CS140 Final Review

Winter 2014

Administrivia

- Friday, March 21, 12:15-3:15pm
- Open book, covers all 17 lectures
  - (including topics already on the midterm)
- 50% of grade based on exams using this quantity:

  max (midterm > 0 ? final : 0,(midterm + final) /2)

Topics

- Threads & Processes
- Concurrency & Synchronization
- Scheduling
- Virtual Memory
- I/O
- Disks, File systems, Network file systems
- Protection & Security
- Virtual machines

Suggestions: you should know pintos material well by now, so study material not covered in pintos (equal representation on exam)

Peterson's Algorithm

```
/* Set all fields to zero on initialization */
struct peterson {
    int wants[2]; /* wants[i] = 1 means thread i wants or has lock */
    int not_turn; /* not this thread's turn if other one wants lock */
};

  /* i is thread number, which must be 0 or 1 */
void peterson_lock (struct peterson *p, int i) {
    p->wants[i] = 1;
    p->not_turn = i;
    while (p->wants[1-i] && p->not_turn == i);
}


  /* i is thread number, which must be 0 or 1 */
void peterson_unlock (struct peterson *p, int i){
    p->wants[i] = 0;
}
```

By declaring a global struct peterson globallock;, the two threads can then implement critical sections as follows:

```
    peterson_lock (&globallock, i);
    critical_section_for_thread_i ();
    peterson_unlock (&globallock, i);
```

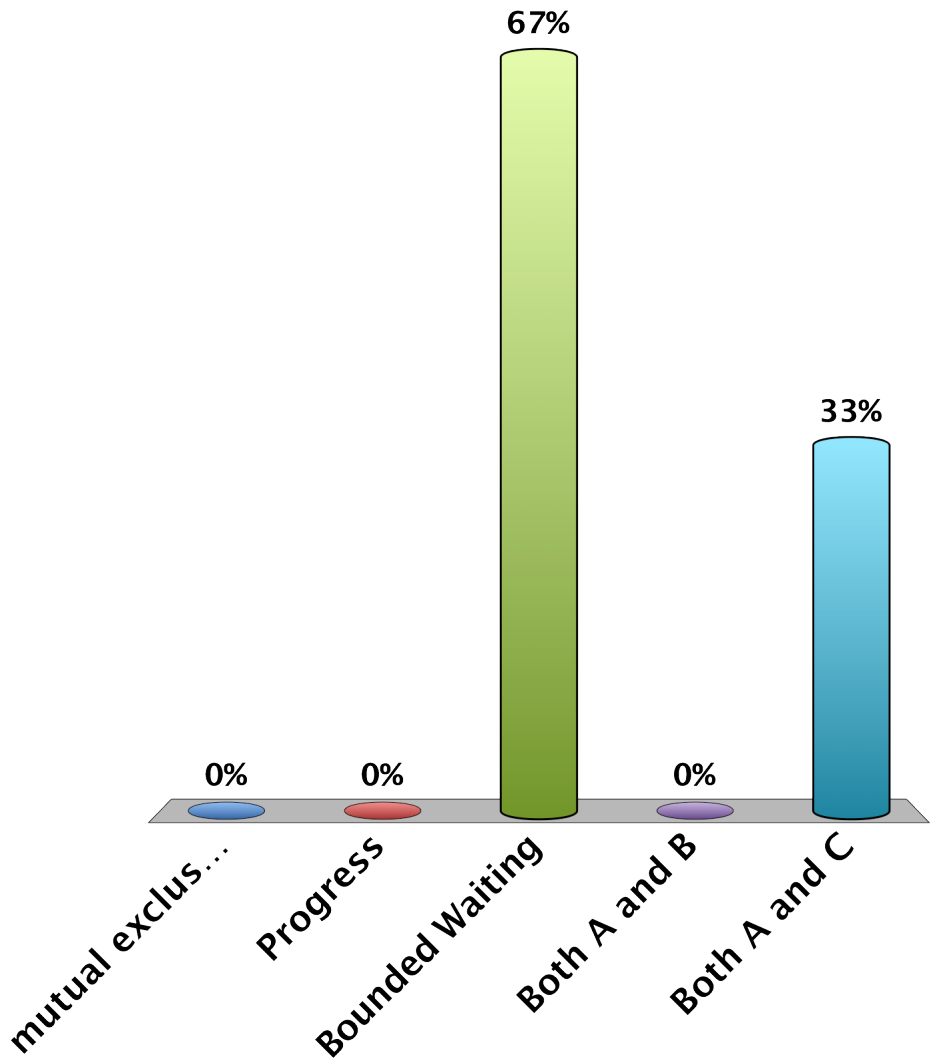
```
/* Set turn to zero on initialization */
struct mylock { int not_turn; };

/* i is thread number, which must be 0 or 1 */
void my_lock (struct mylock *p, int i) {
	while (p->not_turn == i) ;
}

/* i is thread number, which must be 0 or 1 */
void my_unlock (struct mylock *p, int i) {
	p->not_turn = i;
}
```

<u>Does this fail to guarantee any of the following? (Circle All That Apply)</u>

A) mutual exclusion.
B) progress.
C) bounded waiting.

