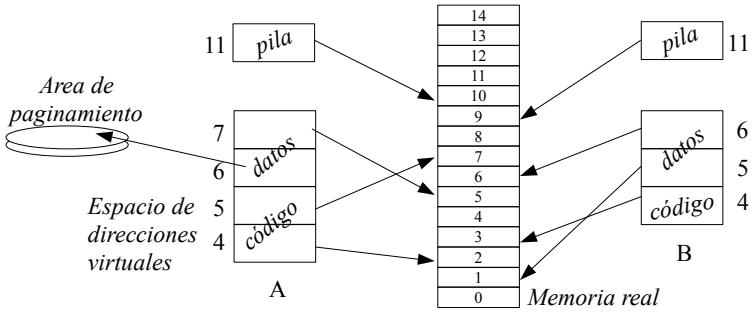
CC4302 Sistemas Operativos – Control 3 – Semestre Primavera 2017 – Prof.: Luis Mateu

Pregunta 1

I. Considere un procesador con una MMU que *sí implementa* el bit D (*dirty*). Modifique la implementación de la estrategia del reloj que aparece en el reverso de este enunciado de manera que use eficientemente el bit D. No copie las funciones que no cambian. Escriba completamente las funciones que sí modificó.

II. El diagrama de más abajo muestra la asignación de páginas en un sistema Unix que ejecuta los procesos A y B. Las páginas son de 4 KB. El núcleo utiliza la estrategia *copy-on-write* para implementar *fork*. (a) Construya la tabla de páginas del proceso A después de que este invoca *sbrk* pidiendo 10 KB adicionales. (b) Considere que el proceso B invocó *fork*. Construya la tabla de páginas para el proceso hijo justo después de que este modificó la página 11. No construya la tabla del padre. En las tablas indique página virtual, página real y atributos de validez y escritura.



III. 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

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

Tiempo

(a) Indique el valor del atributo *Referenced* para todas las páginas al inicio del intervalo E y al final de ese intervalo. (b) Indique para los períodos C a F qué accesos pueden producir *page-faults*. Utilice coordenadas del estilo *(G, 4, 1er. acceso)*. (c) Suponga que las páginas 2 y 3 son reemplazadas en C. En F es necesario reemplazarlas nuevamente. Explique si se deberán escribir nuevamente en disco considerando un sistema de paginamiento optimizado.

Pregunta 2

a.- (1,5 puntos) Se tiene un archivo de 397 KB en una partición Unix con bloques de 4 KB. Haga un diagrama mostrando inodo, bloques de datos y de indirección. Conteste además: ¿Cuanto espacio en disco se ocupa realmente para almacenar este archivo?

b.- (1,5 puntos) Considere una partición con bloques de 4 KB en un disco duro tradicional con un tiempo de acceso de 10 milisegundos y una velocidad de 100 MB/seg. Estime cuanto tomaría leer 100 archivos de 4 KB y cuanto tomaría leer un solo archivo de 400 KB. Repita su estimaciones considerando ahora un *solid state drive* (SSD) capaz de realizar 50000 IOPS (*i/o operations per second*) y velocidad de 500 MB/seg.

c.- (3 puntos) Modifique el driver simplicado de una memoria de 8 KB (*/dev/mem*) incluido en el reverso de este enunciado de manera que se comporte exactamente como el siguiente 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***	***shell 4***
$ **echo hola > /dev/mem**			
	$ **echo que > /dev/mem**		
		$ **echo tal > /dev/mem**	
$	$	$	$ **cat /dev/mem** hola que tal $
$ **echo como > /dev/mem**			
	$ **echo estas > /dev/mem**		
$	$		$ **cat /dev/mem** como estas $

Restricción: Debe usar semáforos para la sincronización.

Ayuda

No se preocupe por usos distintos al ejemplo. En particular ignore las señales. El comando *echo* invoca *write* una sola vez. No programe *open* ni *release*.

En *mem_write* haga esperar a los procesos escritores en un arreglo de 3 semáforos (en el ejemplo tendrá que hacer esperar a lo más 3 procesos). El driver original *borra* el contenido previo. Su driver modificado debe *agregar* al contenido previo. Esto significa que en *mem_write* debe ignorar **f_pos*.

En *mem_read* sí considere *f_pos. Cuando detecte que se agotó el contenido previo retorne 0 bytes (para hacer que *cat* termine), despierte a los procesos en espera y borre el contenido previo.

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
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;
}
```