QUIZ

Started on	Wednesday, 2 August 2023, 12:23 PM
State	Finished
Completed on	Wednesday, 2 August 2023, 12:28 PM
Time taken	4 mins 43 secs
Grade	**0.00** out of 20.00 (**0**%)

Question 1

Not answered

Marked out of 1.00

It is usually considered as unwise to remove a USB key without ejecting it before. Check all the true claims regarding the handling of USB keys by Linux.

- ☐ a. Memory Management Unit is in charge of ejecting USB keys
- ☐ b. Data written to USB keys are really written to USB keys only after the key has been ejected
- ☐ c. For performance reason, data written to USB keys are first written to RAM
- ☐ d. USB keys are handled by the Linux kernel I/O subsystem
- ☐ e. A swap out operation must be performed for data to be written to USB keys

Your answer is incorrect.

The correct answers are:
USB keys are handled by the Linux kernel I/O subsystem,

For performance reason, data written to USB keys are first written to RAM

Question 2

Not answered

Marked out of 1.00

Check all the true claims below regarding the sub-code of "grep" provided below. This code is taken from the Apple open-source code of grep.

```
/*
 * Reads searching patterns from a file and adds them with add_pattern().
 */
static void
read_patterns(const char *fn)
{
        struct stat st;
        FILE *f;
        char *line;
        size_t len;

        if ((f = fopen(fn, "r")) == NULL)
                err(2, "%s", fn);
        if ((fstat(fileno(f), &st) == -1) || (S_ISDIR(st.st_mode))) {
                fclose(f);
                return;
        }
        while ((line = fgetln(f, &len)) != NULL)
                add_pattern(line, line[0] == '\n' ? 0 : len);
        if (ferror(f))
                err(2, "%s", fn);
        fclose(f);
}
```

- ☐ a. Each line of the file given as input is read as a new search pattern
- ☐ b. This code opens a file in read and write mode
- ☐ c. This code returns in the second "if" if the file given as parameter is a directory
- ☐ d. This code takes as parameter a path to a file

Your answer is incorrect.

The correct answers are:
This code returns in the second "if" if the file given as parameter is a directory,

This code takes as parameter a path to a file,

Each line of the file given as input is read as a new search pattern

Question 3

Not answered

Marked out of 1.00

Check all the true claims about page replacement algorithms.

☐ a. Page replacement algorithm commonly monitor how pages are read and write to decide of the page to be evicted

☐ b. Page replace algorithms are always triggered when the Memory Management Unit generates a trap

☐ c. A "segmentation fault" is the result of running page replacement algorithms

☐ d. Page replacement algorithms are systematically used when malloc() is called

☐ e. Memory Management Units trigger a page replacement algorithm each time a page cannot be accessed

Your answer is incorrect.

The correct answer is:
Page replacement algorithm commonly monitor how pages are read and write to decide of the page to be evicted

Question 4

Not answered

Marked out of 1.00

Select all correct claims below. These claims concern the CPU kernel / supervisor mode and the administrator user (root, admin) of an Operating System.

☐ a. An administrator can execute any software in kernel mode

☐ b. The compilation of a new kernel must be done in kernel mode

☐ c. An administrator can select which mode (kernel, user) of the processor must be used to execute each software installed in a system

☐ d. An administrator can install a new kernel module that will execute in kernel mode

Your answer is incorrect.

The correct answer is:
An administrator can install a new kernel module that will execute in kernel mode

Question **5**

Not answered

Marked out of 1.00

Check all the true claims about the following code (includes have been removed)

```
int main(int argc, char*argv[]) {

  int out_fd;
  int written;

  if (argc < 3) {
    printf("usage: writeToFile <file> <text>\n");
    exit(0);
  }

  char *file = argv[1];

  if ( (out_fd = open(file, O_WRONLY | O_SYNC | O_CREAT)) < 0) {
    printf("Could not open the file %s because:%d\n", file, errno);
    exit(1); // bash assumes a non zero code means an error
  }


  char * toBeWritten = argv[2];

  written = write(out_fd, toBeWritten, strlen(toBeWritten) );

  if (written < strlen(toBeWritten)) {
    printf("Write in file %s failed\n", file);
    exit(1); // bash assumes a non zero code means an error
  }

  if (close(out_fd) < 0) {
    printf("Could not close the file %s\n", file);
  }

  printf("Text %s successfully written to %s\n", toBeWritten, file);
  exit(0);
}
```