Does not guarantee….

A. mutual exclusion
B. Progress
C. Bounded Waiting
D. Both A and B
E. Both A and C



Answer

Answer: Fails to guarantee progress.

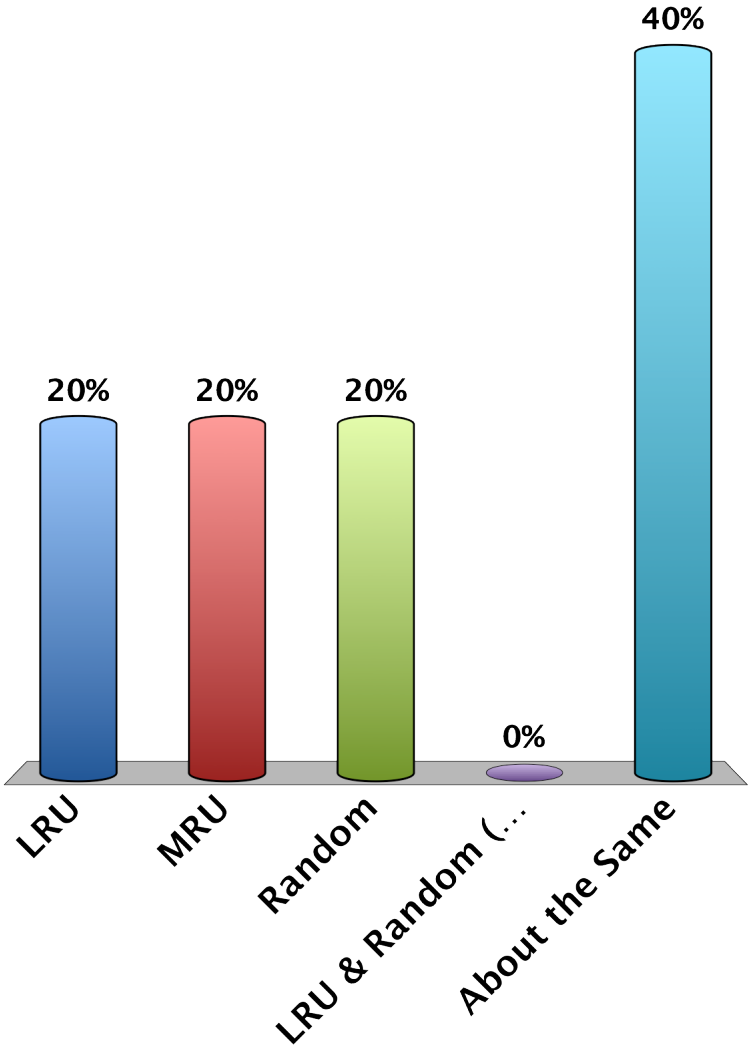
Explanation: If thread 0 spends all of its time in the remainder section (i.e., does not enter the critical section), then thread 1 will never get into the critical section.

```
void loop (void) {
    char *p, *begin = sbrk (0);          /* end of the heap */
    char *end = begin + 0x20000000;      /* grow heap by 512 MB */
    brk (end);
    for (p = begin; p += 4096) {
        if (p >= end)
            p = begin;
        *p += 1;                         /* modify first byte of page */
    }
}
```

Page Size = 4096 bytes, 256MB Physical Mem

Which eviction policy minimizes the expected # of page faults

A. LRU
B. MRU
C. Random
D. LRU & Random (Tie)
E. About the Same



Answer: B (MRU)

Explanation: Because the function is looping over memory, the most recently accessed page is the one that will be accessed furthest in the future, and thus the best one to evict. With MRU, roughly the first half of the area being scanned will be brought into memory and stay there, while the second half of the loop will be all page faults. In contract, LRU will fault in every iteration of the loop, and random will perform between the other two.

Suppose you run a single virtual machine with 1,024 MB of “physical” memory on top of a VMM, yet the underlying hardware only has 768 MB of machine memory available. The guest OS uses LRU.

Will the system have better performance if the VMM uses LRU page replacement or random page replacement (keeping in mind that the guest OS always does LRU)?

