Midterm 2, November 2, 1998 - solutions

1. Explain thrashing. Define the Working Set Paging algorithm and explain how one of the features of the Working Set paging algorithm is to avoid thrashing. (11)

Thrashing is a situation in which the processor spends most of its time either processing page faults or in page fault idle. This occurs either because the degree of multiprogramming is too high, or one or more processes are badly behaved and use too much memory. In either case, it is because demand for memory exceeds supply.

The working set paging algorithm keeps in memory exactly those pages for each process that have been referenced in the preceding T virtual time units. Or - the working set paging algorithm keeps in memory those pages that a process needs to run efficiently.

The working set paging algorithm will not allow a process to run (i.e. put it in the "in memory queue") unless there is enough memory available for its working set.

2. For the following page reference string, give the number of page faults for memories of sizes 3 and 4, and for working set, for a working set parameter value of 4.5. No partial credit, but show your work. (21)
.ft CW

2 3 4 5 2 3 1 2 3 4 5 1 2 3

\	3	4
LRU	12	10
FIFO	11	12
OPT	8	6

WS 10
==========
work:

3. What is the function of a page table? What is the function of the memory map(also called the "core map")? What is the relation of one to the other. Why do you need both?

A page table contains entries that map virtual page numbers to physical page numbers.

A memory maps maps physical page frames to virtual page numbers. The information that we really care about in this mapping is not to be able to associate page numbers, but to be able to locate Page Table Entries that point to a particular physical page frame.

We need to locate these PTE's when a page is to be replaced. The PTE's can contain referenced and modified bits that helps the OS make its page replacement decision. Most importantly, the PTE for the page that is to be swapped out must be modified to reflect the fact that it's no longer in memory.

You received 4 points for knowing what a page table is. The remaining 8 points are given for how well you knew what a memory map is and WHY we need it to do paging.

4. Suppose that you have a region of memory shared between two processes. Suppose that within the shared region are memory addresses (e.g. there is a linked list). Why is that a problem? How would or could you solve the problem?

A user process can only reference memory using a virtual address. Thus, any address stored in the shared region is a virtual address. The translation of this virtual address depends on the context, i.e. what process is doing the reference. Thus, a virtual address stored in the shared region can map to different physical locations when referenced by different processes. Also, the address will not always be valid for all processes.

One solution is to make sure that the shared region resides at the same virtual memory location for all processes. For example, we can reserve the top

portion of the virtual address space for a shared segment.

Another solution is to interpret all address values as relative locations (for example, relative to the location of that address value). The C code below illustrates this concept.

```
Struct LinkedList {
  Data d;
  LinkedList *next;
}

LinkedList * getNextElement(LinkedList *element) {
  return &(element->next) + element->next;
}
```

5-6 each for identifying the problem and proposing ONE valid solution. Prohibiting sharing is not a valid solution!

You received 2 points if you mentioned the need for mutual exclusion. You got these points because you said something that is not wrong, but does not address the more fundamental problem.

5. What are the tradeoff between contiguous file allocation and linked file allocation? Explain the use of extents for contiguous allocation and explain how they improve things.

Contiguous file allocation means allocating the files in consecutive blocks on the disk.

Contiguous File Allocation
+ fast sequential access
+ fast random access
+ low disk space overhead
- external fragmentation
- hard to grow file

Linked File Allocation
+ no external fragmentation
+ easy to grow file
- very slow random access (must search through the whole list)

- potentially slow sequential access (if the list blocks are randomly allocated over the disk surface)
- some disk space overhead to store pointers to the next block.

The use of extents is a hybrid of the above two layout methods. The idea is that the entire file does not have to be contiguous, but we try to allocate a file in as few contiguous, variable sized blocks as possible.

There are two ways to keep track of the contiguous blocks allocated to a file. We can have a linked list of extents. It would have all the advantages of linked list allocation, plus faster random access since fewer reads would be required to locate a block.