- ☐ a. This program can crash because "toBeWritten" is not allocated
- ☐ b. The last printf crashes because toBeWritten does not terminate with '\0'
- ☐ c. The destination file is created if it does not exist
- ☐ d. If this program were to omit the call to close(p), then the file would remain open after the program terminates
- ☐ e. The program does not accept more than 2 arguments
- ☐ f. The program exits if the destination file given as parameter does not exist

Your answer is incorrect.

The correct answer is: The destination file is created if it does not exist

Question **6**

Not answered

Marked out of 1.00

Check all the true claims about the following manual page.

NAME
read - read from a file descriptor

SYNOPSIS
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION
read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.

On files that support seeking, the read operation commences at the file offset, and the file offset is incremented by the number of bytes read. If the file offset is at or past the end of file, no bytes are read, and read() returns zero.

If count is zero, read() may detect the errors described below. In the absence of any errors, or if read() does not check for errors, a read() with a count of 0 returns zero and has no other effects.

According to POSIX.1, if count is greater than SSIZE_MAX, the result is implementation-defined; see NOTES for the upper limit on Linux.

RETURN VALUE
On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because read() was interrupted by a signal. See also NOTES.

On error, -1 is returned, and errno is set appropriately. In this case, it is left unspecified whether the file position (if any) changes.

ERRORS
EAGAIN The file descriptor fd refers to a file other than a socket and has been marked nonblocking (O_NONBLOCK), and the read would block. See open(2) for further details on the O_NONBLOCK flag.

EAGAIN or EWOULDBLOCK
The file descriptor fd refers to a socket and has been marked nonblocking (O_NONBLOCK), and the read would block. POSIX.1-2001 allows either error to be returned for this case, and does not require these constants to have the same value, so a portable application should check for both possibilities.

EBADF fd is not a valid file descriptor or is not open for reading.

EFAULT buf is outside your accessible address space.

EINTR The call was interrupted by a signal before any data was read; see signal(7).

EINVAL fd is attached to an object which is unsuitable for reading; or the file was opened with the O_DIRECT flag, and either the address specified in buf, the value specified in count, or the file offset is not suitably aligned.

EINVAL fd was created via a call to timerfd_create(2) and the wrong size buffer was given to read(); see timerfd_create(2) for further information.

EIO I/O error. This will happen for example when the process is in a background process group, tries to read from its controlling terminal, and either it is ignoring or blocking SIGTTIN or its process group is orphaned. It may also occur when there is a low-level I/O error while reading from a disk or tape. A further possible cause of EIO on networked filesystems is when an advisory lock had been taken out on the file descriptor and this lock has been lost. See the Lost locks section of fcntl(2) for further details.

EISDIR fd refers to a directory.

Other errors may occur, depending on the object connected to fd.

CONFORMING TO
SVr4, 4.3BSD, POSIX.1-2001.

BUGS
According to POSIX.1-2008/SUSv4 Section XSI 2.9.7 ("Thread Interactions with Regular File Operations"):

All of the following functions shall be atomic with respect to each other in the effects specified in POSIX.1-2008 when they operate on regular files or symbolic links: ...

Among the APIs subsequently listed are read() and readv(2). And among the effects that should be atomic across threads (and processes) are updates of the file offset. However, on Linux before version 3.14, this was not the case: if two processes that share an open file description (see open(2)) perform a read() (or readv(2)) at the same time, then the I/O operations were not atomic with respect updating the file offset, with the result that the reads in the two processes might (incorrectly) overlap in the blocks of data that they obtained. This problem was fixed in Linux 3.14.

