

Final Exam .

The **Microsoft FAT 16 file system** (supported by all of Microsoft's operating systems from latter versions of MS-DOS through Windows 10, as well as all Linux versions) is an example of a **file allocation table** (FAT) implementation, while the standard **Unix/Linux File System** (UFS/Ext[n]) is an example of an **indexed** (using index nodes known as **i-nodes**) implementation. Both implementations are reasonably robust and provide relatively fast access to a file's data, but the FAT 16 implementation has a **significant disadvantage** when compared to the i-node system in terms of memory usage requirements.

- A. Explain what this memory usage problem is for **FAT when compared to an i-node based system** ?

The entire **FAT table** (**indexes for all files in the file system**) is loaded into **RAM** when a **volume** is initially referenced, while only the specific **i-nodes** associated with current open files are brought into **RAM** using the **Ext type file systems** (**only i-nodes of objects in use brought into RAM**)

- B. Explain why Microsoft's **FAT 16** implementation **limits the size** of a file system to **2GB of total space**. Make sure you **clearly identify the limiting factor**.

Since a **FAT 16** table has only 2^{16} entries, and since Microsoft limits the maximum allocation unit size (cluster size) to 2^{15} (32 KB) the maximum single file size (as well as the maximum entire file system size) is limited to $2^{16} * 2^{15} = 2^{31} = 2\text{GB}$

$$12 + 15 + 16 = 43$$

For a typical **UNIX/Linux** file system using the **I-NODE** organization discussed in class (assume the 15 pointers used with the Ext2,3,4 file systems and discussed in class; 12 direct plus first, second and third level indirect): pointers.

1111111
128 64 32 16 8 4 2 1

- A. What is the size limit on a file created in a **8192 Byte (16 disk sectors) per element** (or logical block) **file system**, assuming that **8 bytes** are required for all pointers ?

8'192 | 8
0 19 | 1024
16
32

NOTE:

- You may express the answer in KBytes, MBytes, GBytes or powers of 2 as well as decimal, octal or hex, **just make it clear what kind of answer** you provide.

An 8 KB allocation unit using 8 byte pointers can hold 1024 pointers, so $(12 \text{ direct} * 8 \text{ KB}) + (1024 * 8\text{KB}) + (1024^2 * 8\text{KB}) + (1024^3 * 8\text{KB}) \approx 8\text{TB}$

FILE SIZE LIMIT IS: _____

Journaling provides support for rapid reboots after power failures by limiting the file system checking to only those things active at failure.

- B. The Linux **ext2** file system is built around the architecture discussed above, but, unlike the **ext3 file system**, ext2 does not include support for **journaling**. Explain what **journaling is**, and why it is **important enough** to be included in virtually all new file systems. (i.e., What is the **principal advantage** of a journaled file system ?)

Journaling provides support for rapid reboots after power failures by limiting the file system checking to only those things active at failure

Reference .

Let $\omega = 2 \ 3 \ 1 \ 3 \ 2 \ 4 \ 6 \ 7 \ 4 \ 5 \ 6 \ 7 \ 2 \ 1$, be a page reference stream. You are asked to work with two **stack algorithms** below. Remember, when 2 locations on the stack are being **compared**, a swap of locations is **only allowed** if the lower location has a **greater priority** than the upper location, **not** the same priority. You may find the work sheet on the back of this exam useful.

A. Assuming the primary memory is initially unloaded, how many **page faults** will the given reference stream have using the replacement algorithm **OPT** for:

1. A memory with **3 physical frames** $7 \text{ infinite} + 2 \text{ crossings} = 9$
2. A memory with **5 physical frames** $7 \text{ infinite} + 0 \text{ crossings} = 7$

ω	2	3	1	3	2	4	6	7	4	5	6	7	2	1
1	2	3	1	3	2	4	6	7	4	5	6	7	2	1
2	2	3	1	1	2	4	7	7	6	7	2			
3		2	2	3	1	2	6	6	6	5	6	7		
4			3	1	2	2	2	2	2	5	6			
5				3	1	2	1	1	1	1	5			
6					3	1	1	1	1	1				
7						3	4	4	4	4				
c1							3	3	3	3				
c2								2	2	2				
c3									1	1				
c4										1				
c5											1			
c6												8/4		
c7												7/5		
8	1	2	3	3	3	4	5	6	6	7	7	7	7	7

The code shown below **compiles with no errors**, and has **no system call or library call errors**. It is to be **executed** by a process that has just used the `execlp()` system call to load and run it from an `a.out` type file (the line numbers are included for reference in answering part B below). As it begins normal execution, it runs the **main()** function in its initial thread (**IT**), where it initializes a global enumeration called **color** to the constant value **RED**. The IT then initializes a mutex, and a condition variable and creates **two new threads**. After creating the threads, the IT **safely changes** the color variable to **ORANGE**. The IT then moves to a **join** call, waiting for the two other threads to finish so it can print its final message and exit, but **it never finishes**.

- A. Show what **output is produced** by this process, based on the code provided:
- B. Although **some progress** is made in this process (producing the output you listed above in Part A) the process **never finishes**.
 1. **Explain why the process never finishes**
 2. Using the line numbers included for reference, specify **what code** you would put **at what line locations**, to allow the process to come to a **normal termination**.

