January 29 – Feb 2

Processes and threads are both fundamental concepts in operating systems and concurrency, but they serve slightly different purposes and have different characteristics:

1. **Process**:
   - A process is an instance of a program that is being executed.
   - Each process has its own memory space, containing the program code, data, and resources allocated to it.
   - Processes are independent entities; they cannot directly access the memory of other processes.
   - Communication between processes typically involves more overhead, such as inter-process communication (IPC) mechanisms like pipes, sockets, or shared memory.
   - Processes are heavyweight entities in terms of resource consumption. Creating a process involves significant overhead, including memory allocation, initialization, and context switching.
   - Processes provide more isolation and protection because of their separate memory spaces, making them more resilient to faults.
2. **Thread**:
   - A thread is a lightweight process within a process.
   - Threads share the same memory space as their parent process and other threads within the same process.
   - Threads within the same process can communicate with each other more easily and efficiently, as they share memory directly.
   - Communication between threads is typically faster compared to processes due to the shared memory space.
   - Threads are lighter in terms of resource consumption compared to processes. Creating a thread involves less overhead, as it shares resources with its parent process.
   - Threads are used to achieve concurrency within a process. Multiple threads within a process can execute concurrently, performing different tasks simultaneously.
   - Threads are more prone to issues like race conditions and deadlocks due to shared memory, requiring careful synchronization mechanisms to manage.

Context Switching - Context switching refers to the process of saving and restoring the state of a CPU (Central Processing Unit) so that multiple processes or threads can share the same CPU. This occurs when the CPU switches from executing the instructions of one process or thread to executing the instructions of another.

Here's what happens during a context switch:

1. **Save the Current State**: The CPU saves the current state of the currently running process or thread, including the values of its registers, program counter, and other relevant information. This is necessary to ensure that the process or thread can resume its execution later from the same point.
2. **Load the State of the Next Process/Thread**: The CPU loads the saved state of the next process or thread that is scheduled to run. This involves retrieving the saved values of registers, program counter, etc., so that the execution can continue from where it left off for this process or thread.
3. **Update Memory Management**: If the context switch involves switching between processes (as opposed to just threads within the same process), the memory management unit may need to be updated to reflect the memory mappings of the new process.
4. **Update CPU Scheduler**: The scheduler, which is responsible for deciding which process or thread should run next, updates its data structures to reflect the new state of the CPU and the processes/threads in the system.

1. **Race Conditions**:
    - A race condition occurs in concurrent programming when the outcome of the execution depends on the relative timing of events.
    - It arises when multiple threads or processes access shared resources (such as variables or data structures) concurrently, and the final result depends on the order of their execution.
    - Race conditions can lead to unpredictable or erroneous behavior in a program because the outcome may vary depending on factors like scheduling decisions made by the operating system.
    - To mitigate race conditions, synchronization mechanisms such as locks, mutexes, semaphores, or atomic operations are used to coordinate access to shared resources and ensure that only one thread can modify the resource at a time.
2. **Deadlocks**:
    - A deadlock occurs in concurrent systems when two or more processes or threads are waiting indefinitely for each other to release resources that they need to proceed.
    - Deadlocks typically arise in systems with shared resources where processes hold resources while waiting to acquire additional resources. If multiple processes are each waiting for resources held by others, a deadlock can occur.

- Deadlocks can halt the progress of all affected processes, leading to system-wide unresponsiveness.
- **Deadlocks are often detected and resolved through careful design, avoiding circular waits, and using techniques such as resource allocation graphs or timeouts to break deadlocks.**
- Prevention strategies include ensuring that processes acquire resources in a fixed order, using resource hierarchy, or implementing deadlock avoidance algorithms.
- Example : -

```
Thread 1:                    Thread 2:
pthread_mutex_lock(L1);      pthread_mutex_lock(L2);
pthread_mutex_lock(L2);      pthread_mutex_lock(L1);
```

Figure 32.6: **Simple Deadlock (deadlock.c)**

• A critical section is a piece of code that accesses a shared resource, usually a variable or data structure.

• A race condition (or data race [NM92]) arises if multiple threads of execution enter the critical section at roughly the same time; both attempt to update the shared data structure, leading to a surprising (and perhaps undesirable) outcome.

**-ROUND ROBIN (RR) aka time-slicing**

- More generally, any policy (such as RR) that is fair, i.e., that evenly divides the CPU among active processes on a small time scale, will perform poorly on metrics such as **turnaround time**. Indeed, this is an inherent trade-off: if you are willing to be unfair, you can run shorter jobs to completion, but at the cost of response time; if you instead value fairness, response time is lowered, but at the cost of turnaround time. This type of trade-off is common in systems; you can't have your cake and eat it too4 . We have developed two types of schedulers. The first type (SJF, STCF) optimizes turnaround time, but is bad for response time. The second type (RR) optimizes response time but is bad for turnaround.

-RR is fair when it comes to response time and SJF,STCF is fair when it comes to turnaround time.

-**Bounded Buffer Problem aka The Producer/Consumer**

-Imagine one or more producer threads and one or more consumer threads. Producers generate data items and place them in a buffer; consumers grab said items from the buffer and consume them in some way. This arrangement occurs in many real systems. For example, in a multi-threaded web server, a producer puts HTTP requests into a work queue (i.e., the bounded buffer); consumer threads take requests out of this queue and process them.

- Bounded buffer is also used when you pipe the output of one program into another, e.g., grep foo file.txt | wc -l. This example runs two processes concurrently; grep writes lines from file.txt with the string foo in them to what it thinks is standard output; the UNIX shell redirects the output to what is called a UNIX pipe (created by the pipe system call). The other end of this pipe is connected to the standard input of the process wc, which simply counts the number of lines in the input stream and prints out the result. Thus, the grep process is the producer; the wc process is the consumer;

**-Semaphores**

-A semaphore is a synchronization primitive used in concurrent programming and operating systems to control access to shared resources by multiple processes or threads. Semaphores can be thought of as simple integer variables that are used to coordinate access to resources. They typically have two primary operations:

-**Wait (P) Operation**: Also known as "down" or "acquire". When a process or thread wants to access a shared resource, it performs a wait operation on the semaphore. If the semaphore's value is greater than zero, indicating that the resource is available, the semaphore's value is decremented, and the process continues execution. If the semaphore's value is zero, indicating that the resource is currently being used, the process is blocked until the semaphore's value becomes positive.

**Signal (V) Operation**: Also known as "up" or "release". When a process or thread finishes using a shared resource, it performs a signal operation on the semaphore. This increments the semaphore's value, indicating that the resource is now available. If there are other processes or threads waiting for the resource (i.e., blocked on a wait operation), one of them is unblocked, allowing it to proceed.

Example =

```
// Initialize semaphore with value 1 (indicating resource is initially available)
Semaphore mutex = 1;

// Process 1
wait(mutex);   // Attempt to acquire the semaphore
// Critical section: access the shared resource
// (e.g., updating a global variable, writing to a file)
signal(mutex); // Release the semaphore

// Process 2
wait(mutex);   // Attempt to acquire the semaphore
// Critical section: access the shared resource
// (e.g., reading from the global variable, accessing the file)
signal(mutex); // Release the semaphore
```

In this example:

1. We initialize a semaphore called **mutex** with an initial value of 1. This semaphore will be used to control access to the shared resource.
2. Each process executes the **wait(mutex)** operation to acquire the semaphore before accessing the critical section (the shared resource). If the semaphore's value is 1 (indicating the resource is available), the semaphore's value is decremented to 0, allowing the process to enter the critical section. If the semaphore's value is 0 (indicating the resource is currently being used), the process will wait until the semaphore's value becomes 1.
3. Once a process finishes accessing the critical section, it executes the **signal(mutex)** operation to release the semaphore. This increments the semaphore's value from 0 to 1, indicating that the resource is now available. If there are other processes waiting for the semaphore, one of them will be unblocked and allowed to enter the critical section.