- ☐ a. An error is returned if the input buffer "buf" is not part of the process address space
- ☐ b. read() ignores the current file offset to read data from a file
- ☐ c. With the latest Linux kernels, a programmer can safely open the same file in two different processes and perform concurrent calls to read(2).
- ☐ d. Receiving a signal while reading bytes with read() may trigger an error.
- ☐ e. Read takes as argument the number of bytes "count" to be read. If the effective number of bytes read is smaller than "count", then an error is returned

Your answer is incorrect.

The correct answers are:
With the latest Linux kernels, a programmer can safely open the same file in two different processes and perform concurrent calls to read(2).,

An error is returned if the input buffer "buf" is not part of the process address space,

Receiving a signal while reading bytes with read() may trigger an error.

Question **7**

Not answered

Marked out of 1.00

```
#include <p t h r e a d . h>
#include <s t d i o . h>
pthread_mutex_t m;
pthread_t a , b ;

void ∗ f ( void ∗param ) {
 while ( 1 ){
   pthread_mutex_lock(&m) ;
   p r i n t f ( " % s " , param ) ;
   pthread_mutex_unlock(&m) ;
   p t h r e a d _ y i e l d ( ) ;
 }
}

i n t main ( ) {
 pthread_mutex_init (&m, NULL ) ;
 p t h r e a d _ c r e a t e (&a , NULL, f , " Hello " ) ;
 p t h r e a d _ c r e a t e (&b , NULL, f , " World \ n " ) ;
 p t h r e a d _ j o i n ( a , NULL ) ;
 p t h r e a d _ j o i n ( b , NULL ) ;
}
```

☐ a. This program guarantees to print "Hello world" many times

☐ b. This program could crash because param is never allocated

☐ c. "p t h r e a d _ j o i n (b , NULL) ;" is not likely to be executed

☐ d. "pthread_t a , b ;" could be moved to the beginning of the main function

Your answer is incorrect.

The correct answers are:
"p t h r e a d _ j o i n (b , NULL) ;" is not likely to be executed,

"pthread_t a , b ;" could be moved to the beginning of the main function

Question **8**

Not answered

Marked out of 1.00

Does this program always provoke a segmentation fault? (includes have been removed)

```
int main(int argc, char ** argv) {
 char *name;
 name = (char *) (malloc (20 * sizeof (char)));
 name[22] = 'h';
}
```

○ a. Yes

○ b. No

Your answer is incorrect.

The correct answer is:
No

Question **9**

Not answered

Marked out of 1.00

When executing the following code, assuming it never fails, how many times is "OS is great!" printed? (includes have been removed)

```
--
i n t  main ( int nbOfArgs , char ** args ) {
 int i =0;
 pid_t ret ;
 for ( i =0; i <2; i ++){
   ret  =  f o r k ();
   if ( ret == -1) exit (-1);
 }
 p r i n t f  ( "OS is great!\n" );
 r e t u r n  1;
}
```

Answer: ✗

The correct answer is: 4

Question **10**

Not answered

Marked out of 1.00

Check all the true claims about brk() and malloc(). To help you, we provide the manul page of brk() and of malloc() below.

```
BRK(2)                          Linux Programmer's Manual                          BRK(2)

NAME
      brk, sbrk - change data segment size

SYNOPSIS
      #include <unistd.h>

      int brk(void *addr);

      void *sbrk(intptr_t increment);


DESCRIPTION
      brk()  and  sbrk()  change  the location of the program break, which defines the end of the process's data segment (i.e., the program break is the first location after the end of the uninitialized data segment).  Increasing the program break  has  the effect of allocating memory to the process; decreasing the break deallocates memory.

      brk()  sets  the end of the data segment to the value specified by addr, when that value is reasonable, the system has enough memory, and the process does not exceed its maximum data size (see setrlimit(2)).

      sbrk() increments the program's data space by increment bytes.  Calling sbrk() with an increment of 0 can be used to find the current location of the program break.

RETURN VALUE
      On success, brk() returns zero.  On error, -1 is returned, and errno is set to ENOMEM.

      On  success,  sbrk()  returns  the  previous program break.  (If the break was increased, then this value is a pointer to the start of the newly allocated memory).  On error, (void *) -1 is returned, and errno is set to ENOMEM.


COLOPHON
      This page is part of release 4.16 of the Linux man-pages project.  A description of the project, information about  reporting
      bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.
```

```
MALLOC(3)                          Linux Programmer's Manual                          MALLOC(3)

NAME
      malloc, free, calloc, realloc - allocate and free dynamic memory

SYNOPSIS
      #include <stdlib.h>

      void *malloc(size_t size);
```

