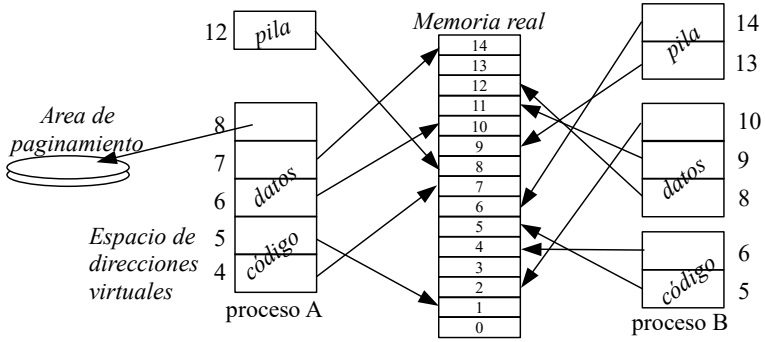
CC4302 Sistemas Operativos – Control 3 – Semestre Otoño 2024 – Profs.: Mateu, Torrealba, Arenas

Pregunta 1

A) (3 puntos) **Modifique** la implementación de la estrategia del *working set* que aparece en el reverso de este enunciado de manera que funcione correctamente con una MMU que *no implementa* el bit R (*reference*). No copie toda la implementación, escriba sólo las líneas que modificó más 2 líneas antes y 2 líneas después de la modificación.

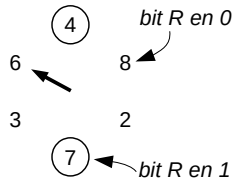
Ayuda: use astutamente el bit de validez (V) en su reemplazo, agregando otro bit VE que indique la validez efectiva de una página, con funciones *setBitVE* para establecer el valor de VE y *bitVE* para leer su valor.

B) (3 puntos) 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 2 KB adicionales. (b) A continuación el proceso B invocó *fork*. Construya la tabla de páginas para el proceso hijo justo después de que este modificó la página virtual 8. No construya la tabla del padre. En las tablas indique página virtual, página real y atributos de validez y escritura.



Pregunta 2

A) (2 puntos) Considere un sistema Unix que implementa la estrategia del reloj. El sistema posee 6 páginas reales disponibles y corre un solo proceso. La figura indica el estado inicial de la memoria, mostrando las páginas residentes en memoria, la posición del cursor y el valor del bit R.



Dibuje los estados por los que pasa la memoria para la siguiente traza de accesos a páginas virtuales: 7, 4, 6, 5, 7, 3, 9.

B) (4 puntos) La función *void ocupar( )* solicita ocupar de manera exclusiva un recurso compartido por múltiples threads. Si otro thread está ocupando el recurso, está función debe esperar. La función *void desocupar( )* notifica que ya no se necesita el recurso. Si hay threads esperando el recurso, uno de ellos puede ocuparlo.

Programe *ocupar* y *desocupar* usando spinlocks para la sincronización. El recurso debe ser otorgado ***por orden de llegada***. Este problema fue resuelto en cátedra usando mutex, condiciones y el patrón request.

Pregunta 3

A) (4 puntos) La siguiente es la parte relevante de la implementación del driver para el dispositivo incógnito */dev/inc*:

```
static struct semaphore sr, sw;
static char inc_buf[8192];
static int size;
static ssize_t inc_read(
    struct file *filp, char *buf,
    size_t cnt, loff_t *f_pos) {
  down(&sr);
  if (cnt>size)
    cnt= size;
  copy_to_user(buf, inc_buf, cnt);
  inc_buf[0] += 1;
  up(&sw);
  return cnt; // ¡Nunca retorna 0!
}
```

```
int inc_init(void) {
  …
  sema_init(&sr, 0);
  sema_init(&sw, 0);
}
static ssize_t inc_write(
    struct file *filp, char *buf,
    size_t cnt, loff_t *f_pos) {
  copy_from_user(inc_buf, buf, cnt);
  size= cnt;
  up(&sr); up(&sr); up(&sr);
  down(&sw);
  return cnt;
}
```

Este driver no tiene mucho sentido, pero lo que interesa es determinar qué hace. La siguiente tabla es un ejemplo de uso incompleto. Complete la tabla con el mensaje que despliega cada comando e indique con el *prompt* $ el momento exacto en que termina el comando (si es que termina).

shell 1	*shell 2*
`$ echo "abcd" > /dev/inc`	
	`$ cat < /dev/inc`
`$ echo "efgh" > /dev/inc`	

B) (2 puntos) Un programador propone hackear un sistema Unix de la siguiente manera. El sabe en qué dirección de la memoria está el código que implementa la llamada al sistema *read*. Entonces, *ejecutando en modo usuario*, él propone reescribir esa parte de la memoria con su propio código para hacer maldades en el núcleo cuando algún proceso invoque *read*. Explique si puede o no lograr hackear la máquina.

Implementación de la estrategia del working set para un núcleo clásico monocore:

```
// Se invoca para recalcular el working set
void computeWS(Process *p) {
  int *ptab= p->pageTable;
  for (int i= p->firstPage; i<p->lastPage; i++) {
    if (bitV(ptab[i]) {          // ¿Es válida?
      if (bitR(ptab[i])) {       // ¿Fue referenciada?
        setBitWS(&ptab[i], 1); // Sí, se coloca en el working set
        setBitR(&ptab[i], 0);
      }
      else {
        setBitWS(&ptab[i], 0); // No, se saca del working set
} } } }
// 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 q 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 needsSwapping= 0; // Se necesita para no caer en ciclo infinito
  for (;;) {
    int realPage= getAvailableRealPage();
    if (realPage>=0)       // ¿Quedan páginas reales disponibles?
      return realPage;     // Sí, retornamos esa página
    // no, hay que hacer un reemplazo
    if (cursor_process==NULL) {// ¿Quedan páginas en proceso actual?
      // no
      if (!hasNext(it)) {    // ¿Quedan procesos por recorrer?
        // no
        if (needsSwapping) { // ¿Segunda vez que pasamos por aquí?
          // sí, revisamos todos los procesos sin encontrar reemplazo posible
          doSwapping();      // hacemos swapping
          needsSwapping= 0;
          continue;          // recomenzamos el ciclo for(;;)
        }
        resetIterator(it);   // partiremos con el primer proceso nuevamente
        // Si pasamos otra vez por acá haremos swapping
        needsSwapping= 1;
      }
      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?
           !bitWS(qtab[cursor_page]) && // ¿no está en WS?
           !bitR(qtab[cursor_page]) ) { // no fue referenciada
        // 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;
} }
```