## Conditions for Deadlock

Four conditions need to hold for a deadlock to occur [C+71]:

- **Mutual exclusion:** Threads claim exclusive control of resources that they require (e.g., a thread grabs a lock).
- **Hold-and-wait:** Threads hold resources allocated to them (e.g., locks that they have already acquired) while waiting for additional resources (e.g., locks that they wish to acquire).
- **No preemption:** Resources (e.g., locks) cannot be forcibly removed from threads that are holding them.
- **Circular wait:** There exists a circular chain of threads such that each thread holds one or more resources (e.g., locks) that are being requested by the next thread in the chain.

If any of these four conditions are not met, deadlock cannot occur. Thus, we first explore techniques to *prevent* deadlock; each of these strategies seeks to prevent one of the above conditions from arising and thus is one approach to handling the deadlock problem.

### Hold-and-wait

The hold-and-wait requirement for deadlock can be avoided by acquiring all locks at once, atomically. In practice, this could be achieved as follows:

```
1  pthread_mutex_lock(prevention);   // begin acquisition
2  pthread_mutex_lock(L1);
3  pthread_mutex_lock(L2);
4  ...
5  pthread_mutex_unlock(prevention); // end
```

[1]Hint: "D" stands for "Deadlock".

- The proposed solution suggests acquiring all required locks atomically to prevent the hold-and-wait condition. Before acquiring individual locks, a global prevention lock (**prevention**) is acquired. This lock ensures that no untimely thread switch occurs during the process of acquiring multiple locks, thus preventing deadlock.

//----------

**WINTER 2023 (ROB - MIDTERM)**

**Q1) Define mutex like you were explaining the concept to your 90 year old. Use real world example to illustrate your case**

-

Imagine you have a cookie jar, and you love cookies. Now, imagine that you have a bunch of grandkids who also love cookies. You all want to get cookies from the jar, but only one person can have the jar at a time, or there will be chaos!

A mutex is like a special rule you make for your grandkids. You say, "Only one person can hold the cookie jar at a time. If someone has the jar, others have to wait until it's their turn."

So, when one grandkid wants a cookie, they have to ask for permission. If nobody else has the jar, they get it and take a cookie. While they have the jar, they are said to have the "mutex" or control over the jar. Nobody else can take it until they're done and put it back.

Now, imagine if two grandkids try to take the jar at the same time without asking. That could cause a mess, right? They might bump into each other or drop the jar. But with the mutex rule, only one person can have the jar at a time, so there's no chaos.

In computer programs, a mutex works similarly. It helps different parts of the program take turns using shared resources, like memory or files. Just like with the cookie jar, it prevents chaos by making sure only one part of the program can access the shared resource at any given time.

**Q2)**

2. (3 points) Consider the following locking code:

```
1 void mutex_lock(_Atomic int *lock)
2 {
3     while(*lock == 1) ; -> A preempted after this line, due to interrupt
4     *lock = 1;            B runs all the way, A resumes at 4.
5 }
6 void mutex_unlock(_Atomic int *lock)
7 {
8     *lock = 0;
9 }
```

Does this lock create a mutex lock? If yes, explain why it is successful at creating a mutex section. If no, explain why it does not succeed at creating a mutex section.

For either, feel free to doodle on the code above.

-No this does not create a mutex lock:-

Thread A and Thread B, both trying to acquire the same lock.

```plaintext
_Thread A_

1. Thread A calls mutex_lock(&lock).
2. It enters the while loop and continuously checks the value of *lock.
3. Eventually, it observes that *lock == 0 and proceeds to set *lock = 1.

_Thread B_

1. Thread B also calls mutex_lock(&lock) almost simultaneously with Thread A.
2. It enters the while loop and continuously checks the value of *lock.
3. It also observes that *lock == 0 at the same time as Thread A.
4. Thread B proceeds to set *lock = 1, as it assumes the lock is available.
5. Now both Thread A and Thread B believe they have acquired the lock, leading to a c
```

-

- Thread A and Thread B both check if the lock is available at the same time.
- They both see that it's available (because they check simultaneously).
- They both try to take the lock, thinking they've got it.

This leads to a situation where both threads think they've got exclusive access to the critical section (like playing with the toy), but in reality, they're both accessing it at the same time, breaking the rule.

In a correct mutex implementation, only one thread should successfully take the lock while others wait until it's free. This ensures that the critical section is accessed by only one thread at a time, just like Alice and Bob should take turns with the toy.

**Q3) b) What does dup2 do? Where is the data it is changing stored? What is dup2 doing above to make the pipeline work?**

-Dup2 duplicates the open file descriptor onto another file descriptor. The data it is changing is stored in file descriptor table maintained by operating system kernel. In the above code dup2 duplicates the STDIN and STDOUT of a process to pipe in and pipe

out. `dup2()` is used to set up the communication between the processes through pipes. It ensures that the output of one process is directly fed as input to another process, thereby establishing the pipeline

**c) When exec is called, what about the process is changing? Explain the changes make when exec is called, highlighting how the process is the same and/or different afterwards.**

-When exec is called it replaces the current process image with a new one. This new program completely replaces current process including code, data, heap, stack and open file table. This means that any files opened by the previous program will be closed and new files opened by new program will be added to the open file table. Only Process_ID, parent process id will remain same which means the process remains same from the OS perspective.

**Q4) What is a system call? What is different about calling a system call than a function in your own code?**

-A System call is a request made by an active process to the operating system kernel. It's a way for user-level programs to request services from the operating system, such as reading or writing to a file, creating a new process, or allocating memory.

- System call is built-in kernel mode, while normal functions are provided by user code.

-When calling function in code makes system call, it has to transition from user mode to kernel mode, where the operating system has more privileges and can perform tasks that user-mode programs cannot, like managing hardware resources. This means that system call have more privileges and freedom to manage resources rather then asking the CPU to switch user to kernel mode and handle system call.

**Q5) a) There are 10 users logged into a single aviary machine running many simulations to collect data for an assignment. Assuming that the aviary machines use a MLFQ scheduler, would it be fair to all 10 users. Why or why not?**

- It will be less fair to all 10 users. For example – since these are all similar programs running for a long time and if those programs require intensive resources, they will compete for CPU, memory or I/O resources, which could hog the CPU or consume a large portion of memory, potentially affecting the performance of other simulations and users.

-Additionally, the fairness may be compromised because some users may game the scheduler by using time slice to stay in the top priority queue.

**b) If we would swap the scheduler to round robin scheduler, would the system be more, the same, or less fair to these 10 users. Why?**

-Round Robin would be the same or slightly more fair because it will allocate CPU time equally among all processes or users so each will receive a fair share of CPU time.

**Q 6) Livelock**

**a) what is livelock? What deadlock-avoidance strategy is already employed if we are seeing livelock?**

- Livelock is a situation in concurrent systems where two or more processes continuously change their states in response to the actions of one another, but no progress is made. Unlike deadlock, where processes are blocked and unable to proceed, in a livelock situation, processes are active but unable to make any meaningful progress.

- Livelock often occurs when multiple processes are trying to resolve a resource conflict by repeatedly yielding or retrying their operations, leading to a cyclic pattern of behavior that prevents any process from making progress.

- Deadlock avoidance strategies: Not doing Hold and Wait. Ordering of resource requests. Resource allocation graph.

**b) Without introducing deadlock, what strategies can we use to avoid livelock?**

-Preemption, that is forcibly removed resources from threads that are holding them.

-Fair resource allocation

- Timeouts

-Randomized backoffs.