- ☐ a. If sbrk() fails, a pointer to address -1 is returned
- ☐ b. malloc() is a syscall
- ☐ c. brk() can be used to disallocate memory
- ☐ d. This manual page of brk() is correct for Linux kernel 4.16
- ☐ e. brk() is a syscall

Your answer is incorrect.

The correct answers are:
This manual page of brk() is correct for Linux kernel 4.16,

brk() can be used to disallocate memory,

brk() is a syscall,

If sbrk() fails, a pointer to address -1 is returned

Question **11**

Not answered

Marked out of 1.00

Check the memory sections that can be allocated at run time after the process has been initialized.

--

- ☐ a. Heap
- ☐ b. Environment data
- ☐ c. Stack
- ☐ d. Text

Your answer is incorrect.

The correct answers are:
Stack,

Heap

Question **12**

Not answered

Marked out of 1.00

Check all the true claims about the functions shmat() and shmdt(). An excerpt of their manual page is given below.

NAME
shmat, shmdt - System V shared memory operations

SYNOPSIS
```
#include <sys/types.h>
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg);

int shmdt(const void *shmaddr);
```

DESCRIPTION
shmat()
shmat() attaches the System V shared memory segment identified by shmid to the address space of the calling process. The attaching address is specified by shmaddr with one of the following criteria:

* If shmaddr is NULL, the system chooses a suitable (unused) page-aligned address to attach the segment.

* If shmaddr isn't NULL and SHM_RND is specified in shmflg, the attach occurs at the address equal to shmaddr rounded down to the nearest multiple of SHMLBA.

* Otherwise, shmaddr must be a page-aligned address at which the attach occurs.

In addition to SHM_RND, the following flags may be specified in the shmflg bit-mask argument:

SHM_EXEC (Linux-specific; since Linux 2.6.9)
Allow the contents of the segment to be executed. The caller must have execute permission on the segment.

SHM_RDONLY
Attach the segment for read-only access. The process must have read permission for the segment. If this flag is not specified, the segment is attached for read and write access, and the process must have read and write permission for the segment. There is no notion of a write-only shared memory segment.

SHM_REMAP (Linux-specific)
This flag specifies that the mapping of the segment should replace any existing mapping in the range starting at shmaddr and continuing for the size of the segment. (Normally, an EINVAL error would result if a mapping already exists in this address range.) In this case, shmaddr must not be NULL.

The brk(2) value of the calling process is not altered by the attach. The segment will automatically be detached at process exit. The same segment may be attached as a read and as a read-write one, and more than once, in the process's address space.

A successful shmat() call updates the members of the shmid_ds structure (see shmctl(2)) associated with the shared memory segment as follows:

shm_atime is set to the current time.

shm_lpid is set to the process-ID of the calling process.

shm_nattch is incremented by one.

shmdt()
shmdt() detaches the shared memory segment located at the address specified by shmaddr from the address space of the calling process. The to-be-detached segment must be currently attached with shmaddr equal to the value returned by the attaching shmat() call.

On a successful shmdt() call, the system updates the members of the shmid_ds structure associated with the shared memory segment as follows:

shm_dtime is set to the current time.

shm_lpid is set to the process-ID of the calling process.

shm_nattch is decremented by one. If it becomes 0 and the segment is marked for deletion, the segment is deleted.

RETURN VALUE
On success, shmat() returns the address of the attached shared memory segment; on error, (void *) -1 is returned, and errno is set to indicate the cause of the error.

On success, shmdt() returns 0; on error -1 is returned, and errno is set to indicate the cause of the error.

