ICS Homework 13

June 6, 2020

1 Organization

1.1

Let's consider about a page-removal algorithm: clock algorithm. Suppose we have a primary device which has 3 physical blocks, every time a reference string P come, it will follow the pseudo-code:

```
if hit(P)
    res_block = block contains P
    referenced_bit[res_block] <- True
    clock_arm does not change
else
    while referenced_bit[clock_arm]
        referenced_bit[clock_arm] <- False
        clock_arm <- next block
    res_block = block[clock_arm]
    referenced_bit[clock_arm] <- True
    clock_arm <- next block
return res_block
```

Fill in the table (If you don't know what to fill just write down a '-'). Note: '$*$' means the position of the clock arm; you also need to tell what the referenced bit is for each page block at that time (1 for True and 0 for False).

Time	0	1	2	3	4	5	6	7	8
Reference string	-	3	4	2	6	4	3	7	4
Primary Device Contents	* - -	3 * -	3 4 *	3 * 4 2	6 4 * 2	6 4 * 2	6 * 4 3	6 7 3 *	4 7 * 3
Referenced Bit	0 0 0	1 0 0	1 1 0	1 1 1	1 0 0	1 1 0	1 0 1	0 1 1	1 1 0
Page Absent	-	Y	Y	Y	Y	N	Y	Y	Y

1.2

Please consult the related information. Find and complement the definition of struct vm_area_struct in Linux v5.7.

(https://elixir.bootlin.com/linux/v5.7/source/include/linux/mm_types.h#L297)

```c
struct vm_area_struct {
	/* The first cache line has the info for VMA tree walking. */

	unsigned long vm_start;   /* Our start address within vm_mm. */
	unsigned long vm_end;   /* The first byte after our end address
						within vm_mm. */

	/* linked list of VM areas per task, sorted by address */
	struct vm_area_struct *vm_next, *vm_prev;

	struct rb_node vm_rb;

	/*
	 * Largest free memory gap in bytes to the left of this VMA.
	 * Either between this VMA and vma->vm_prev, or between one of the
	 * VMAs below us in the VMA rbtree and its ->vm_prev. This helps
	 * get_unmapped_area find a free area of the right size.
	 */
	unsigned long rb_subtree_gap;

	/* Second cache line starts here. */

	struct mm_struct *vm_mm; /* The address space we belong to. */

	/*
	 * Access permissions of this VMA.
	 * See vmf_insert_mixed_prot() for discussion.
	 */
	pgprot_t vm_page_prot;
	unsigned long vm_flags;   /* Flags, see mm.h. */

	/*
	 * For areas with an address space and backing store,
	 * linkage into the address_space->i_mmap interval tree.
```

```
	 */
	struct {
		struct rb_node rb;
		unsigned long rb_subtree_last;
	} shared;

	/*
	 * A file's MAP_PRIVATE vma can be in both i_mmap tree and anon_vma
	 * list, after a COW of one of the file pages.  A MAP_SHARED vma
	 * can only be in the i_mmap tree. An anonymous MAP_PRIVATE, stack
	 * or brk vma (with NULL file) can only be in an anon_vma list.
	 */
	struct list_head anon_vma_chain; /* Serialized by mmap_sem &
					  * page_table_lock */
	struct anon_vma *anon_vma;	/* Serialized by page_table_lock */

	/* Function pointers to deal with this struct. */
	const struct vm_operations_struct *vm_ops;

	/* Information about our backing store: */
	unsigned long vm_pgoff;		/* Offset (within vm_file) in PAGE_SIZE
					   units */
	struct file * vm_file;		/* File we map to (can be NULL). */
	void * vm_private_data;		/* was vm_pte (shared mem) */

#ifdef CONFIG_SWAP
	atomic_long_t swap_readahead_info;
#endif
#ifndef CONFIG_MMU
	struct vm_region *vm_region;	/* NOMMU mapping region */
#endif
#ifdef CONFIG_NUMA
	struct mempolicy *vm_policy;	/* NUMA policy for the VMA */
#endif
	struct vm_userfaultfd_ctx vm_userfaultfd_ctx;
```

```
} __randomize_layout;
```

2 System Software

2.1

Let p denote the number of producers, c the number of consumers, and n the buffer size in units of items. Consider the following buffer implementation. For each of the following scenarios, indicate whether the **mutex** semaphore is necessary or not to implement function **sbuf_insert** and **sbuf_remove**.

```c
typedef struct {
  int *buf;      /* Buffer array */
  int n;         /* Maximum number of slots */
  int front;     /* buf[(front+1)%n] is first item */
  int rear;      /* buf[rear%n] is last item */
  sem_t mutex;   /* Protects accesses to buf */
  sem_t slots;   /* Counts available slots */
  sem_t items;   /* Counts available items */
} sbuf_t;
```

A. $p = 1, c = 1, n > 1$
B. $p = 1, c = 1, n = 1$
C. $p > 1, c > 1, n = 1$
ANS:

A. $p = 1, c = 1, n > 1$: No. Using semaphore **slots** and **items** is enough, and there is no concurrent access to **front** and **rear**.

B. $p = 1, c = 1, n = 1$: No. This case is the same as case A.

C. $p > 1, c > 1, n = 1$: No. In this case, **front** and **rear** are always 0. Consumers will be waiting for **items**, and producers will be waiting for **slots**.