**Q7) Could code that is using Semaphores have a deadlock? Why or why not? If yes, provide an example. If no, explain why the semaphore solves the deadlock.**

-Yes the code that is using Semaphore may have a deadlock. For example – if two processes are set to wait for each other on the same semaphore, neither will signal the other and the system will eventually be deadlocked.

**SUMMER – 2023 (Franklin - midterm)**

**Q1 (4 points) Write pseudocode to implement a program called runfor that takes two arguments: a command to run, and a number of seconds to run for before terminating that program.**

Here's an example of what runfor might look like when it's running:

> runfor 4 "yes"

Y

Y

Y

Y

// 4 seconds go by, lots

// of "y" printed to the

// terminal, then yes

// is ended by runfor

Here's a header file called utils.h with some functions that you can call (you don't need to implement these, they are implemented somewhere else for you):

#pragma once

// block for the specified number of seconds, then return

void sleep(int seconds);

// split a command up into parts separated by whitespace with a NULL terminator

// e.g., split_cmd("cat filename. tat") would return

// char *split_cmd(char *cmd);

Here's some C code to get you started, but remember: you're being asked to write pseudocode. Don't worry if you don't remember the exact names or order of arguments to functions (other than what are stated above).

#include <unistd.h>

#include <sys/warth.h>

->

```
function runfor(command, seconds):
    start_time = current_time()
    execute(command)  // Execute the given command
    while current_time() - start_time < seconds:
        sleep(1)  // Sleep for 1 second
    terminate(command)  // Terminate the command

function execute(command):
    // Use the provided split_cmd function to split the command
    command_parts = split_cmd(command)
    // Execute the command with proper error handling
    if fork() == 0:
        execvp(command_parts[0], command_parts)
        // If execvp returns, then there's an error
        print("Error: Command execution failed")
        exit(1)
```

**Q2. Here's a data structure and some supporting code that will be executed concurrently by two or more pthreads.**

```
1.typedef struct NODE{
2.    char *value;
3.    struct NODE *next;
4. }node;
5.
6. typedef struct NODE_DELETEP{
7.    node *delete_next;
8. }node_delete;
9.
```

```
10. //the list is constructed
11. //elsewhere, and ends with
12.//a NULL
13.node *head;
14. void *thread_delete_next(void *args){
15.     node_delete *delete_operation = (node_delete*) args;
16.     node *deleted = NULL;
17.     //make sure we're not trying to delete tail's next
18.     if(delete_operation->delete_next->next != NULL){
19.         deleted = delete_operation->delete_next ->next;
20.         //delete the node by setting next to next's next.
21.         delete_operation->delete_next =
22.               delete_operation->delete_next->next->next;
23.     }
24    // the caller can free this node if they need to
25.     return deleted;
26. }
```

a)

1.1 (1 mark) Identify the critical section in this code. There is a critical section in this code. Identify the range of line numbers (eg – 'lines 1-3').
Ans = 18-23?

1.2 (3 mark) Explain why this is a critical section. Your answer should show how multiple threads of execution could concurrently make changes to the data structure and result in an inconsistent data structure, or how concurrent modifications to this structure could make the program crash (e.g., Segmentation fault), either in your own words, or by drawing and annotating a diagram.
Ans – For ex, thread 1 will try to delete the node before the tail node (2$^{nd}$ last) and thread 2 will try to delete the tail node. So thread 1 will run the line 18 and will be interrupted after its execution. Thread 2 will start by checking the condition on line 18 and will succeed to delete the tail node of the list. And when thread 1 will resumes and will execute line 21-22 to delete the tail end, and the program will crash (Seg fault).

(b) Protect the critical section using a lock. Your approach will be assessed on both correctness and efficiency.

i. (2 points) Between which consecutive line numbers (e.g.. "1-2") would you declare the lock?

Ans-> Before line 1, or at line 5 or after line 13. Implement lock outside the function and outside the struct as a file scope lock. But the most efficient is to implement at line 2-3 inside the NODE struct

ii. (1 point) Between which consecutive line numbers (e.g., "1-2") would you acquire the lock? Ans-> 17 (before the conditional statement)

Iii, (1 point) Between which consecutive line numbers (e.g., "1-2") would you release the lock? Ans-> 24 (after the conditional statement)

**Q3. Here is a code segment that a process will run.**

```
1.void *my_thread(void *args){
2.    (void) args;
3.    int *numbers =
4.        malloc(sizeof(int) * 100);
5.        //do stuff with numbers
6.        return NULL;
7.}
8.int main(void){
9.    pthread_t thread;
10.    pid)t child;
11.    int pipes[2]
12.    uint8_t bytes[10];
13.    pipes(pipes);
14    child = fork();
15
16    //child branch
17    if(child == 0){
18        pthread_create(&thread,
19            NULL, my_thread, NULL);
20        close(pipes[0]);
21        read(pipes[1], bytes, 10);
22        close(pipes[1]);
23        exit(EXIT_SUCCESS;
24    }
25    //parent branch
26    else{
27        pthread_create(&thread, NULL, my_thread, NULL);
28        close(pipes[1]);
29        sleep(5);
```

```
30          write(pipes[0], "hey kid! \n\0", 10);
31          pthread_join(thread);
32      }
33      return EXIT_SUCCESS
34}
```

(a) Our textbook describes three states for processes: Running, blocked, and ready (and implicitly an end state of "exited"). Describe the lifecycle of these two processes (how they move through those states) assuming that the state of the process could change at any system call being made. The system calls made in the above code are: fork, close, read, exit, and write. A process ends when it calls exit or returns from the main function. Your answer should clearly show which line number causes the process to change states.

i. (1 and half points) Describe the lifecycle of the parent process.

Ans – In the parent process starting at line 8 it is in running state. For lines 9,10,11,12 the parent process will be in running state as no system calls has been made to OS. At line 13 the parent process will switch the mode to ready to make change in its data structure and after that change it will be switch again to running and continue executing. The same change will reflects on line 15 when the system call fork() has been made (i.e, running to ready and then running again). After that it will be returned to that parent branch of the conditional statement. On line :

Line 27 (State: running to ready to running)

Line 28 (State: running to ready to running)

Line 29 (State : blocked as the OS is not doing anything)

Line 30 (State : blocked as it is waiting for the child to read from the pipe)

Line 33 (exit state)

ii. (1 and half points) Describe the lifecycle of the child process.

Ans – Pretty much same as above but the line 21 in child is considered as blocked state as it will wait for the parent to write into the pipe.

(b) (5 points) List the lines of code that will create or modify a PCB or any other related OS structure, and state how those lines create or modify a PCB or any other related OS structure.

You can choose to do this in either of the following ways:

1. In your own words and in plain-English, referring to specific line numbers.

2. By drawing PCB(s) and open file table(s) and listing property names and values.

If the OS would pick an arbitrary value for a specific property (e.g.. the PID), you should also pick an arbitrary value.

Ans – At line 13 the pipe system call is modifying the open file table and it's putting 2 new file descriptors and it's creating a queue in the background. At line 15 it is duplicating PCB about everything in the parent process to the child process. At line 18 it will create a new TCB (thread control block). The close system call is removing the entry from open file table and at line 21 it just change the state of OS. In the parent process, it will pretty much remain the same as child process, where the only change is at line 30 where it is writing to the queue and hence the PCB will be changed.