ERRORS
When shmat() fails, errno is set to one of the following:

EACCES The calling process does not have the required permissions for the requested attach type, and does not have the CAP_IPC_OWNER capability in the user name-
space that governs its IPC namespace.

EIDRM shmid points to a removed identifier.

EINVAL Invalid shmid value, unaligned (i.e., not page-aligned and SHM_RND was not specified) or invalid shmaddr value, or can't attach segment at shmaddr, or
SHM_REMAP was specified and shmaddr was NULL.

ENOMEM Could not allocate memory for the descriptor or for the page tables.

When shmdt() fails, errno is set as follows:

EINVAL There is no shared memory segment attached at shmaddr; or, shmaddr is not aligned on a page boundary.

NOTES
After a fork(2), the child inherits the attached shared memory segments.

After an execve(2), all attached shared memory segments are detached from the process.

Upon _exit(2), all attached shared memory segments are detached from the process.

- ☐ a. shmat() attaches a shared memory segment at whatever address provided as argument (void * shmaddr)
- ☐ b. When a process that has performed a shmat() exits, all shared memory segments it has attached are automatically disallocated.
- ☐ c. After a fork, both the parent and child processes share the memory segments previously attached with shmat() by the parent process
- ☐ d. A shared memory segment detached with shmdet() is immediately disallocated by the OS
- ☐ e. If the "void * shmaddr" argument of shmat() is NULL, then an error is returned

Your answer is incorrect.

The correct answer is:
After a fork, both the parent and child processes share the memory segments previously attached with shmat() by the parent process

Question 13

Not answered

Marked out of 1.00

Check all the true claims about the execution of the C code provided below.

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


int funnyAllocation(char *buf, int b) {
   b = b+ 2;
   buf = (char *) ( malloc(sizeof(char) * 5));
   memcpy(buf, "hello", 6);
   return b;
}


int main( int argc, char*argv[] ) {
   int b = 8;
   char *buf = (char *) ( malloc(sizeof(char) * 5));
   int returned = funnyAllocation(buf, b);
   printf("The content of buf is: %s\n", buf);
}
```

- ☐ a. The memory allocated by the call to malloc() in main() is filled in funnyAllocation()
- ☐ b. The execution can provoke a memory allocation error
- ☐ c. At the end of this program execution, the value of b is 10
- ☐ d. hello is always printed in the standard output
- ☐ e. argv is not used

Your answer is incorrect.

The correct answers are:
argv is not used,

The execution can provoke a memory allocation error

Question 14

Not answered

Marked out of 1.00

What is the average waiting time of the following set of tasks scheduled with Shortest Job First?

Processes ; Arrival Time ; Duration

p1 ; 5 ; 3

p2 ; 2 ; 4

p3 ; 7 ; 2

p4 ; 6 ; 1

p5 ; 10; 1

Answer: ✕

The correct answer is: 1.2

Question **15**

Not answered

Marked out of 1.00