Choose The Best Answer

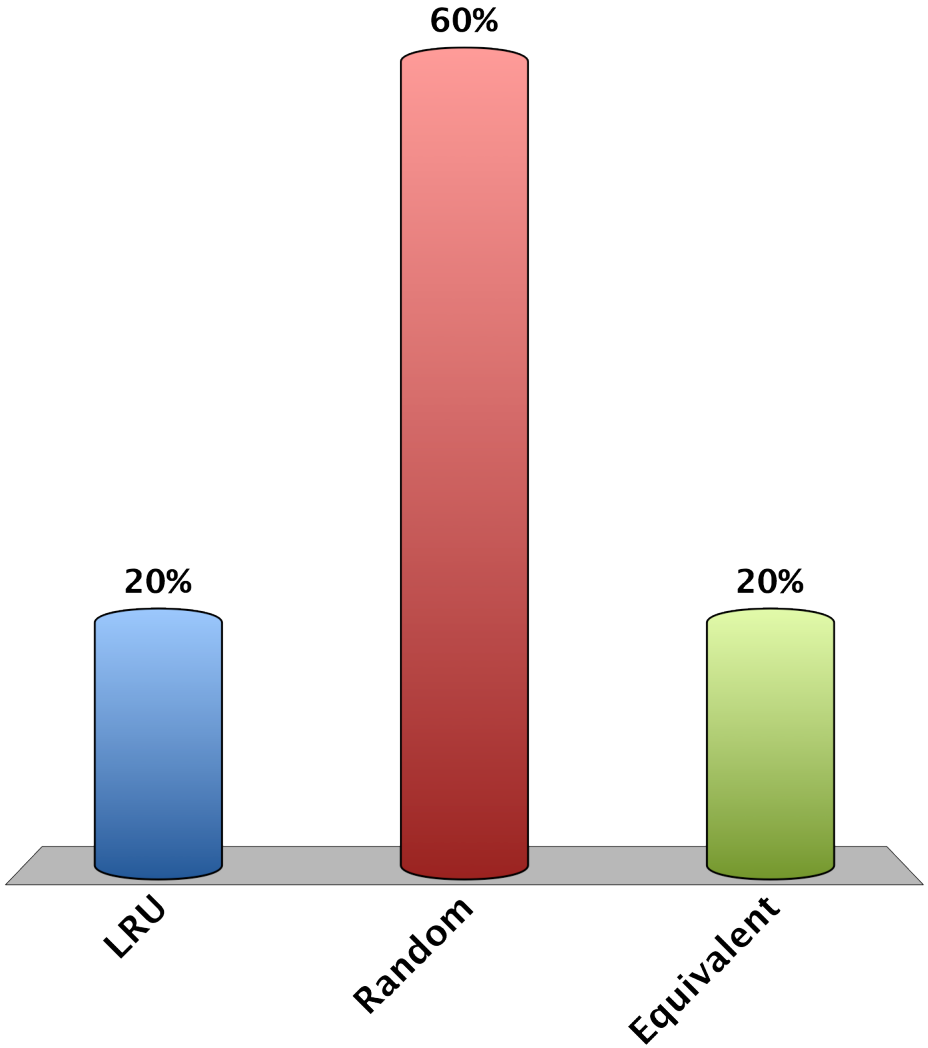
A) Performance will be better if the VMM uses LRU replacement.

B) Performance will be better if the VMM uses random replacement.

C) Expected performance will be the same whether the VMM uses LRU or random replacement.

Which policy gets better performance?

A. LRU
B. Random
C. Equivalent



Answer: B

Explanation: With LRU replacement, the kernel will constantly page out the least recently used page, but this means the virtual page least recently used by an application. The kernel itself will make use of the physical page while paging it out (if it is dirty), and then immediately recycle the frame to satisfy its own memory pressure. Thus, the least recently used physical page is the most likely to be used by the kernel as soon as it needs to do a page eviction, and thus is a good candidate not to page out.

Suppose you have a single-core machine with a Unix-like OS offering two thread packages:

- **libuthreads.a** is a user-level implementation of threads, in which each process has only one kernel thread (which runs all application-level threads). Hence, all thread scheduling decisions are made by the user-level thread library. (This is what we called n : 1 threads in class.)

- **libkthreads.a** uses kernel-level threads to implement application-level threads, so that each kernel-level thread runs only one application-level thread and all scheduling decisions are made in the kernel. (This is what we called 1 : 1 threads in class.)

You decide to run a fine-grained simulation of 100 particles. Each particle has its own thread, and, once per period, updates its position based on the positions of all other particles, requiring 200 instructions. The simulation does not perform any I/O. With which of the two thread packages would you expect your simulation to run faster?

Circle the best answer.