**Q4. Here is some code that uses locks and cannot deadlock:**

```
1typedef struct{
2    int balance;
3    pthread_mutex_t balance_mutex;
4}bank_account;
5
6typedef struct{
7    bank_account *from;
8    bank_account *to;
9    int amount;
10}deposit_thr_args;
11
12pthread_mutex_t transfer_mutex = PTHREAD_MUTEX_INITIALIZER;
13
14void *deposit(void *ptr){
15    int result;
16    deposit_thr_args *args = (deposit_thr_args *)ptr;
17
18    pthread_mutex_lock(&transfer_mutex);
19    pthread_mutex_lock(&(args->from->balance_mutex));
20
21    //does 'from' have enough money to transfer this much?
22    if(args->from->balance < args->amount){
23        //no, don;t transfer anything, just bail out
24        pthread_mutex_unlock(&(args->from->balance_mutex));
```

```
25          pthread_mutex_unlock(&transfer_mutex);
26      }else{
27          //yes,continue locking
28          pthread_mutex_lock(&(args->to->balance_mutex));
29          args->from->balance = 0; -= args->amount;
30          args->to->balance += args->amount;
31
32          pthread_mutex_unlock(&(args->from->balance_mutex));
33          pthread_mutex_unlock(&(args->to->balance_mutex));
34          pthread_mutex_unlock(&transfer_mutex);
35      }
36      return (void*)ptr;
37}
```

a) Why this code cannot deadlocked, it is not very efficient.

i. (1 point) Why it is not efficient?

Ans – It is not efficient because at line 12 it is using a file scope lock to lock the critical section which means that only 1 thread at a time is allowed to call the deposit function. The rest of the threads will be blocked at line 18 so no matter how many accounts we're transferring will be blocked and only 1 will happen at a time.

ii. (2 points) How would you change this code to be both more efficient and still guarantee that deadlock will not happen? You can explain in plain English, you don't need to describe in deatil how you would change the code.

Ans – We can define the acc_ID into the struct (line 1-4) and sort the account (line 16-18) to make sure that "from" will have the lowest acc_ID and "to" will have the highest acc_ID and that will avoid the circular wait condition of deadlocked.

b) (6 points) Explain why this code cannot deadlock by listing the four conditions for deadlock(1/2 each) and describing how these four conditions apply to this code fragment ("why does the condition hold?", or "why does the condition does not hold?")(1 point each).

Ans – 4 conditions (No-preemption, mutual exclusion, circular wait and "hold and wait")

Pthread_mutex guarantees the mutual exlusion

No-preemption = pthread guarantee no-preemption as once the lock is acquired then unless the thread voluntarily gives up, we're not going to forcibly remove it.

Circular wait and “hold and wait” are broken because of having file scope lock. The transfer_mutex lock causes all of the other lock to be irrelevant. We cannot hold lock while waiting for another lock because only 1 thread at a time can hold it. So it means that the code from lines 19-36 are not concurrent. But deadlock problem is because code from lines are 19-36 are concurrent. So if we can’t run the line 19-36 concurrent then we don’t have hold and wait problem.

**Q5. Here is an attempt at implementing a lock:**

```
1typedef enum STATE{
2    UNLOCKED = 0;
3    LOCKED = 1,
4}state;
5typedef struct MY_LOCK{
6    state locked;
7}lock_t;
8
9uint64_t counter = 0;
10
11void lock_init(lock_t *lock){
12    lock->locked = UNLOCKED;
13}
14
15void lock(lock_t *lock) {
16    while(lock->locked){counter++;}
17
18    lock->locked = LOCKED;
19}
20
21void unlock(lock_t *lock){
22    lock->locked = UNLOCKED;
23}
```

a) (2 points) This lock implementation does not guarantee mutual exclusion when executed by multiple threads of execution. Describe a sequence of events (in words or using diagram) between two threads of execution where mutual exclusion is not guaranteed for threads trying to protect their own critical sections by calling the lock and unlock functions. Refer to line numbers in the above code segment.

Ans – Lock function at lines (15-19) is just doing test and set and it’s not setting the lock atomically. The order of operation will be to have a thread that test while condition so the lock is unclocked and between line 16-18 that thread gets interrupted, so it will not

execute line 18. After that another thread calls lock and it will test the same while loop and will be able to lock is at line 18, at the same time thread 1 will come back and able to lock as well. So this does not provide mutual exclusion as multiple threads will be able to lock and enter into critical section.

b) (2 points) This lock implementation is also not fair. Explain why this lock implementation is not fair.

Ans – It is not fair because this is pure spin lock and it doesn't guarantee that every thread that tries to acquire the lock will get access to it so it is completely possible for one thread to try and acquire the lock, spin on while loop, where another thread gets it , then another thread gets it and so on. There has to be some kind of a queue implementation for a lock to be a fair implementation of a lock so that every thread that tries to get that lock at least has a chance to get it.

**Q6. (5 points) Here are some tasks, including their arrival time and how much time they take to complete:**

A: Arrives at 0, takes 4

B: Arrives at 2, takes 1

C: Arrives at 4, takes 2

In this situation, time starts at 0 and the quantum length is 2. Compare the first in and first out and round robin scheduling policies on this workload. Be sure to include
- A summary of when the tasks first run and when they finish.

- The relative performance of the scheduling policies measured by response time and turnaround time.

- The general behaviour of the policies on this workload (e.g., when or how often do tasks get switched off?)

Ans – FIFO does a good job of turnaround time but does a terrible job or response time because task A is taking a long time and the other two tasks are arriving before task A finishes. FIFO is not a preemptive scheduling policy so it is just starting task A, then B and then C. The Round Robin scheduling policy has quantum length which is very small enough that task A gets switched off by running from 0 to 2 and at which point task B

arrives and start running. The response time is closer to zero where turnaround time is slightly higher.

- The general behaviour policies is FIFO is not preemptive and Round Robin is preemptive. The task will get switched off in FIFO when they finish and with round robin it gets switched of every 2 units of time

**Q7. 1 point (bonus) In "Technology Connections", Alec is describing the differences between Sony's Betamax and Betacam (two different ancient technologies for recording video onto a magnetic tape).**

Why because Betacan uses compressed time division multiplexing.

Here's a brief transcript of what Alec said about Betacam

-Why, because Betacam uses compressed time division multiplexing, silly!

Yeah, here's where things get a little bit nuts. The head that records the color information is actually recording two color signals at the same time. Well, actually, no, not at the same time.

Time-division multiplexing is a very old technique in which you essentially just keep switching between multiple signals so they can share a single channel. In this case, we only have two signals: Pb and Pr, so that head records the Pb signal for a teeny tiny bit, then the Pr signal for a teeny tiny bit.

then back to Pb.

then to Pr.

then to Pb,

then to Pr.

then to Pb.

and after enough of that an ice cold PBR with a PB&J is well deserved.

My (me. Franklin, weirdo brain thought: "Wow! That sounds like a lot like what happens in an operating system!" Can you make the same (technology) connection I did?

Ans – He's talking about switching from one signal to the other back and forth. So it is explaining about context switching and scheduling and concurrency.

//----------

RAID, Exfat, segmentation fault (why?)

What to do when file system got damaged

What exfat file check?

//----------

## Week (Feb 12-16)

## Drive Scheduling:

**Shortest seek time first** : One early disk scheduling approach is known as shortest-seek-time-first (SSTF) (also called shortest-seek-first or SSF). SSTF orders the queue of I/O requests by track, picking requests on the nearest track to complete first. For example, assuming the current position of the head is over the inner track, and we have requests for sectors 21 (middle track) and 2 (outer track), we would then issue the request to 21 first, wait for it to complete, and then issue the request to 2. But there is only 1 problem "**STARVATION**". Imagine in our example above if there were a steady stream of requests to the inner track, where the head currently is positioned. Requests to any other tracks would then be ignored completely by a pure SSTF approach.

**Elevator (aka SCAN or C-SCAN)**

The answer to this query was developed some time ago (see [CKR72] for example), and is relatively straightforward. The algorithm, originally called **SCAN**, simply moves back and forth across the disk servicing requests in order across the tracks. Let's call a single pass across the disk (from outer to inner tracks, or inner to outer) a *sweep*. Thus, if a request comes for a block on a track that has already been serviced on this sweep of the disk, it is not handled immediately, but rather queued until the next sweep (in the other direction).