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <errno.h>
4
5 enum COLOR {RED, GREEN, ORANGE} color;
6
7 pthread_t           thread_id[2];
8 pthread_mutex_t     color_lock;
9 pthread_cond_t      color_condx;
10
11 void *th0();
12 void *th1();
13
14 int main(int argc, char *argv[])
15 {
16     color = RED;
17     printf("\nCOLOR INITIALIZED TO RED\n");
18     pthread_mutex_init(&color_lock, NULL);
19     pthread_cond_init(&color_condx, NULL);
20     if(pthread_create(&thread_id[0],NULL, th0, NULL) != 0) {
21         perror("pthread_create failed ");
22         exit(3);
23     }
24     if(pthread_create(&thread_id[1],NULL, th1, NULL) != 0) {
25         perror("pthread_create failed ");
26         exit(3);
27     }
28     pthread_mutex_lock(&color_lock);
29     color = ORANGE;
30     pthread_mutex_unlock(&color_lock);
31     pthread_cond_broadcast(&color_condx); // signal
32     pthread_join(thread_id[0], NULL);
33     pthread_join(thread_id[1], NULL);
34     printf("\nPROGRAM COMPLETE\n");
35     exit(0);
36 }
```

Code continued next page:

Continued from previous page:

```
37
38 void *th0() {
39     pthread_mutex_lock(&color_lock);
40     while(color != ORANGE)
41         pthread_cond_wait(&color_condx, &color_lock);
42     color = GREEN;
43     printf("\nCOLOR ORANGE CHANGED TO COLOR GREEN\n");
44     pthread_mutex_unlock(&color_lock);
45     return NULL;
46 }
47
48 void *th1() {
49     pthread_mutex_lock(&color_lock);
50     while(color != GREEN)
51         pthread_cond_wait(&color_condx, &color_lock);
52     color = ORANGE;
53     printf("\nCOLOR GREEN CHANGED TO COLOR ORANGE\n");
54     pthread_mutex_unlock(&color_lock);
55     return NULL;
56 }
```

- A. Show what **output is produced** by this process, based on the code provided:

COLOR INITIALIZED TO RED

COLOR ORANGE CHANGED TO COLOR GREEN

- B. Although **some progress** is made in this process (producing the output you listed above in Part A) the process **never finishes**.

1. Explain why the process never finished

The condition var is never signalled by th0, so th1 cannot leave its cond wait and IT is stuck in join

2. Using the line numbers included for reference, specify **what code** you would put **at what line locations**, to allow the process to come to a **normal termination**.

After line 44 and 54, must signal condx var:

`pthread_cond_signal(&color_condx);` OR

`pthread_cond_broadcast(&color_condx);`

```
38 void *th0() {
39     pthread_mutex_lock(&color_lock);
40     while(color != ORANGE)
41         pthread_cond_wait(&color_condx, &color_lock);
42     color = GREEN;
43     printf("\nCOLOR ORANGE CHANGED TO COLOR GREEN\n");
44     pthread_mutex_unlock(&color_lock);
→     pthread_cond_signal(&color_condx); // broadcast
45     return NULL;
46 }

47

48 void *th1() {
49     pthread_mutex_lock(&color_lock);
50     while(color != GREEN)
51         pthread_cond_wait(&color_condx, &color_lock);
52     color = ORANGE;
53     printf("\nCOLOR GREEN CHANGED TO COLOR ORANGE\n");
54     pthread_mutex_unlock(&color_lock);
→     pthread_cond_signal(&color_condx); // broadcast
55     return NULL;
56 }
```

The **inode based UNIX** file system generally allocates space to growing files by providing a **fixed size element (or block)** when more space is needed. An **element (or block)** is a contiguous collection of sectors on disk, and a common element size used in many implementations is **8KB**.

- A. Why are **elements** allocated instead of just allocating **one sector at a time** to a growing file ? **Elements conserve pointers and provide better disk performance**
- B. What **type of fragmentation** does this kind of allocation lead to ?
Internal fragmentation of $\frac{1}{2}$ an allocation unit (element) per file object
- C. If we have a **UNIX type** file system which uses **64KB elements**, and contains a set of files that all grow **sequentially** over time, what can we expect the **average amount of fragmentation per file** to be at any given time ?

$\frac{1}{2}$ an allocation unit (element) in the case of a 64 KB element leads to 32 KB internal fragmentation per file object on average

Consider the following details regarding a collection of UNIX objects:

Process A

Euid 0

Egid 1

	Directory	/
	uid 0	
	gid 1	
	Permissions	d rwx r-x r-x
	Directory	/abc
	uid 310	
	gid 20	
	Permissions	d rwx r-x ---
	File	/abc/file_one
	uid 320	
	gid 20	
	Permissions	- r-s rwx rwx

Process C

Euid 310

Egid 20

- Can process A **write** on **/abc/file_one** ? **Explain**

YES – process A has super user UID = 0 and has full access to everything

- Can process C use the **chmod 644 /abc/file_one** shell command successfully ? **Explain.**

NO – a **chmod** command by a non-super user process requires the process EUID to match the UID on the target i-node (no match here)