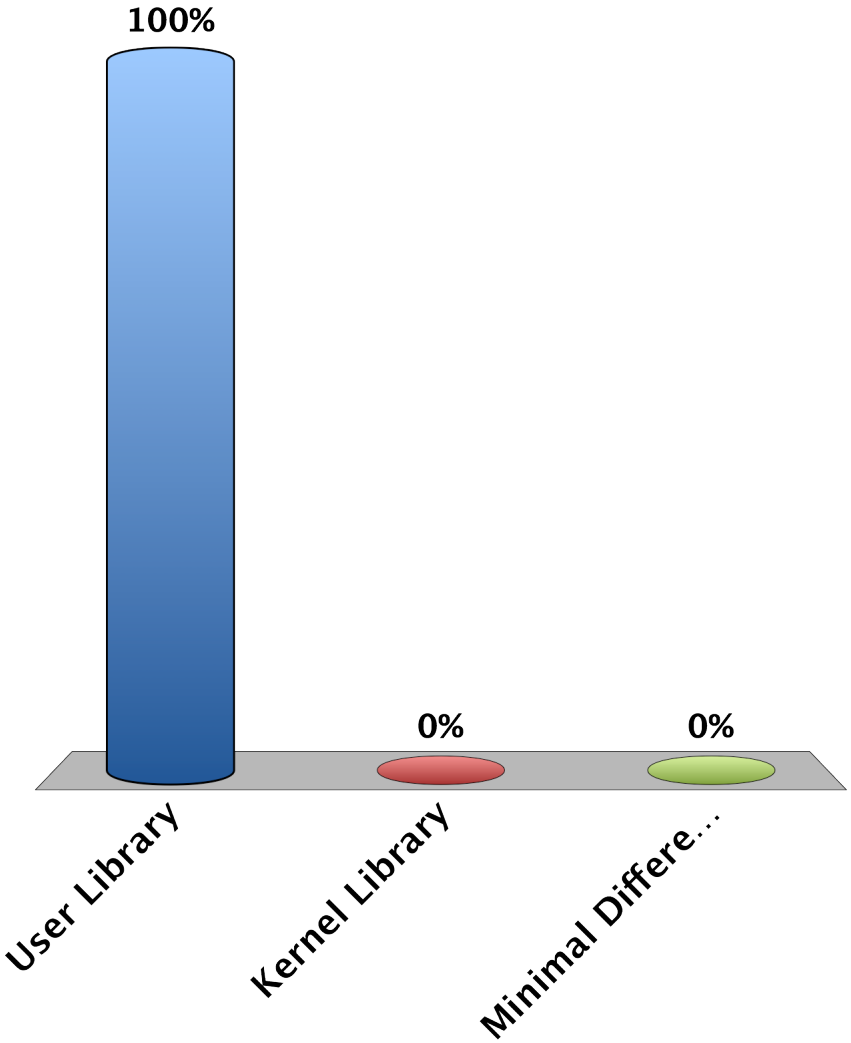
A. The simulation will run faster with libuthreads.a (n : 1 threads).

B. The simulation will run faster with libkthreads.a (1 : 1 threads).

C. The choice of thread package will only affect performance minimally, and it could go either way.

Which approach is best?

A. User Library
B. Kernel Library
C. Minimal Difference



Answer: **libuthreads.a** (n : 1 threads).

Explanation: The program requires switching threads every few hundred instructions. With 1 : 1 threads, such switches require a system call to yield the CPU. By contrast, with n : 1 threads, a yield is just a function call. Function calls are much faster than system calls. Moreover, the system call overhead will likely be much more than a few hundred instructions.

Now you decide to run a web server on a big machine with 10 disks in it. The server gets concurrent requests from dozens of clients for random files. Because the 10 disks are storing so much data, the overwhelming majority of requests miss in the buffer cache and require reading data from disk (which takes an average of 10 milliseconds). In order to process requests concurrently, the web server handles each request in a separate thread. Which thread package would allow the server to handle more requests per second?

Circle the best answer.

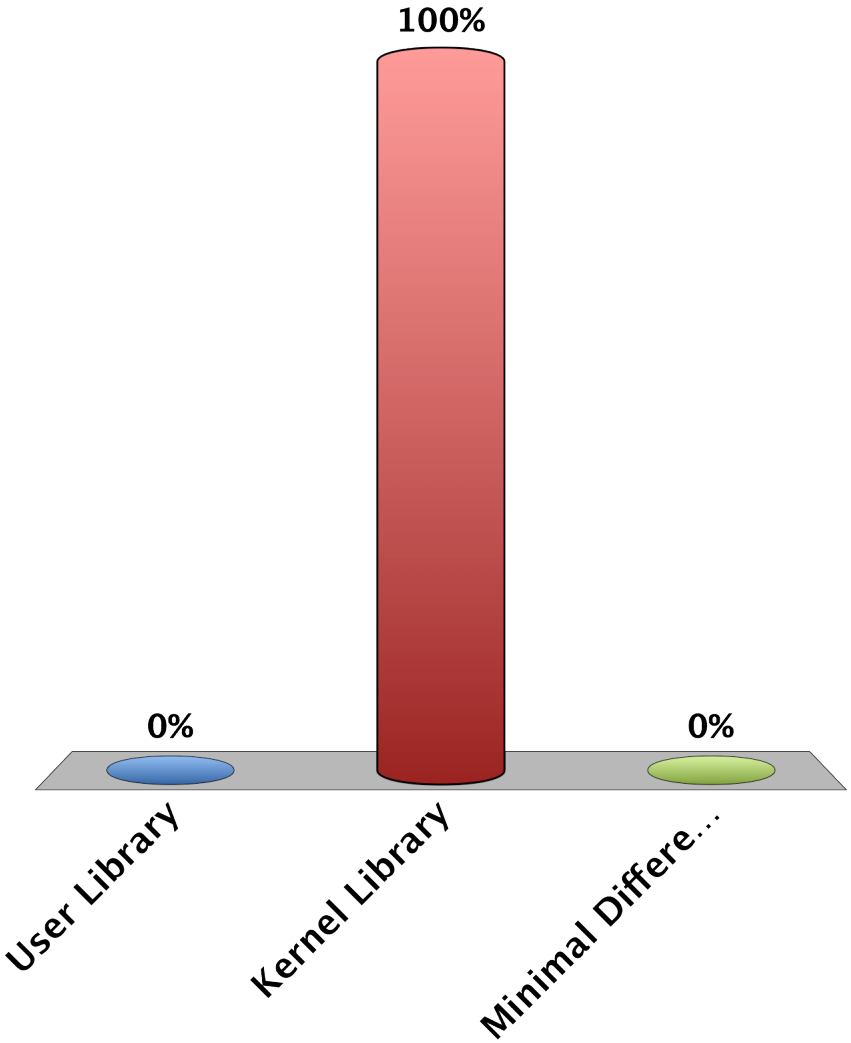
A. The simulation will run faster with libuthreads.a (n : 1 threads).