SCAN has a number of variants, all of which do about the same thing. For example, Coffman et al. introduced **F-SCAN**, which freezes the queue to be serviced when it is doing a sweep [CKR72]; this action places requests that come in during the sweep into a queue to be serviced later. Doing so avoids starvation of far-away requests, by delaying the servicing of late-arriving (but nearer by) requests.

**C-SCAN** is another common variant, short for **Circular SCAN**. Instead of sweeping in both directions across the disk, the algorithm only sweeps from outer-to-inner, and then resets at the outer track to begin again. Doing so is a bit more fair to inner and outer tracks, as pure back-and-forth SCAN favors the middle tracks, i.e., after servicing the outer track, SCAN passes through the middle twice before coming back to the outer track again.

For reasons that should now be clear, the SCAN algorithm (and its cousins) is sometimes referred to as the **elevator** algorithm, because it behaves like an elevator which is either going up or down and not just servicing requests to floors based on which floor is closer. Imagine how annoying it would be if you were going down from floor 10 to 1, and somebody got on at 3 and pressed 4, and the elevator went up to 4 because it was "closer" than 1! As you can see, the elevator algorithm, when used in real life, prevents fights from taking place on elevators. In disks, it just prevents starvation.

**Week March 4-8**

1. **RAID Level 0 (Striping)**:
   - In RAID Level 0, data is spread across multiple disks in a process called striping.
   - It offers increased performance because data is distributed evenly across all disks, allowing for parallel access.
   - However, it does not provide redundancy or fault tolerance. If one disk fails, all data is lost.
   - RAID 0 is typically used in situations where performance is crucial, such as for video editing or gaming, but data redundancy is not a concern.

## Chunk size

1. **Effect on Performance**:
   - Chunk size significantly affects the performance of the RAID array.
   - A smaller chunk size means that individual files will be spread across more disks, increasing the parallelism of reads and writes for a single file. This parallelism can enhance performance, especially in situations where multiple users or processes are accessing different parts of the same file simultaneously.
   - Conversely, a larger chunk size reduces the intra-file parallelism because fewer disks are involved in storing each file. However, it relies on multiple concurrent requests to achieve high throughput, which can still be beneficial for certain workloads.
2. **Positioning Time**:
   - Positioning time refers to the time taken to locate and access data on the disks.
   - With smaller chunk sizes, accessing data may require seeking across multiple disks to retrieve all the relevant chunks of a file. This increases the overall positioning time because the maximum positioning time among all the disks is considered.
   - On the other hand, larger chunk sizes reduce the need for seeking across multiple disks for accessing a single file. If a file fits entirely within a chunk and is stored on a single disk, the positioning time incurred while accessing it will only be the positioning time of that single disk.

## Back To RAID-0 Analysis

Let us now evaluate the capacity, reliability, and performance of striping. From the perspective of capacity, it is perfect: given $N$ disks each of size $B$ blocks, striping delivers $N \cdot B$ blocks of useful capacity. From the standpoint of reliability, striping is also perfect, but in the bad way: any disk failure will lead to data loss. Finally, performance is excellent: all disks are utilized, often in parallel, to service user I/O requests.

## Evaluating RAID Performance

In analyzing RAID performance, one can consider two different performance metrics. The first is *single-request latency*. Understanding the latency of a single I/O request to a RAID is useful as it reveals how much parallelism can exist during a single logical I/O operation. The second is *steady-state throughput* of the RAID, i.e., the total bandwidth of many concurrent requests. Because RAIDs are often used in high-performance environments, the steady-state bandwidth is critical, and thus will be the main focus of our analyses.

As you can tell, sequential and random workloads will result in widely different performance characteristics from a disk. With sequential access, a disk operates in its most efficient mode, spending little time seeking and waiting for rotation and most of its time transferring data. With random access, just the opposite is true: most time is spent seeking and waiting for rotation and relatively little time is spent transferring data. To capture this difference in our analysis, we will assume that a disk can transfer data at $S$ MB/s under a sequential workload, and $R$ MB/s when under a random workload. In general, $S$ is much greater than $R$ (i.e., $S \gg R$).

**//----------**

**Summer 2023 – Franklin final**

**Q1.) (4 points) One of the settings that we can change in the multi-level feedback queue (MLFQ) scheduling policy was S. The S setting was what we used to decide how often all tasks in the system should be bumped back up to the highest priority.**

**What kind of behaviour could we observe if S is too small (e.g., it's smaller than a quantum length) or too larger (e.g., it's on the order of hours)?**

**Compare the behaviour of MLFQ with an S that's too small or too large to the other scheduling policies like shortest time to completion first (2 points) and round robin (2 points).**

**You're not being asked to evaluate the policies on a specific workload, but you should compare the general behaviour of these policies to the specific behaviour**

**of MLFQ with these two extreme settings for S. You're welcome to provide a sample workload, if you want, but you're not required to.**

- MLFQ with S(small/large) compared to STCF:

MLFQ with a small S may exhibit higher overhead due to frequent priority boosting, which can result in high context switching. STCF prioritizes tasks based on their expected completion time, leading to better throughput and reduced avg response time. However, STCF may suffer from starvation of long-running tasks if there are constantly short new tasks arriving.

MLFQ with large S results in tasks remaining at lower priority levels for extended periods, leading to poor responsiveness for tasks requiring immediate attention. STCF prioritizes tasks based on their expected completion time, which provides better response time compared to MLFQ.

- MLFQ with S(small/large) compared to Round Robin:

MLFQ with small S have higher context switching overhead compared to Round Robin due to frequent priority changes. Round Robin provides fairness by allocating equal time slices to each task. However, it leads to inefficiency if tasks have different execution time.

MLFQ with large S has ability to adjust priorities based on behavior, which provide advantage over Round Robin because it can react to what tasks need. However, it suffer from decreased responsiveness compared to Round Robin for tasks requiring immediate attention.

**Q2.)**

**a)**

2. I've got this great idea for a website: I want to collect favicons from websites around the web. I'm going to call it "Favicollector"™ (please don't steal my idea).

There are millions and millions of websites, so I am expecting to have millions and millions of files. But favicons are tiny (let's say they're all less than 4KB), so I'm going to have millions and millions of *really teeny tiny* files.

(a) [6 points] I wrote a spider to collect all the favicons, and it's working for about 100 web sites that I've been testing with. I'm getting ready to start running this on the entire web, but I'm realizing I should probably get a new server to run this on (I've been using my laptop for testing).

I'm expecting the total overall size of the favicons to be reasonably large because I'm downloading so many of them, maybe on the order of one terabyte. The spider is going to take days (or weeks) to finish once I start it — there are millions and millions of websites (maybe even billions!) and I'm trying to be polite to web servers and rate-limit my spider to maybe 200 web sites per minute.

Let's do some math:

$$200\frac{\text{web sites}}{\text{minute}} \times 60\frac{\text{minutes}}{\text{hour}} \times 24\frac{\text{hours}}{\text{day}} \times 30\frac{\text{days}}{\text{month}} = 8,640,000 \text{ web sites}$$

OK, it's going to take months (and months and months and months) to run this spider. I *really* don't want to run it again.

But I'm also very cheap. Yeah, a 1TB disk is as cheap as $50, but I'm *really* cheap. I found a bunch of these 100GB disks literally sitting in a dumpster and they were **free**!

So I'm going to need to use RAID. Of RAID-0, RAID-1, or RAID-4/5, which RAID-level would you suggest I use? **Pick one** RAID level and explain why I should use it (2 points). For the remaining RAID levels, explain why I shouldn't (or can't) use them (2 points each).

