Pregunta 1

I. (3 puntos) Considere un procesador con una MMU que *no implementa* el bit D (*dirty*). Modifique la implementación de la estrategia del reloj que aparece en el reverso de este enunciado de manera que no se grabe una página en disco si esa página ya había sido grabada previamente. No copie toda la implementación, coloque sólo las líneas que modificó más 2 líneas antes y 2 líneas después de la modificación.

Ayuda: Use el bit W para emular el bit D y suponga que existe un bit de software W2 que es el que realmente indica si una página se puede escribir o no. Note que el hardware también gatilla un page-fault cuando se escribe en una página cuyo bit V es 1, pero su bit W es 0.

II. (1 punto) El proceso A posee código en el intervalo de direcciones virtuales [20 KB, 32 KB[, datos en [32 KB, 40 KB] y pila en [56 KB, 64 KB[. El proceso B posee código en [12 KB, 20 KB[, datos en [20 KB, 28KB[y pila en [60 KB, 64 KB[. Confeccione las tablas de páginas de ambos procesos, indicando número de página real y atributos V y W. La máquina posee 128 KB de memoria real organizada en páginas de 4 KB. Decida Ud. la asignación de páginas reales.

III. (1 punto) Explique qué sucede cuando el proceso B desborda su pila.

IV. (1 punto) Explique qué sucede cuando se invoca *malloc* en el proceso A solicitando 100 bytes de memoria, pero esa cantidad de memoria no está disponible en su área de datos actual.

Pregunta 2

a.- (1 punto) Considere el siguiente diagrama de lecturas y escrituras (r, w) en memoria de un proceso Unix en un sistema que usa la estrategia del *working set*. Las filas corresponden a las páginas (0, 1, 2, …) y las columnas a los intervalos de cálculo del working set (A, B, C, …).

página								
	6	w r	r	r		r		
	5	w r r	r	r			r	
	4	w r		r		r w		r
	3	w r r			r r			r r r
	2	w r			r w			r r
	1	w	r	r	w r	r w	w	r
	0	w r	r	r		w r	r	r
		A	B	C	D	E	F	G

Tiempo

(i) Indique qué accesos a las páginas 4 y 5 podrían producir un *page-fault* (use como notación 3, G, 1er acceso). (ii) Suponiendo que se usa el bit D (*dirty*) para optimizar los reemplazos y que producto de un *page-fault* de otro proceso las página 2 y 3 fueron escogidas para ser reemplazadas en los intervalos C y F, indique para cada uno de esos 4 reemplazos si fue necesario grabar la página en el área de paginamiento para hacer el reemplazo.

b.- (1 punto) El sistema operativo de los smartphones no implementa paginamiento en demanda. Explique entonces cuál es la *principal* utilidad de los espacios de direcciones virtuales en un smartphone.

c.- (1 punto) Se argumenta que el paginamiento es lento porque por cada acceso a la memoria se debe hacer un acceso adicional en la tabla de páginas para traducir la dirección virtual a una dirección real. ¿Qué respondería Ud.? Explique

d.- (3 puntos) El reverso de este enunciado incluye el driver simplicado de una memoria de 8 KB (*/dev/mem*). Modifique la función *mem_write* de manera que los escritores esperen hasta formar un trío. El siguiente es un ejemplo de uso. El prompt $ indica el momento exacto en que termina el comando. Lo que ingresó el usuario está en **negritas**.

shell 1	***shell 2***	***shell 3***
$ **echo hola > /dev/mem**		
	$ **echo que > /dev/mem**	
$	$	$ **echo tal > /dev/mem** $
$ **echo otra > /dev/mem**		
	$ **echo vez > /dev/mem**	
$	$	$ **echo igual > /dev/mem** $

Restricciones: Debe usar semáforos para la sincronización. El orden de atención no es FIFO.

La estrategia del reloj para un núcleo clásico monocore

```
// Se invoca cuando ocurre un pagefault,
// es decir bit V==0 o el acceso fue una escritura y bit W==0
void pagefault(int page) {
  Process *p= current_process; // propietario de la página
  int *ptab= p->pageTable;
  if (bitS(ptab[page]))          // ¿Está la página en disco?
    pageIn(p, page, findRealPage()); // sí, leerla de disco
  else
    segfault(page);              // no
}
// Graba en disco la página page del proceso q
int pageOut(Process *q, int page) {
  int *qtab= q->pageTable;
  int realPage= getRealPage(qtab[page]);
  savePage(q, page);  // retoma otro proceso
  setBitV(&qtab[page], 0);
  setBitS(&qtab[page], 1);
  return realPage;     // Retorna la página real en donde se ubicaba
}
// Recupera de disco la página page del proceso p colocándola en realPage
void pageIn(Process *p, int page, int realPage) {
  int *ptab= p->pageTable;
  setRealPage(&ptab[page], realPage);
  setBitV(&ptab[page], 1);
  loadPage(p, page);  // retoma otro proceso
  setBitS(&ptab[page], 0);
  purgeTlb();          // invalida la TLB
  purgeL1();           // invalida cache L1
}
Iterator *it; // = processIterator();
Process *cursor_process= NULL;
int cursor_page;

int findRealPage() {
  // recorre las páginas residentes en memoria de todos los procesos buscando una
  // página que no pertenezca a ningún working set y que no haya sido referenciada
  int realPage= getAvailableRealPage();
  if (realPage>=0)       // ¿Quedan páginas reales disponibles?
    return realPage;     // Sí, retornamos esa página
  // no, hay que hacer un reemplazo
  for (;;) {
    if (cursor_process==NULL) { // ¿Quedan páginas en proceso actual?
      // no
      if (!hasNext(it))       // ¿Quedan procesos por recorrer?
        resetIterator(it);    // partiremos con el primer proceso nuevamente
      cursor_process= nextProcess(it);         // pasamos al próximo
      cursor_page= cursor_process->firstPage; // primera pág.
    }
    // Estamos visitando la página cursor_page del proceso cursor_process
    int *qtab= cursor_process->pageTable;
    // mientras queden páginas por revisar en cursor_process
    while (cursor_page<=cursor_process->lastPage) {
      if (bitV(qtab[cursor_page])) {    // ¿Es válida?
        if (bitR(qtab[cursor_page]))    // no fue referenciada
          setBitR(&qtab[cursor_page], 0);
        else // sí, se reemplaza la página cursor_page de cursor_process
          return pageOut(cursor_process, cursor_page++);
      }
      cursor_page++;
    }
    // Se acabaron las páginas de cursor_process,
    // hay que buscar en el próximo proceso
    cursor_process= NULL;
  }
}
```

Driver simplicado para una memoria de 8 KB

```
#define MAX 8192                // Tamaño máximo (8KB)
static struct semaphore mutex; // Con un 1 ticket inicial
static char mem_buf[MAX];       // Contenido de la memoria
static int size;                // Tamaño real almacenado

ssize_t mem_write(struct file *filp, char *buf,
                  size_t count, loff_t *f_pos) {
  down(&mutex);
  loff_t last= *f_pos+count;
  if (last>MAX) count -= last-MAX;
  copy_from_user(mem_buf+*f_pos, buf, count);
  *f_pos += count;
  size= *f_pos;
  up(&mutex);
  return count;
}

ssize_t mem_read(struct file *filp, char *buf,
                 size_t count, loff_t *f_pos) {
  down(&mutex);
  if (count>size-*f_pos) count= size-*f_pos;
  copy_to_user(buf, mem_buf+*f_pos, count);
  *f_pos += count;
  up(&mutex);
  return count;
}
```