2018 Operating System Examination Solution

1. a. The definition of `struct sem_t` is as follows

```
struct sem_t {
  process_t** waiters; // a queue of processees waiting for this semaphore
  int counter; // the counter initialized upon creaetion
}
```

The counter of the semaphore will be initialized to a value indicating the number of threads/processes that can go into the critical section. When the counter becomes 0, new processes calling on this semaphore will go into the `waiters` list where they will stay until the semaphor counter becomes bigger than 0.

b. `sem_init(&s, i)` will initialize a semaphore called `s` with its initial counter value being `i`. `sem_down(&s)` will try to decrease the counter of `s` if it is greater than 0; if not, the current process/thread will be added in the waiters list and become blocked until the counter increases. `sem_up(&s)` will increase the counter of `s` by 1 and wake up one of the thread/process in the `waiters` list.

c. The thread-safe version of the code is shown below

```
1  int buf[BUF_SIZE];
2  unsigned int next_write = 0;
3  unsigned int next_read  = 0;
4  sem_t s;
5  sem_t producer;
6  sem_t consumer;
7
8  sem_init(&s, 1);  // ensure mutex
9  sem_init(&producer, BUF_SIZE);  // ensure buffer is not full/overflown
10 sem_init(&consumer, 0);  // ensure buffer is not empty
11
12 void writer(int value) {
13   sem_down(&producer);
14   sem_down(&s);
15   buf[next_write] = value;
16   sem_up(&s);
17   sem_up(&consumer);
18   next_write = (next_write + 1) % BUF_SIZE;
19 }
20
21 int reader() {
22   sem_down(&consumer);
23   sem_down(&s);
24   int r = buf[next_read];
25   sem_up(&s);
26   sem_up(&consumer);
```

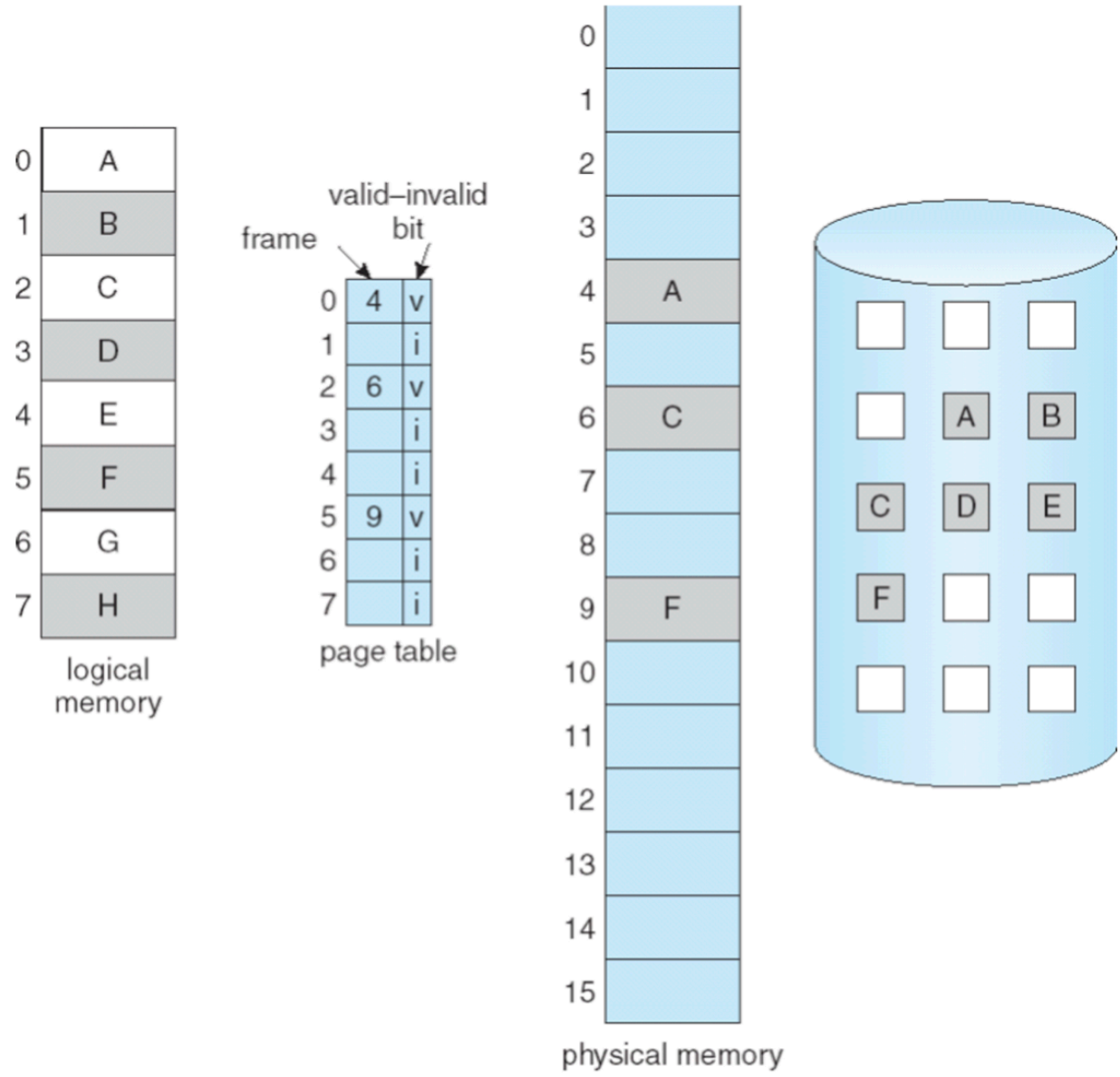
```
27    next_read = (next_read + 1) % BUF_SIZE;
28    return r;
29 }
```

d. The minimum number of lines to swap in order to introduce a deadlock is 1. We can swap either line 13 and 14, or line 22 and 23. When the buffer is full, swapping line 13 and 14 will cause a deadlock since instead of downing `producer` first, it will down `s` which blocks the reader from `sem_up(&producer)`. When the buffer is empty, swapping line 22 and 23 will also cause a deadlock as the `reader()` will first acquire `s` and then wait for `writer()` to `sem_up(&consumer)`, but `writer()` is blocked by `reader()`'s `sem_down(&s)` in the first place.