```
$ ls -l /dev/
...
crw-r-----   1 root kmem      1,     1 Jan 16 17:07 mem
crw-rw-rw-   1 root root      1,     8 Jan 16 17:07 random
crw-------   1 root root    249,     0 Jan 16 17:07 rtc0
...
```

- ☐ a. The real-time clock can be read by all users
- ☐ b. A member of the "root" group can set the real-time clock
- ☐ c. The "random" device can be read by all users
- ☐ d. The "random" device is a character device
- ☐ e. The "random" device can return a block of characters

Your answer is incorrect.

The correct answers are:
The "random" device is a character device,

The "random" device can be read by all users

Question **16**

Not answered

Marked out of 1.00

Check all the true claims about a C program.

- ☐ a. The program is in charge of dis-allocating resources since the Operating System cannot do it once the program has terminated.
- ☐ b. A C program cannot start another C program without using a system call.
- ☐ c. The execution of a C program involves the Operating System
- ☐ d. A C program currently executing a sub-function from its main cannot exit until this sub-function has returned to the main function.
- ☐ e. Calling the "exit()" function immediately terminates the C program that returns to the OS.

Your answer is incorrect.

The correct answers are:
The execution of a C program involves the Operating System,

A C program cannot start another C program without using a system call.

Question 17

Not answered

Marked out of 1.00

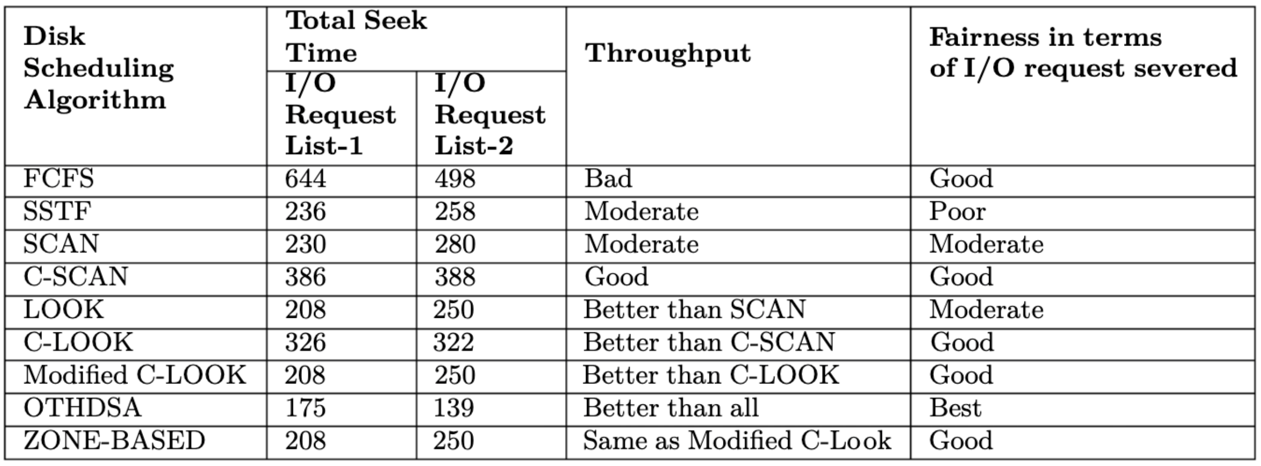
The paper entitled "Analysis of Traditional Hard Disk Scheduling Algorithms: A Review" was published in 2021 by Bamboat et al. It compares the fairness and seek time for different datasets of transactions on disks., and for different disk scheduling algorithms (e.g., SSTF, C-SCAN, etc.).

Check all the true claims that follow the paper content.

Section 5 of their paper is as follows:

This paper discussed different disk scheduling algorithms and compared their performances on two lists of I/O requests to determine each algorithm's time to execute the queue processes. Disk head scheduling takes into account two key concerns: throughput and fairness. Throughput refers to the number of jobs completed per unit time. Whereas, fairness refers to each job being treated equally and is ultimately served. An effective scheduling algorithm seeks a balance between these two issues. The best algorithm has the highest level of fairness and throughput. Table 11 shows the comparison between total seek time of all disk scheduling techniques and their overall throughput and fairness for each track in the requested queue.

Table 11 shows the comparison of total seek time of all disk scheduling techniques along with their overall throughput and fairness for each track in the request queue. The measures set for throughput and fairness include better, good, bad, moderate, and poor, as shown in Table 11. Figure-1 shows that the Optimize Two Head Disk Scheduling Algorithm (OTHDSA) has outstanding performance among all Disk Scheduling Techniques.



Disk Scheduling Algorithm	**Total Seek Time**		**Throughput**	**Fairness in terms of I/O request severed**
	I/O Request List-1	**I/O Request List-2**		
FCFS	644	498	Bad	Good
SSTF	236	258	Moderate	Poor
SCAN	230	280	Moderate	Moderate
C-SCAN	386	388	Good	Good
LOOK	208	250	Better than SCAN	Moderate
C-LOOK	326	322	Better than C-SCAN	Good
Modified C-LOOK	208	250	Better than C-LOOK	Good
OTHDSA	175	139	Better than all	Best
ZONE-BASED	208	250	Same as Modified C-Look	Good

TABLE 11: Result and analysis

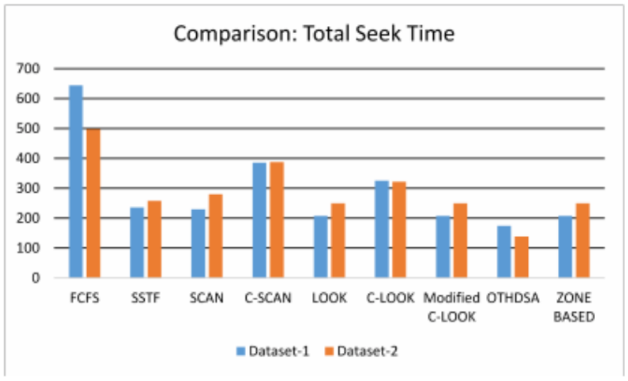


Fig. 1: Comparison of total seek time

☐ a. This study takes into account three different loads on disk

☐ b. Taking into account only the total seek time, SCAN is better than C-LOOK

☐ c. From their study, we can easily conclude on the best disk scheduling algorithm for real-time systems

☐ d. The curve of Figure 1 contains more information than Table 11

☐ e. According to the two selected datasets, and according to their their testing approach, one algorithm always performs better than the others

Your answer is incorrect.

The correct answers are:
According to the two selected datasets, and according to their their testing approach, one algorithm always performs better than the others,

Taking into account only the total seek time, SCAN is better than C-LOOK

Question **18**

Not answered

Marked out of 1.00

Check all the true claim about the code below. This code is taken from the dump1090 software used to analyze ADSB messages sent by aircrafts.