B. The simulation will run faster with libkthreads.a (1 : 1 threads).

C. The choice of thread package will only affect performance minimally, and it could go either way.

Which approach is best?

A. User Library
B. Kernel Library
C. Minimal Difference



Answer: **<u>libkthreads.a</u>** (1 : 1 threads).

To get high throughput you want to maximize utilization of the CPU and disks. However, an n : 1 thread package will put all threads to sleep when any thread blocks on a disk read. By contrast, 1 : 1 threads can have all 10 disks and the CPU busy at the same time.

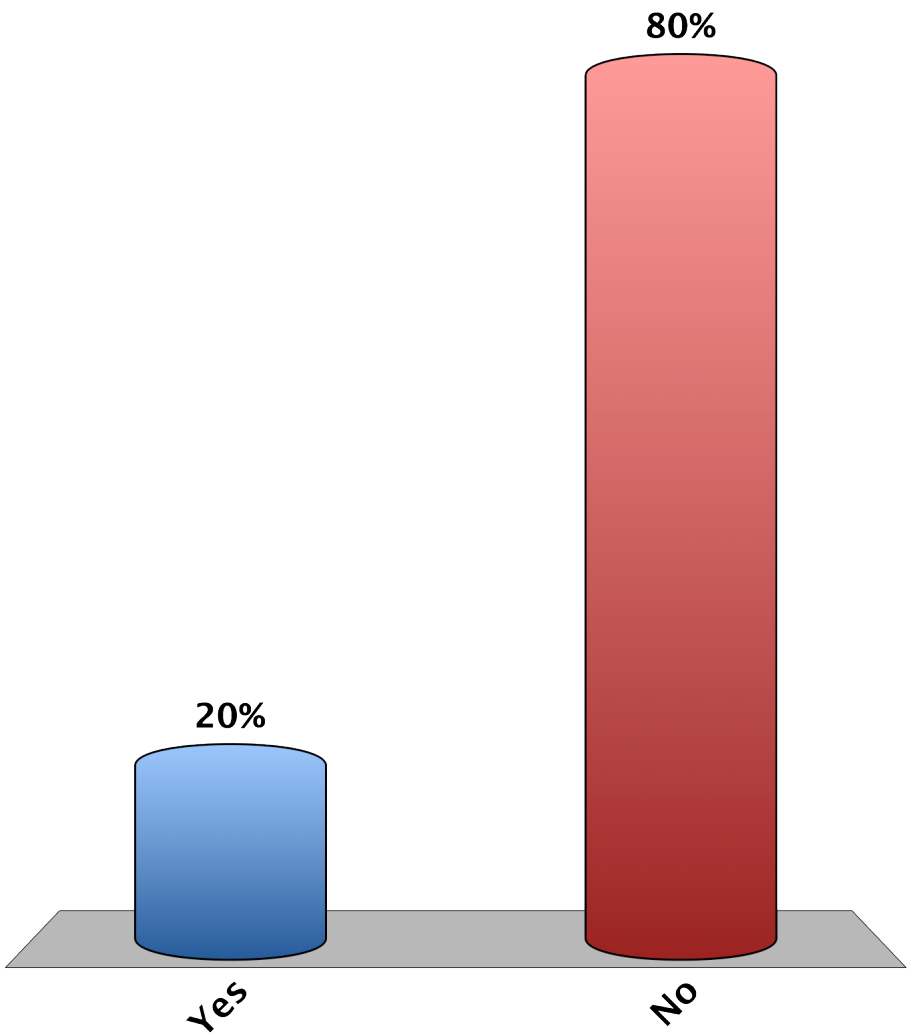
A hacker gains access to your Linux filesystem. Can the attacker use the files on your laptop to recover your login username/password? If so, describe how, if not, explain why not.

A. Yes, the attacker can figure out your password from the contents of files on your hard disk.
B. No, with overwhelming probability the attacker will not be able to recover your password despite having the contents of your hard disk.

Can the attacker recover your password?

A. Yes

B. No



Answer: No

Explanation: The laptop stores a hash of the password (H(salt,password)), not the actual password. The attacker can guess the password by computing H(salt,guess) and seeing if the hash matches the one stored in your file system—just as login verifies your password when you log in. However, even if the attacker can guess a billion (~ 230) passwords per second (a highly unlikely rate), it would still take an expected 249 seconds (~18 million years) to guess the right one.

