Cmpt 300 – Operating Systems
Sample Final

Summer 1999

Read the following carefully:

- You *may* refer to notes or textbooks during this exam.
- Read each question thoroughly before you begin it.
- Some questions must be answered on the question paper.
- This sample examination paper is provided as a revision aid only. Questions on the final exam may cover different topics, and differ in difficulty from these questions.
- You may use the back of your answer booklet for any rough work. More paper is available if you need it.
- When you are asked to finish, please do so immediately. Continuing to write after the official end of the examination is unfair to fellow students, and carries an appropriate penalty.

1. *(16 marks)*

   Which of the following are true and which are false? (Shade the leftmost circle for statements that are true, and the rightmost circle for statements that are false. You will lose ½ mark for each incorrect answer.)

   (a) A logical block can be smaller than a physical block. Ⓣ Ⓕ

   (b) Object files passed to a linker contain relocation entries. Ⓣ Ⓕ

   (c) A load image (or executable) is created by linking a binary object file with the system libraries. Ⓣ Ⓕ

   (d) If we added appropriate user-mode code (but made no kernel changes), Windows NT could be certified as a Unix operating system. Ⓣ Ⓕ

   (e) Relocation information is only required for programs written in a high-level language Ⓣ Ⓕ

   (f) The Macintosh Finder and the Windows Program Manager are examples of shells. Ⓣ Ⓕ

   (g) The processor's primary cache is the fastest storage available inside the computer. Ⓣ Ⓕ

   (h) An operating system implements processes by maintaining a process image for each process. Ⓣ Ⓕ

   (i) Lamport's "bakery algorithm" can solve the critical-section problem for n tasks. Ⓣ Ⓕ

   (j) The "banker's algorithm" is rarely used in current operating systems. Ⓣ Ⓕ

   (k) If deadlock has been avoided, starvation is impossible. Ⓣ Ⓕ

   (l) The "banker's algorithm" can be simply extended to avoid livelock instead of deadlock. Ⓣ Ⓕ

   (m) Code that uses a *Semaphore* variable with signal and wait can be trivially rewritten to use a *Condition* variable with csignal and cwait. Ⓣ Ⓕ

   (n) A program running from inside a logical address space may still use position-independent code. Ⓣ Ⓕ

   (o) Operating system kernels can reside in virtual memory Ⓣ Ⓕ

   (p) With paging and segmentation, it is possible for multiple instances of a process to share their code. Ⓣ Ⓕ

2. *(3+3 marks)*

   (a) How could an operating system that uses global page replacement detect thrashing?

   (b) Once detected, what should the operating system do to remedy the problem? Give two reasonable solutions and their trade-offs with respect to each other.

3. *(3 marks)*

   In an object-oriented language, all the methods for each class are stored together. Explain why this may not be an ideal layout on a system that expects efficient use of virtual memory.

4. *(3 marks)*

   Suppose a computer has 10,000 bytes of physical memory with a page size of 1000 bytes. The operating system has just begun a program which uses 20,000 bytes of virtual memory. None of the program is currently in memory, and all frames are empty. Assuming the following page reference sequence, which pages are in which frames at the end of the sequence? (Show your working).

   0, 1, 2, 3, 4, 1, 0, 5, 9, 11, 15, 9, 10, 15, 16, 2, 3, 11, 10, 6, 8, 17, 19, 6, 3

5. *(2+2+2 marks)*

   Consider the following three operating systems:

   - MS-DOS — a simple, uniprocessing operating system, developed for personal computers in the early 1980s.
   - NEXTSTEP — a multiprocessing operating system developed for personal workstations in the late 1980s and early 1990s (typically the workstations are used by a single user at a time).
   - OS/390 — a multiuser, multiprocessing, mainframe operating system; in development for many years.

   Based on your expectations as an operating systems designer, which of these operating systems should:

   (a) Prevent deadlock occurring for some of their resources.

   (b) Expect to suffer resource-contention problems in practice.

   (c) Include deadlock detection and/or deadlock avoidance code.