```
struct stDF *interactiveFindDF(uint32_t addr) {
    struct stDF *pDF = NULL;

    if (!pthread_mutex_lock(&Modes.pDF_mutex)) {
        pDF = Modes.pDF;
        while(pDF) {
            if (pDF->addr == addr) {
                pthread_mutex_unlock (&Modes.pDF_mutex);
                return (pDF);
            }
            pDF = pDF->pNext;
        }
        pthread_mutex_unlock (&Modes.pDF_mutex);
    }
    return (NULL);
}
```

- ☐ a. The function guarantees to have the mutex unlock when the function returns
- ☐ b. The function returns NULL if it cannot lock the mutex
- ☐ c. This function may be stuck in an infinite loop if the data structure is badly built
- ☐ d. This function manipulates a binary tree structure

Your answer is incorrect.

The correct answers are:
The function returns NULL if it cannot lock the mutex,

The function guarantees to have the mutex unlock when the function returns,

This function may be stuck in an infinite loop if the data structure is badly built

Question **19**

Not answered

Marked out of 1.00

Check all the true claims about Inter Process Communications.

- ☐ a. Memory can be shared between processes without the permission of the OS
- ☐ b. Killing a process is a communication between two processes
- ☐ c. In the shell, the "|" symbol represents a communication between two processes
- ☐ d. Memory sharing relies on the use of the Memory Management Unit if present
- ☐ e. Files can be used to exchange data between processes

Your answer is incorrect.

The correct answers are:
In the shell, the "|" symbol represents a communication between two processes,

Memory sharing relies on the use of the Memory Management Unit if present,

Files can be used to exchange data between processes,

Killing a process is a communication between two processes

Question **20**

Not answered

Marked out of 1.00

Check all the true claims about interrupts and real-time systems

- ☐ a. To improve the handling of deadlines, Interrupt Service Routines can be split between user mode and kernel mode
- ☐ b. Interrupt Service Routines can always be preempted by urgent software tasks
- ☐ c. The scheduler of RTOS schedules both all real-time tasks and all Interrupt Service Routines
- ☐ d. Interrupt Service Routines can prevent other parts of the kernel and other software tasks to execute by masking interrupts
- ☐ e. Interrupt Service Routines are first executed in user mode

Your answer is incorrect.

The correct answers are:
To improve the handling of deadlines, Interrupt Service Routines can be split between user mode and kernel mode,

Interrupt Service Routines can prevent other parts of the kernel and other software tasks to execute by masking interrupts