

Possible Scenarios:

2. Run Multi-Programs at the Same Time with Different Params
    1. Can be used to test **ALL** Modules **including** CPU scheduling
    2. To load multiple programs & run them at once:

```
FOS> load <prog name> <WS Size> <priority>
FOS> load <prog name> <WS Size> <priority>
…
FOS> runall
```

    3. Examples for Some Scenarios
        1. Run **instances of the same program** with different priorities. Play with the starvation threshold.
        2. Run **set of programs** with different priorities. Play with the starvation threshold.

Bonuses

Bonuses

1. Free the Entire Process (env_free) V.1 (without shared & semaphores)

1. All pages in the page working set
2. Working set itself
3. All page tables in the entire user virtual memory
4. Directory table
5. User kernel stack
6. All pages from page file, this code ***is already*** written for you ☺️

Bonuses

2. Free the Entire Process (env_free) V.2 (including shared & semaphores)

1. All pages in the page working set
2. Working set itself
3. **ALL shared objects (if any)**
4. **ALL semaphores (if any)**
5. All page tables in the entire user virtual memory
6. Directory table
7. User kernel stack
8. All pages from page file, this code ***is already*** written for you ☺️

Bonuses

3. Efficient Implementation of N^{th} Chance Clock Replacement

- Implement the main logic of replacement in O(N) (N: WS size)
  - Neglecting complexity of read/write from/to page file
- Compare the performance of this implementation with the naïve one

Bonuses

4. FOS Enhancement

- If you **discover** any **issue** in the FOS:
  1. Performance issue
  2. Security/Protection issue
  3. Any other design/technical issue to act as a real OS
- Try to **get a solution** for it. (no implementation, just the solution idea)
- Fill-up this document with TWO main sections:
  - The **issue** explained in a detailed and clear way.
  - The **solution** explained in a detailed and clear way.

DELIVERY: Discuss it with the Lecturer in online session isA

Agenda

- Logistics
- Part 0: Code Updates
- Part 1: Fault Handler II (Replacement)
- Part 2: User-Level Semaphores
- Part 3: Priority RR Scheduler
- OVERALL Testing
- BONUSES
- **Summary & Quick Guide**
- **How to submit?**

Summary

Module	Function	Difficulty	Testing	Files
Fault Handler II	N^{th} Chance Clock Replacement [NORMAL & MODIFIED]	**L3**	**[NORMAL Version]** **FOS>** nclock 5 1 **FOS>** run tpr1 11 **FOS>** run tpr2 6 **FOS>** run tnclock1 11 **[MODIFIED Version]** **FOS>** nclock 5 2 **FOS>** run tpr1 11 **FOS>** run tpr2 6 **FOS>** run tnclock2 11	Kern/trap/fault_handler.c

"Congratulations!! test [TEST NAME] completed successfully."

To ensure the test success, a congratulations message like this **MUST appear without any ERROR messages or PANICs**.

Summary

Module	Function	Diff.	Testing	Files
User-Level Semaphore	Create Semaphore	**L1**	**SEEN Tests** **1. FOS>** run tsem1 100 **2. FOS>** run tsem2 100 Enter total # cust's: **100** Enter shop capacity: **30** **3. FOS>** run midterm 100 **UNSEEN Tests**	lib/semaphore.c
	Get Semaphore	**L1**		
	Wait Semaphore	**L2**		
	Signal Semaphore	**L2**		

Module	Function	Diff.	Testing	Files
Priority RR Scheduler	Set Process Priority & Threshold	**L2**	**Test at your own…** **UNSEEN Tests**	kern/cpu/sched_helpers.c & system call files
	Initialize Priority RR Scheduler	**L2**		kern/sched/sched.c, kern/cmd/commands.c
	Schedule Next Process	**L1**		kern/sched/sched.c
	Timer Tick Handler	**L1**		

Module	Testing	Evaluation
OVERALL TESTING	**UNSEEN FIVE Testing Scenarios** to test the entire project	5 MARKS (1/each)

REMEMBER:

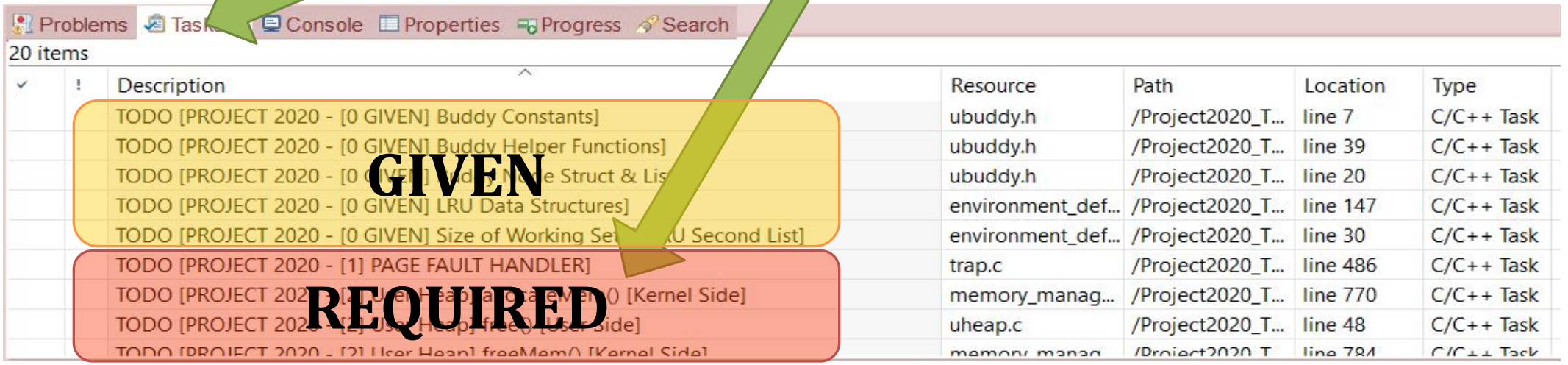
- **UPDATE** YOUR CODE ACCORDING TO PREVIOUSLY DESCRIBED STEPS
- **READ** ATATCHED APPENDICES FOR HELPER FUNCTIONS.

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



[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 }
```

Submission Rules

Read the following instructions as the code correction is done AUTOMATICALLY. Any violation in these rules will lead to 0 and, in this case, nothing could be happened.

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: of Week #13 (21/12 @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 OS** ☺️