Consider the following program:

```
#define FILE "tmp" int main (int argc, char **argv) {
  int i, fd;
  for (i = 0; i < 100; i++) {
    fd = open (FILE, O_CREAT|O_WRONLY|O_TRUNC, 0666);  /* create FILE */
    close (fd);
    unlink (FILE);     /* delete FILE */
  }
   return 0;
}
```

Q: Is the total number of disk writes required by this program (including writes delayed until after the program exits) likely to be more on XFS or on FFS with soft updates?

Circle the best answer.

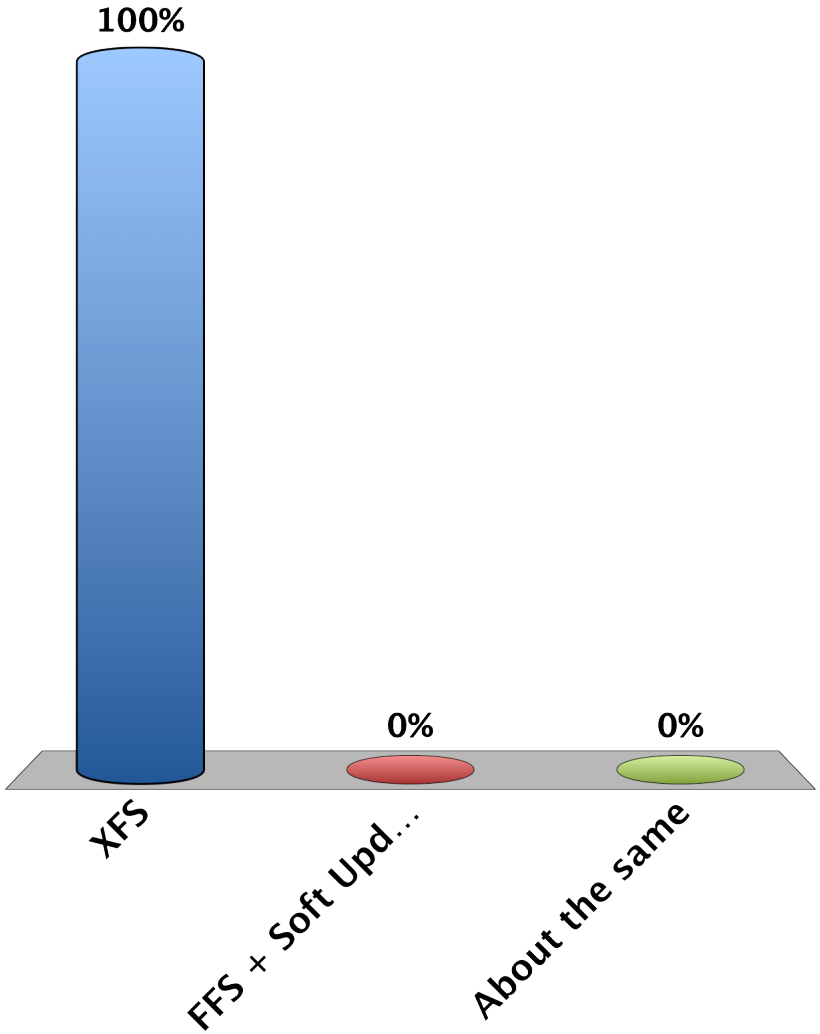
A) The program requires more disk writes with XFS.

B) The program requires more disk writes with FFS + soft updates.

C) The program should require a comparable number of writes in both cases.

Which FS has more disk writes?

A. XFS
B. FFS + Soft Updates
C. About the same



Answer: More with XFS

Explanation:

- With XFS, all metadata changes (such as creating and deleting files) must be written to the log. Thus, the program will require 200 log entries to be written.
- By contrast, soft updates allow the file system to recognize when one operation is simply undoing another operation that has not yet been committed to disk. Thus, FFS + soft updates doesn't require any disk writes for each iteration of the loop.
- (In both cases, the modification time of the directory will eventually have to be updated, but this only has to happen once for the whole execution of the program.)

VMware ESX saves memory by identifying multiple copies of the same physical page and mapping them all copy-on-write to a single machine page. This technique saves memory when, for example, two virtual machines are running the same guest operating system, as the pages containing kernel code will be identical across the two VMs.

Could this technique ever save memory when a single virtual machine is running (assuming the VM is configured to see more "physical" memory than there is machine memory)?