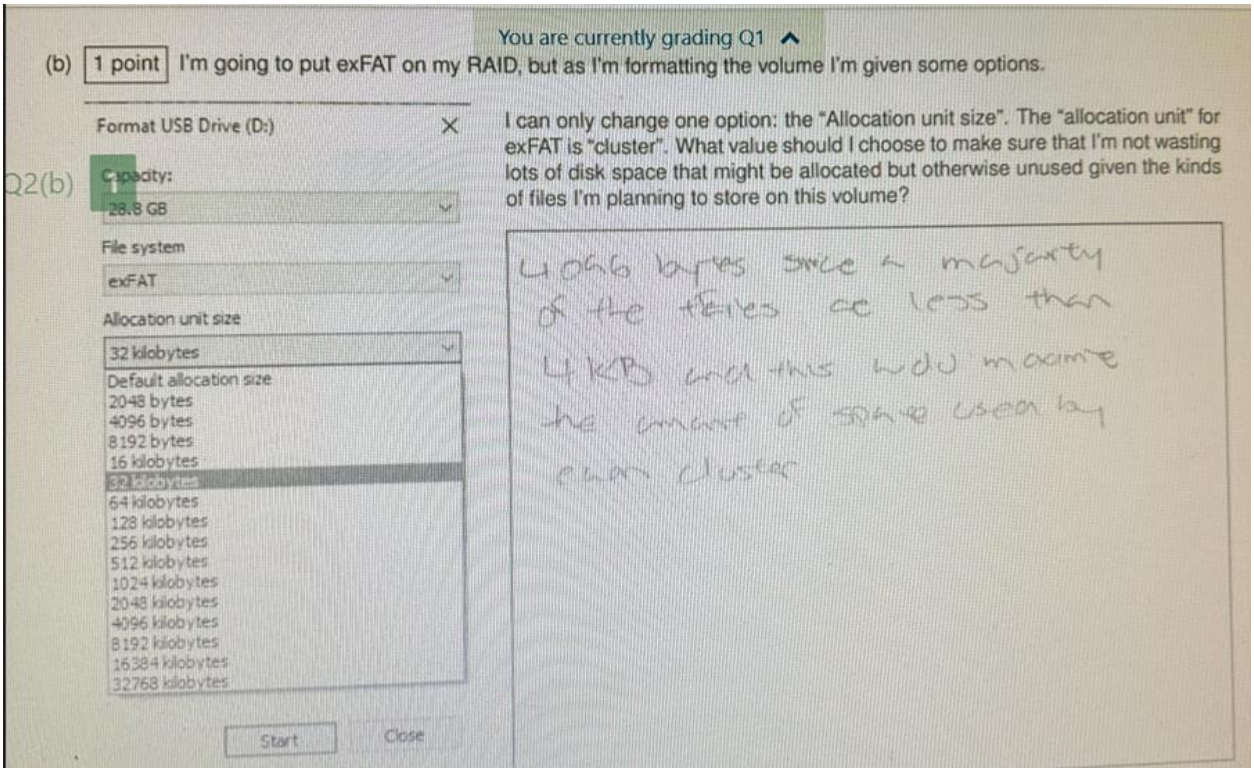
**Ans)** RAID-5 provides a good balance between redundancy and cost. It requires only one additional disk for parity, and since there are large number of relatively small disks available for free, RAID-5 allows to maximize storage capacity without significant additional cost.

RAID-5 provides fault tolerance by distributing the parity information across all disks in the array. In the event of single disk failure, data can be reconstructed using the parity information stored on remaining disks. It offers good performance as data can be read from multiple disks simultaneously. The write operation will be slightly slower due to parity calculations, but considering the small size of favicon files, it should be fine.

RAID-0 offers no redundancy or fault tolerance. It simply stripes data across multiple disks to improve performance and capacity. It increases the risk of data loss since the failure of any disk in array would result in the loss of all data.

RAID-1 require more disks for redundancy or provide less usable storage capacity compared to RAID-5.

**b)**




---

**FAT32 Architecture**

**+----------------------------------------+**

**| Volume Header |**

**+----------------------------------------+**

**| Boot Sector | Reserved Sectors | FAT Area(s) |**

**+----------------------------------------+**

**| Root Directory (optional) |**

**+----------------------------------------+**

**| Data Area (Clusters) |**

**+----------------------------------------+**

a.) **Data Structures Accessible/Modified by the FAT32 Implementation**:

- **Boot Sector**: The code snippet involves opening and closing files, which would typically require accessing information from the boot sector, such as the file system type, cluster size, etc.
- **File Allocation Table (FAT)**: Writing to a file involves updating the FAT entries to allocate clusters for storing the file's data. The close operation may involve updating FAT entries to mark clusters as free if the file is being closed and deleted.
- **Directory Entries**: The close operation might involve updating directory entries to reflect changes in the file size, modification time, etc., after the file has been written to.

---

**Q3.) I (me, Franklin) wrote some code that supports writing to exFAT-formatted file systems. There are two thread A and B running concurrently (in a single process) that use my code, and both of them are trying to run the following code on different files in the same volume/file system.**

```
void *thread_write_file(void *arg){
        char *filename = (char *) arg;
        int fd = franklin_exfat_open (filename, O_WRONLY); //open for write
        for (int i = 0; i < 100000; i++){
                franklin_exfat_write(fd, “hihihihihihi”, 14)// this will write 1.4MB
        }
        Franklin_exfat_close(fd);
}
```

**Let’s say that A’s filename is “A.txt” and B’s filename is “B.txt”. The sector size is 512bytes and the cluster size is 1 sector.**

**I suddenly realize that it never occurred to me that this code might be run concurrently, and uh, I never added any locks to any of the data structures in my implementation of exFAT (oops).**

**a.) Ignoring concurrency and threads: what data structures in the exFAT file system should be accessed or modified by my exFAT implementation?**

**For each of i, ii, iii below: If the function would not access or modify any exFAT data structures, then write "none".**

**i.) (1 point) What data structures in exFAT should franklin_exfat_close access or modify (if any)?**

- FAT (File allocation table) data structure is being accessed or modified to update the FAT entries to reflect changes in file allocation, such as deleting the file or updating its size.

i.) **FAT**: `franklin_exfat_close` may access or modify FAT entries to update file allocation information, especially if the file is being deleted or its size is changing.

**ii.) (2 point) What data structures in exFAT should franklin_exfat_open access or modify (if any)?**

- Boot Sector, File Allocation Table (FAT) and Directory Entries are the data structures that are accessed or modified.

Boot Sector – The function may need to access the boot sector to retrieve essential info about FS, such as cluster size.

FAT – FAT entries are consulted to locate the clusters allocated to the file. If a file is being created, the open operation might modify the FAT entries to allocate clusters of file's data.

Directory Entries – The open operation involves locating the directory entry specified file, which may require accessing entries in root directory and subdirectories.

ii.) **Boot Sector, FAT, Directory Entries**: `franklin_exfat_open` may access the boot sector, FAT entries, and directory entries to retrieve information about the file system and locate the specified file. If the file is being created or truncated, it may also modify the FAT entries to allocate clusters for the file's data.

### iii.) (3 point) What data structures in exFAT should franklin_exfat_write access or modify (if any)?

- FAT, Directory entries, Data clusters are data structures which may be accessed or modified.

FAT – The write operation involves updating FAT entries to allocate additional clusters if file's size exceeds the currently allocated clusters, or to mark clusters free if data is being deleted.

Directory Entries – If the file's size changes because of write operation, the directory entry for file may need to be updated to reflect new size.