6. *(2+2 marks)*

   Consider the following snapshot of a system:

	Allocation	*Max*	*Available*
	A B C D	A B C D	A B C D
P_0	0 0 1 2	0 0 1 2	1 5 2 0
P_1	1 0 0 0	1 7 5 0	
P_2	1 3 5 4	2 3 5 6	
P_3	0 6 3 2	0 6 5 2	
P_4	0 0 1 4	0 6 5 6	

   Answer the following questions using the banker's algorithm:

   (a) Is the system in a safe state?

   (b) If a request from process P_1 arrives for (0,4,2,0), can the request be granted immediately?

   For each part, show how you derived your answer.

7. *(5 marks)*

Choose the most correct answer from the answers offered. (Shade *one* circle out of those given, corresponding to the answer you feel to be the *most correct* of those available.)

(a) The banker's algorithm

- ◯ Is only run when the system is in a safe state
- ◯ Is simpler to implement than the resource-allocation–graph algorithm
- ◯ Can only operate when there is a single instance of each resource type
- ◯ Can only be run inside an operating system kernel

(b) The clock and FPB algorithms both attempt to replace the page that

- ◯ Was written the longest time ago
- ◯ Was loaded into memory the longest time ago
- ◯ Is least likely to be accessed in the near future
- ◯ Will be required again in the shortest time

(c) Batch processing was introduced to

- ◯ Increase throughput
- ◯ Allow background processing
- ◯ Provide multiprogramming
- ◯ Overlap CPU and I/O operations

(d) A processor provides an atomic exchange instruction, where exchange(*a*, *b*) sets *b* to *a*'s old value, and sets *a* to *b*'s old value. This instruction is

- ◯ Less powerful than test_and_set
- ◯ As powerful as test_and_set
- ◯ More powerful than test_and_set
- ◯ As powerful as compare_and_swap

(e) Compared to preemptive multitasking, nonpreemptive multitasking on a uniprocessor machine

- ◯ Gives the operating system more control
- ◯ Is more difficult to implement
- ◯ Provides greater CPU protection
- ◯ Reduces the need to protect critical sections in user programs

8. *(3 marks)*

Explain and contrast the tradeoffs involved in using an inverted page table versus a two-level page table.

9. *(3+3 marks)*

In managing a networked file system, the operating system may assume that the following techniques will improve performance:

(a) Client-side write buffering

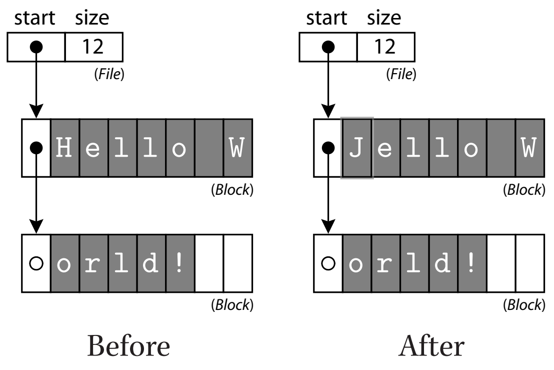
(b) Client-side read caching

For each assumption, describe a scenario that violates it (your scenario should be specific to a networked filesystem).

10. *(2 + 3 + 3 + 3 + 3+8+5+2+3 marks)*

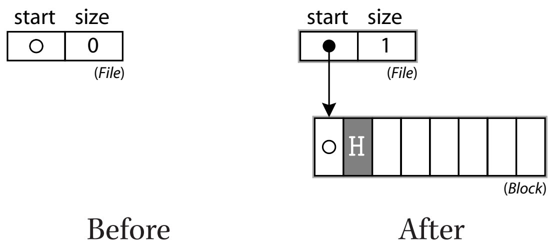
Linked file allocation easily supports two different kinds of writes.

In an *overwriting write*, newly-written data overwrites existing data—an overwriting write cannot make the file any larger or cause new blocks to be allocated. For example, overwriting the first character of a file containing "`Hello World!`" with a `J`, leaving "`Jello World!`" (see right).

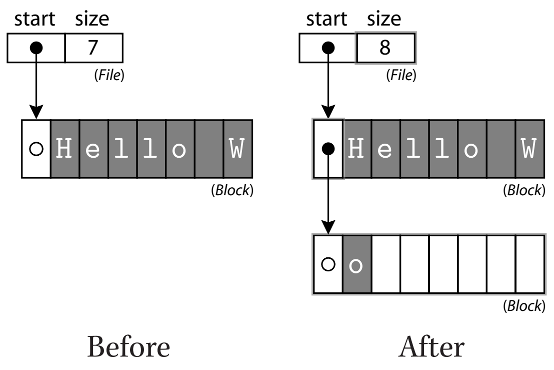


An *appending write* adds new data to the end of a file. An appending write has three possible cases:

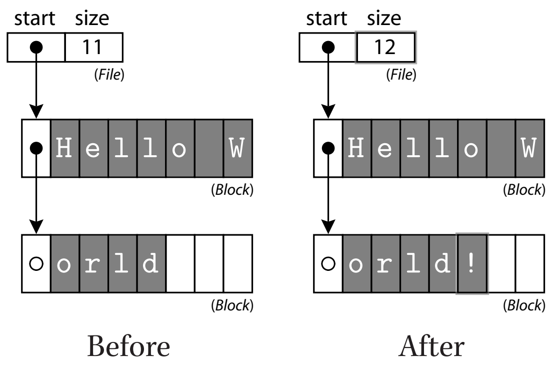
A. The file is empty and has no blocks allocated. When the first character is written to the file, a block must be allocated and written, and the directory entry must be updated to store the location of the first block in the file and the new file size. For example, appending '`H`' to an empty file (see right).



B. The last block in the file is full, requiring that another block be added to the file. As before, the directory entry must be updated to reflect the new file size. For example, appending '`o`' to a file that has one full block containing "`Hello W`" (see right).



C. There is room in the final block of the file for another character, so the file does not need to have another block allocated. The directory entry must once again be updated to reflect the new file length. For example, appending '`!`' to a file containing "`Hello World`", where the last block of the file has space for three more characters (see right).



(continued on next page)

For this question, assume the following:

- Characters and integers are 16 bits wide.
- The logical block size is eight 16-bit words.
- User programs read or write the file one 16-bit character at a time.

(a) In the best case, what fraction of each disk *Block* stores user data?

(b) What is the maximum amount of internal fragmentation that can occur in a file in this filesystem?

(c) Below is an implementation of a routine to read a character from the file. It contains a serious error—what is it?

```
struct File {
        uint16_t start;
        uint16_t size;
};

struct BlockHeader {
        uint16_t next_block;
};

const uint16_t BLOCK_SIZE = 8;
const uint16_t BLOCK_CAPACITY = BLOCK_SIZE − 1;

struct Block {
        BlockHeader header;
        wchar_t data[BLOCK_CAPACITY];
}

wchar_t read_character(File file, uint16_t position) {
        uint16_t diskblock = file.start;

        if (position >= file.size)
                throw InvalidFilePosition();

        for ( ; ; ) {
                Block block = read_block(diskblock);
                if (position < BLOCK_SIZE)
                        return block.data[position];
                diskblock = block.header.next_block;
                position = position − BLOCK_SIZE;
        }
}
```

(continued on next page)

(d) What additional data should you store in the *File* structure to make appends more efficient?

(e) It is possible to provide another kind of write, an *inserting write*. An inserting write is a generalization of an appending write: An inserting write can insert data anywhere in the file without upsetting existing data, whereas an appending write can only add data onto the end of a file. For example, if we open a file containing the phrase "`Hello World!`" and the file pointer is just after the first '`o`', inserting '` `', '`S`', '`m`', '`a`', '`l`', '`l`' leaves the file containing "`Hello Small World!`" (c.f., overwriting writes, which yield "`Hello Small!`", and appending writes, which yield "`Hello World! Small`").

Develop a modification to the linked-allocation scheme outlined above to allow inserting writes. The mechanism you develop *must* have a constant upper bound on the number of blocks it writes for any single insertion.

i. Give your revised definitions for *BlockHeader* and/or *File*.

ii. Draw diagrams to illustrate your allocation scheme for each distinct case of inserting a 16-bit word into the file.

iii. Rewrite read_character to use your new scheme. (Ideally, your changes to this function should be small.)

iv. In the best case, what fraction each of disk *Block* stores file contents?

v. What is the maximum amount of internal fragmentation that can occur in a file under your allocation scheme?