If we place a small upper limit on the number of contiguous blocks, say 16, we can keep an index of pointers to these blocks. In this case, random access will be just as fast as contiguous allocation, since that index can be stored in memory. Sequential access will not suffer too much if the number of contiguous blocks is small (hence each block is large). But there is a potential for external fragmentation, and it may be difficult to grow files.

You received 8 points for listing trade offs between contiguous and linked list allocation, and 4 points for being able to describe and characterize the use of extents. You did not need to comprehensively analyze the trade offs to receive credit.

6. I've said that CPU run times for processes are highly skewed. What does that mean? What does that imply for scheduling algorithms? Suppose that the distribution of CPU run times had a low variance; i.e. was centralized. In that case, what would be the characteristics of good (realizable) scheduling algorithms? In the latter case, please rank the performance (in terms of lowest mean flow time being best) of FIFO, RR and SET. (14)

(2 points) CPU run times being highly skewed means that there is a large variation in run times over all jobs. In practice, there are many small jobs and significantly fewer very long jobs.

(3 points) We don't want small jobs to have to wait for long jobs to complete. The less time a job has run, the less time it is

likely to take before it completes. Thus, we want to give priority
to jobs that have not run for very long,
since those jobs are most likely to finish soon.

(3 points) Good scheduling algorithms for a centralized distribution
of CPU run times would run jobs to completion once they were started.
The longer a job has run, the more likely it is to finish soon, so
replacing a long-running job with a short-running job on the CPU will
increase flow times.

(6 points) FIFO, RR, SET.
FIFO is best because it runs jobs to completion without taking them off
the CPU. RR is worse because it time-slices equally between all jobs,
so earlier jobs will have to give up the CPU to later jobs and that will
cause them to finish later. SET is worst because it gives preference to
the newest jobs, preventing the long-running jobs from completing. With
SET, all jobs will complete around the same time. For this question, the
order FIFO, SET, RR got 3 points of partial credit. Any other order got
0 points.

Note: Many people used the amount of overhead as their sole criterion
for evaluation. While this may be valid (depending on the amount of
overhead), it misses the point that it's better to run the longest-running
jobs when you have a centralized distribution of CPU times. I gave partial
credit for overhead-only answers. Overhead is a major consideration
only if it is quite large.

7. What is a stack algorithm? (Your explanation should mention and
define the inclusion property). For each of the following paging algorithms,
(LRU, MIN, OPT, FIFO, Clock, Working Set, Page Fault Frequency)
say whether it is a stack algorithm or not. (HINT: in my opinion, the
ones that are stack algorithms are obviously so; the rest are not.)
For those that are stack algorithms, explain why (i.e. give an
informal proof or a convincing argument). (For those that are not stack
algorithms, you do not have to prove it or explain why not.)

(3 points) A stack algorithm is one that satisfies the inclusion property.
The inclusion property states that, at time t, the contents of memory
of size k pages is a subset of the contents of memory of size k+1 pages.
In other words, running the same algorithm with more pages will never
increase the number of page faults.

(2 points for each correct identification)

LRU	stack
MIN	stack
OPT	stack
FIFO	not stack
Clock	not stack
Working Set	stack
Page Fault Frequency	not stack

(.5 points for each justification)
In LRU, the k most recently accessed (unique) pages is always a subset of the k+1 most recently accessed (unique) pages.

MIN (= OPT), works like LRU, only looking forward in time instead of backwards. In particular, we can prove this by induction. Assume that the inclusion property holds at time t. Then any replacement from a memory of size k+1 will also be evicted from a memory of size k (if it is there), thus maintaining the inclusion property at time t+1.

Working set keeps the set of pages which were accessed in the last tau time intervals. This is obviously a subset of the set of pages accessed in the last tau+1 time intervals.

Note: This problem asked you to say for each algorithm whether is was a stack algorithm or not. A good number of people lost points because they did not explicitly state this for some algorithms.