Data Clusters: The data clusters containing the file's content will be modified to write new data.

iii.) **FAT, Directory Entries, Data Clusters**: `franklin_exfat_write` may access and modify the FAT entries to allocate or release clusters for the file's data, update directory entries to reflect changes in file size and modification time, and modify the data clusters to write the new data.

**b) We know that writing a single sector is an atomic operation (the drive will either completely write the sector or it will not write the sector at all, we will not have partial sector writes). My code updates exFAT data structures in the volume by reading from the file system, making changes in memory, then writing the structure back out (not an atomic operation).**

**i.) This part of the question has two options, you must pick (and circle) one of a or b. If you choose to answer a, you may earn up to two bonus points if your**

**response is adequate and accurate. If you instead choose to answer b, you will not lose any points. In either case, your answer to i will be user to answer ii.**

**a) (2 points (bonus)) Describe a sequence of operations between the two threads A and B that would result in an inconsistent file system. RememberL I, uh, forgot to have any locks in my exFAT code at all (oops), and the thread might be interrupted in my code (the implementation of, for example, frankln_exfat_write); or**

- Sequence might be:

Thread A and Thread B both attempt to write to their respective files (A.txt and b.txt)

Thread A successfully opens its file (A.txt) and starts writing data to it.

Before Thread A completes its write operation, it gets interrupted by the scheduler.

Thread B successfully opens its file (B.txt) and starts writing data to it.

Thread B completes its write operation and closes its file.

Thread A resumes execution and completes its write operation, updating the FAT entries and directory entries for A.txt

Thread A closes its file.

- At this point the FS would be inconsistent because the changes made by Thread B have been written to the disk, but the changes made by Thread A have not. This could lead to discrepancies in the FAT entries, directory entries, or even data clusters.

**b) (0 points) Describe an inconsistent exFAT file system with these two files.**

**ii.) Describe what the person running the computer might see, if they were to look at these two files or the volume in their file explorer, based on the**

**inconsistent file system you describe above. You must provide an answer to part a or b above to earn points for parts A, B, C below.**

**A) (1 point) What would the contents of the file look like?**

- The contents of A.txt might reflect the data written by Thread A up to the point of interruption and any data after Thread A intends to write will not be present. The contents of B.txt would be consistent with the data written as it completed its write operation before any interruptions occurred.

- So, the person viewing the computer may observe inconsistent or corrupted file contents in their file explorer.

**B) (1 point) How would the size of the file change (if at all)?**

- In this scenario, since Thread A is interrupted before its completion, any changes, or updates to the size of A.txt are not reflected on the disk. The size of B.txt would change, reflecting the data successfully written by Thread B.

**C) (1 point) How would the remaining free space on the volume change (if at all)?**

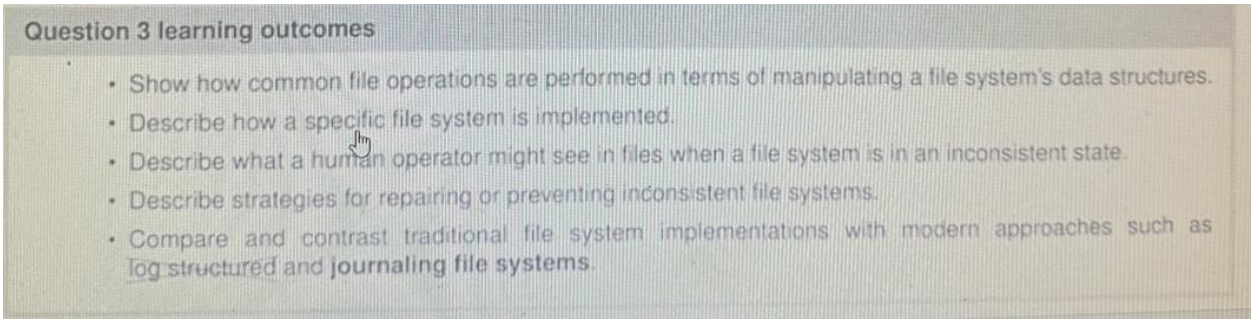
- The remaining free space is unaffected by Thread A's incomplete operation since it didn't allocate any clusters for A.txt due to interruption.

- The remaining free space on the volume decreases by the amount of data successfully written by Thread B, as clusters are allocated for B.txt.

**iii.) (1 point) If exFAT was a journaling file system, would the problem you identified above be resolved? Why or why not?**

- The problem identified wouldn't be resolved with journaling. Journaling might log the changes made by Thread A before they are written to the disk. However, it

wouldn't prevent Thread B from completing its operation and modifying the file system before Thread A's changes are applied from the journal.




**Q4.) Here's a super good program I wrote called name-printer.c:**

```
#include <stdio.h>

#include <stdlib.h>

int main(void){

        char *name = NULL;

        printf("Enter your name: ");

        scanf("%s", name);

        printf("Hi %s\n", name);

        return EXIT_SUCCESS;

}
```

**My code compiled! Even with all the warnings turned on! So I ran it:**

**$> ./name-printer**

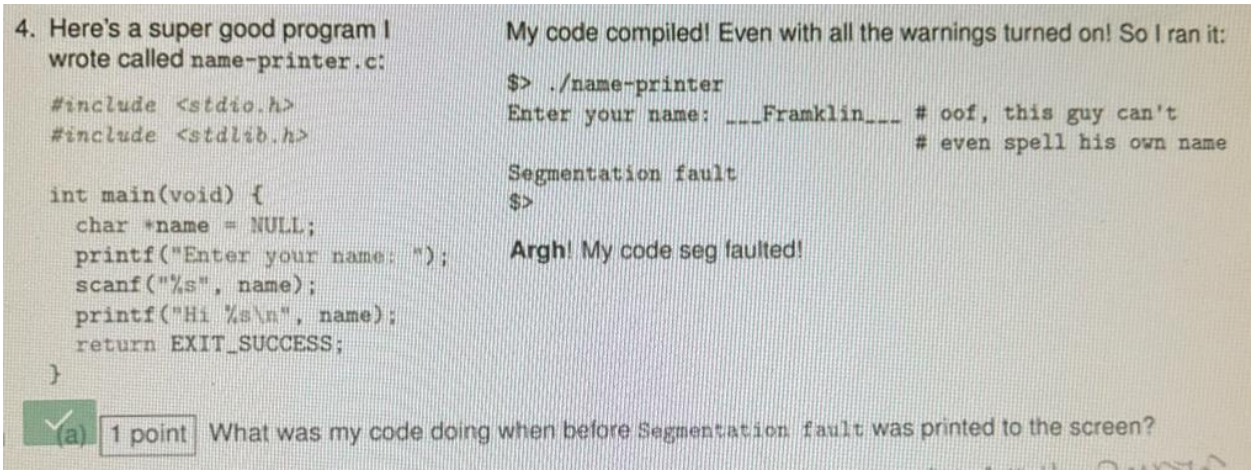
**Enter your name: ___Franklin___ #oof, this guy can't**

**# even spell his own name**

**Segmentation fault**

**$>**

**Argh! My code seg faulted!**



4. Here's a super good program I wrote called `name-printer.c`:

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  char *name = NULL;
  printf("Enter your name: ");
  scanf("%s", name);
  printf("Hi %s\n", name);
  return EXIT_SUCCESS;
}
```

My code compiled! Even with all the warnings turned on! So I ran it:

```
$> ./name-printer
Enter your name: ___Framklin___ # oof, this guy can't
                                # even spell his own name