e. In this case, we can directly manipulate semaphore in the user mode without trapping into the kernel mode. Since a semaphore is only useed in threads within the same process, there is no need for inter-process communication in terms of semaphore. Hence we can localize the semaphore operation within process: increase/decrease or check the value of counter in the user mode and only keep a list of threads (instead of process) that waiting on the semaphore.

2. a. There are two main goals in memory management: 1️⃣ ensure a fair memory allocation strategy so that the memory is used efficiently and have less fragmentation 2️⃣ ensure that memory is not corrupted, e.g. not corrupted in a race condition.

b. The figure will look like the one below

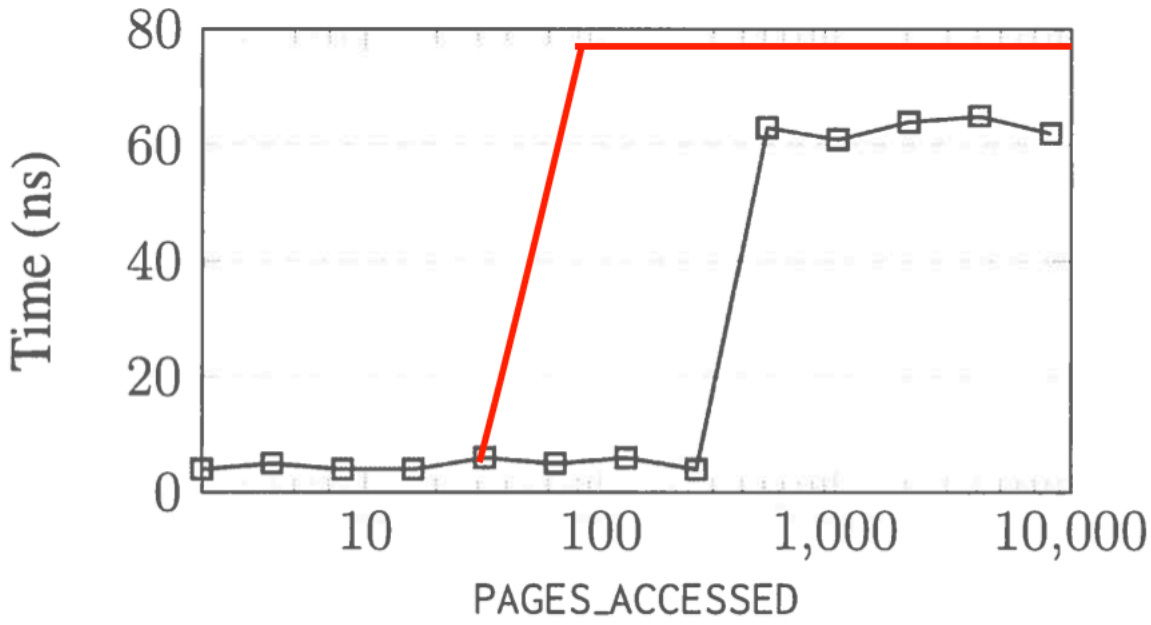


Demand paging is used to bring page into memory only when referenced. The operating system is responsible for maintaining the page table and the reference status of each page and ensure a fair swapping algorithm. Initially, all page has their reference bit set to 0. When a page is referenced but just not in memory, it will be brought from disk storage to memory and the reference bit set to valid. If the page is non-existent, then the operating system simply aborts the operation. When the memory is full, the operating system needs to select a victim to swap out to disk and swap in the correct page into the memory. The disk is responsible for storing the pages that are not recently referenced, which allows the memory to store less unnecessary pages.

c. i) When `PAGES_ACCESSED` grows to a certain value, all the frames in the main memory will be occupied and there is no free frame for new pages. Hence, the page replacement (a.k.a swapping) begins to work, which increases the time to execute line 5 since it needs more time to swap out the old page and bring the new page into the memory.

ii) From the plot, we know that the size of the main memory is around $10^{2.5} \cdot 4 = 316 \cdot 4 \approx 1264$ KB. $10^{2.5}$ comes from the fact that the line begins to rise in around the midpoint between `PAGES_ACCESSED` being 10^2 and 10^3. We also know that the memory access time is below 10 ns and the swapping time takes about 60 ns.

iii) The plateau part will rise to a higher value and the rising part will occur earlier, which result in the red line shown below.



The reason of earlier rising is that when n different processes execute concurrently, they will share the memory space together, which results in less memory space per process. This will let swapping occur earlier. The reason of higher plateau is that swapping has a speed upper limit. When swaping occurs more frequently, processes might need to queue up to "wait to be swapped".

iv) Hardware solutions: increase memory size, upgrade disk with a faster data transfer speed.

Software solutions: optimize scheduler to let each process run longer (i.e. increase the quantum in a round-robin scheduler), implement better swapping/page replacement algorithm, such as second chance page replacement.