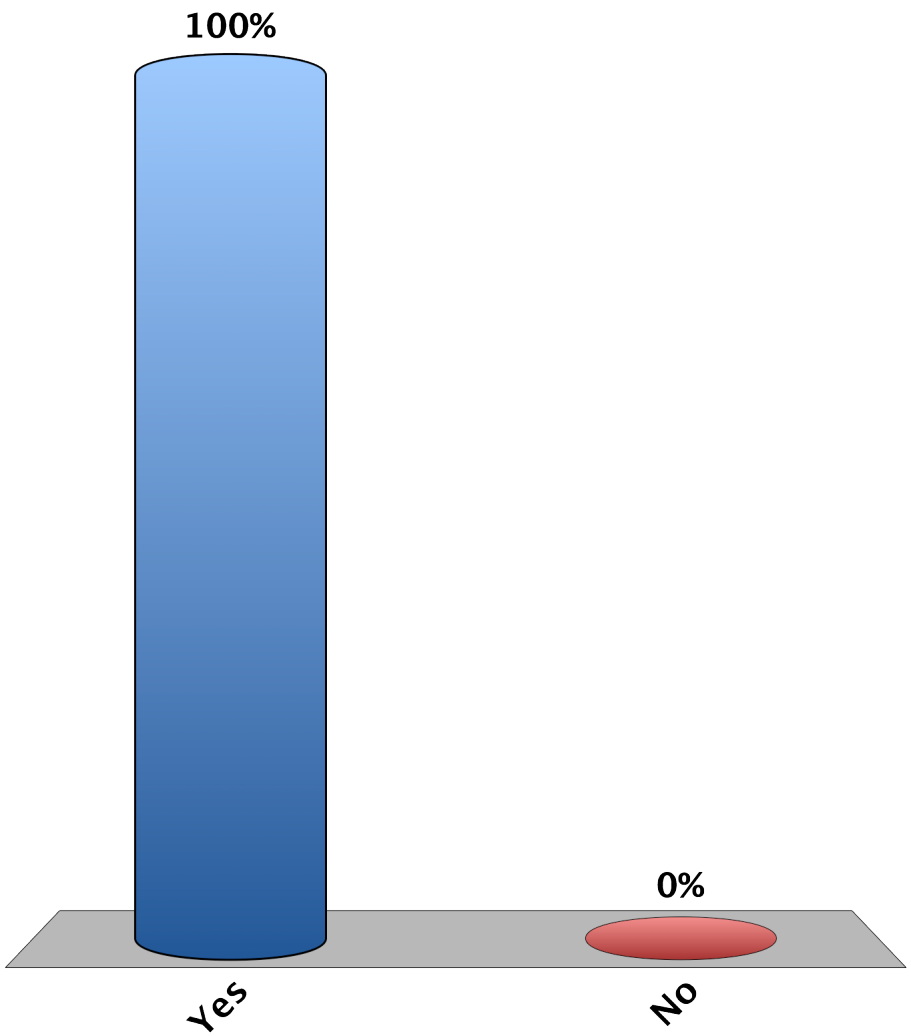
<u>Circle The Best Answer</u>

A) Yes

B) No

Could this save memory?

A. Yes
B. No



Answer: Yes

Explanation:

- If you copy a file, both the old and new file blocks may be stored in the buffer cache.

- Another example is that many OSes zero pages in the idle loop, so as to have many zero-filled pages on hand to map into BSS and stack segments of new processes.

In class we described three file descriptor structures:

(a) Indexed files.

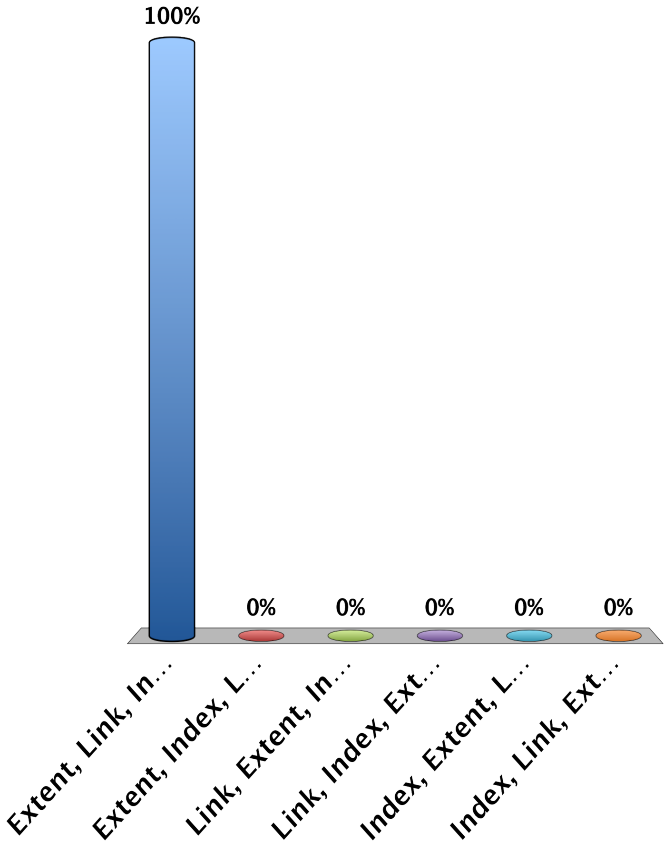
(b) Linked files.

(c) Contiguous (extent-based) allocation.

Each of the structures has its advantages and disadvantages depending on the goals for the file system and the expected file access pattern.

Sequential Access to Large Files

A. Extent, Link, Index
B. Extent, Index, Link
C. Link, Extent, Index
D. Link, Index, Extent
E. Index, Extent, Link
F. Index, Link, Extent

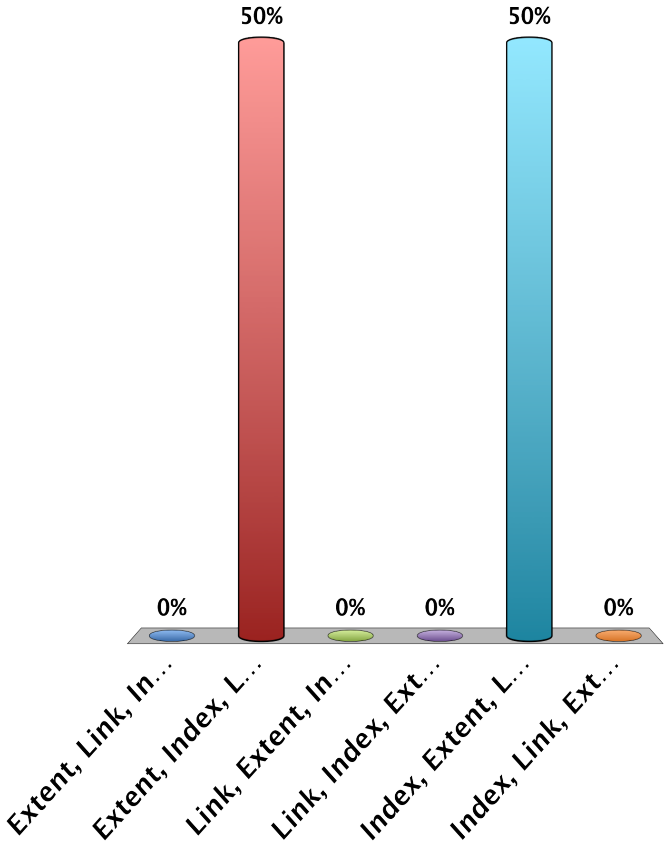


Answer: extent, link, index

Explanation: It is easy to see that extent is the best structure for sequential access to very large files, since in extent files are contiguously allocated and the next block to read is physically the next on the disk. No seek time to find the next block, and each block will be read sequentially as the disk head moves.
Both link and index require some looking up operation in order to know where the next block is. However, (b) performances are worse, since for "very large files" it may require multiple disk accesses to the indirect blocks.

Random Access to Large Files

A. Extent, Link, Index
B. Extent, Index, Link
C. Link, Extent, Index
D. Link, Index, Extent
E. Index, Extent, Link
F. Index, Link, Extent



Answer: Extent, Index, Link

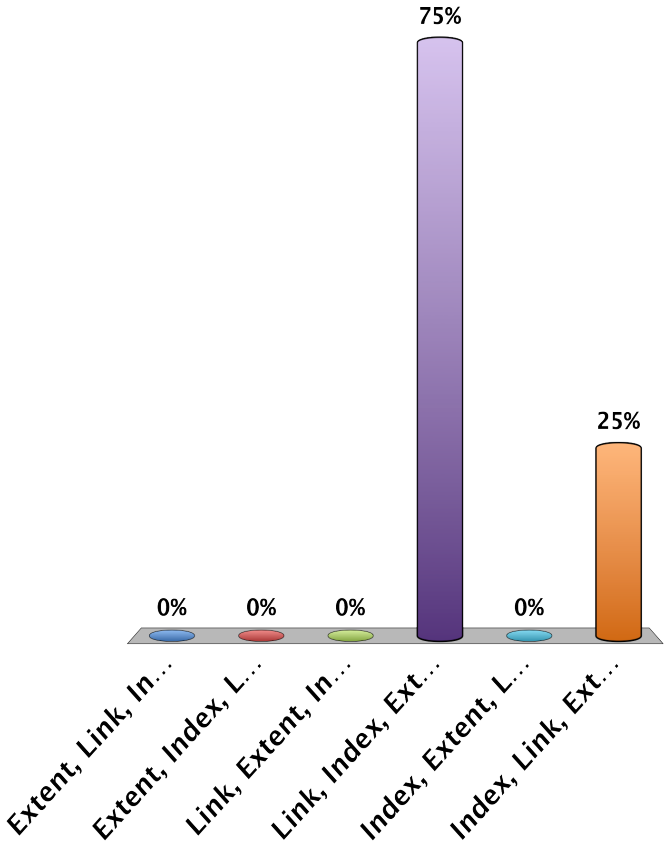
Extent is still the best structure here: just need to use an offset.

Index will probably need to look at some levels of indirect blocks in order to find the right block to access (we are dealing with very large files).

Link is absolutely the worst structure. In fact, in order to find a random block, we will need to traverse a linked list of blocks, which will take a time linear in the offset size.

Disk Utilization

A. Extent, Link, Index
B. Extent, Index, Link
C. Link, Extent, Index
D. Link, Index, Extent
E. Index, Extent, Link
F. Index, Link, Extent



Answer: Link, Index, Extent

Exent can suffer heavily of external fragmentation, especially for large files. So it is not the best stucture for getting the most bytes on the disk, since lots of space will be wasted. However, for small files and large block size, (c) might prove to be better than (a) and (b).
(a) and (b) stuctures are generally more suitable for this question. The metadata overhead for (b) is likely to be smaller than the one for (a), since it only needs pointers to the next allocated block rather than an entire block (or blocks) which may or may not be totally used.

The End