Segmentation fault
$>
```

**Argh**! My code seg faulted!

(a) 1 point What was my code doing when before `Segmentation fault` was printed to the screen?

**a) (1 point) What was my code doing when before Segmentation fault was printed to the screen?**

- The code was trying to initialize a pointer 'name' of type char* to NULL which means it points to nothing. Then code attempts to read the user input using scanf and trying to write to the memory location pointed to by name which is NULL.

**b) (2 points) What was my hardware doing before Segmentation fault was printed to the screen?**

- Hardware was executing the instructions provided by CPU. It was in the process of executing the scanf function call to read user input and attempt to write to the memory location pointed to name variable.

**c) (3 points) What was my shell doing before and after segmentation fault wasd printed to the screen?**

- Before the segmentation fault was printed to the screen, shell was in a state of waiting for the execution of program. When the run command has entered by

user, the shell initiated the execution. During execution, shell was managing input/output streams between program and user. It displayed the prompt ‘Enter your name: ’ and then waited for the user to input data.

- Afte the segmentation fault occurred was printed to the screen, shell handled aftermath of the fault. It recognized that program terminated unexpectedly due to seg fault and then returned to original state, awaiting further command or input from user.

**d) (3 points) What was my OS doing before Segmentation fault was printed to the screen?**

- Before the seg fault was printed to the screen, OS initiated execution of program into shell and created new process to run in. OS allocated memory of program to execute including Code, Heap & Stack. When the code tries to allocate memory for name variable and try to write to a NULL pointer resulted in seg fault. OS then handled I/O operations such as displaying the prompt and reading user input. When a program encountered seg fault due to attempt to access invalid memory, OS detected an exceptional condition. It terminated the process and informed the user about the error.

- After the seg fault was printed to the screen, OS continued to manage the system, returning control to the shell for further user interaction. It may have logged info about the seg fault in system logs for diagnostic purposes.

**Q5.) There is no file system for memory, but gosh, it sure seems like there are a lot of “file-system-like-things” in virtual memory related things.**

**Compare VSFS/ext2 to the virtual memory implementation of page directories/multi-level page tables. Describe the specific data structures or part of the idea in each that is the same. Provide three (and only three) examples of how these two things are similar to each other.**

**Each example is worth a total of two points for describing the data structure or concept that is related in each of VSFS/ext2 (1point) and page directories (1 point).**

**a) (2 points) Example 1**

VSFS – The concept of an inode table is used to store metadata about files, such as permissions, timestamps, and pointers to data blocks.

Page Directories – Similarly, page directories serves as hierarchical structures to organize and manage page tables, which in turn contain metadata about memory pages, such as permissions & physical page frame numbers.

**b) (2 points) Exmaple 2**

VSFS – Data blocks are used to store actual file data. These data blocks are referenced by inodes and ar typically organized in a linked list or tree structure.

Page Directories – Page tables contain entries that map virtual memory pages to physical memory. These entries point to the location of the data in physical memory, similarly to how data blocks are referenced in FS.

**c) (2 points) Exmaple 3**

VSFS – Superblock is a key data structure that contains metadata about FS, such as total number of inodes and data blocks, block size.

Page Directories – There is a root page directory that serves as entry point for address translation. This page directory contains metadata about overall memory layout, such as pointers to lower-level page directories.

**Q6.) Base and bound was … fine, but we wound up having some pretty severe limitations and problems:**

1. **We could only have a fixed number of concurrently running: #ofprocessess= sizeof(physical memory)/(sizeof(virtual address space))**
2. **We could not have virtual address spaces larger than physical memory.**
3. **We wasted a lot of memory with internal fragmentation.**

**a.) Explain how segmentation solved each of these problems.**

**i.) (1 point) Problem 1: Only able to have a fixed number of concurrent processes.**

- Segmentation allows a process to be divided with its own base and bound pair per logical segment of the address space. It allows dynamic memory allocation, enabling processes to vary in size and structure. This flexibility permits a larger number of concurrently running processes compared to fixed size and number imposed by base and bound mechanisms.

**ii.) (1 point) Problem 2: Virtual address spaces must be smaller than physical memory.**

**-** Segmentation, particularly when combined with paging, allows for virtual address spaces larger than physical memory. Paging breaks down segments into smaller units, enabling efficient memory management and transparent swapping of data between main memory and secondary storage.

**iii.) (1 point) Problem 3: Wasting space with internal fragmentation.**

- Segmentation reduces internal fragmentation by providing variable-sized segments that adapt to the actual size requirements of processes. This means that segments can be sized precisely to match the requirement of the data or code they contain. Additionally, dynamic memory allocation and segmentation with paging further optimize memory usage by allowing for efficient allocation and utilization of memory resources.

**b.) Segmentation has its own problems. List and describe two examples of problems with segmentation.**

**i.) (1 point) Problem 1: External Fragmentation**

- Because segments are variable-sized, it causes free memory to be fragmented into irregularly sized pieces. This fragmentation makes it challenging to satisfy memory allocation requests, even if sufficient total free memory exists in the system.

**ii.) (1 point) Problem 2: Inflexibility for Sparse Address Spaces**

- Segmentation lacks flexibility to effectively support fully generalized, sparse address spaces. For instance, if a large but sparsley-used heap is allocated within a single logical segment, the entire heap must reside in memory for access. Segmentation's design may not align with the actual usage pattern of the address space, leading to inefficiencies.

**Q7.) We've got a 16KB virtual address space and we're trying to look up a virtual address at location 0x1234 (that's 0b0001 0010 0011 0100 (0b means 'this is a binary number'))**

**Choose the one best answer for each of the following below. Selecting multiple options will be assessed as 0.**

**a.) (1 point) Let's assume we're using base and bounds, there's 64KB of physical memory in this machine, we're allocating 4 full address spaces, and this process has been placed into physical memory starting at 32KB. What physical address does this virtual address translate to?**

a) **0x1234 (0x0000 [00KB] + 0x1234)**
b) **0x5234 (0x4000 [16KB] + 0x1234)**
c) **0x9234 (0x8000 [32KB] + 0x1234)**
d) **0x11234 (0x10000 [64KB] + 0x1234)**
e) **Segmentation fault**

**b.) (1 point) Let's assume we're using segmentation with an explicit approach. We've got 4 segments in our virtual address space: code, heap, unnamed stack (contiguous, in that order). Which segment does this address refer to?**

**a) Code**

**b) Heap**

**c) That weird unnamed part in the middle between heap and stack**

**d) Stack**

**E.) Segmentation fault**

**c.) (1 point) Let's assume we're using linear paging. We've got 1 KB pages in our virtual address space. Which page (in virtual memory) does the address above refer to?**

**a) Page 0x0 / 0b0000**

**b) Page 0x1 / 0b0001**

**c) Page 0x4 / 0b0100**

**d) Page 0x1234 / 0b1100 1010 1111 1100**

**e) Segmentation fault**

**d.) (2 points) We've got 64KB of physical memory (max address is 0x10000) and 1KB page frames (so 64 or 0x40 page frames). Translate the virtual address to it's corresponding physical address using the linear page table below.**

| VPN | PFN |
|---|---|
| 0x00 | 0x08 |
| 0x01 | 0x0A |
| 0x02 | 0x0C |
| 0x03 | Not allocated |
| 0x04 | Not allocated |
| 0x05 | Not allocated |
| 0x06 | Not allocated |
| 0x07 | Not allocated |
| 0x08 | Not allocated |
| 0x09 | Not allocated |
| 0x0A | Not allocated |
| 0x0B | Not allocated |
| 0x0C | Not allocated |
| 0x0D | 0x11 |
| 0x0E | 0x22 |
| 0x0F | 0x39 |

A. `0x0A × 1024 + 0x0234 = 0x02A34`
B. `0x08 × 1024 + 0x1234 = 0x03234`
C. `0x40 × 65536 + 0x0234 = 0x4000234`
D. `0x0F × 1024 + 0x1234 = 0x04E34`
E. `Segmentation fault`

You can earn part marks (up to 1) if you choose the incorrect option, but show your work: