Sistemas Operativos

Auxiliar Recuperativo
Spinlocks,
Módulos Linux,
Estrategias de páginamiento

Pablo Jaramillo
Vicente González

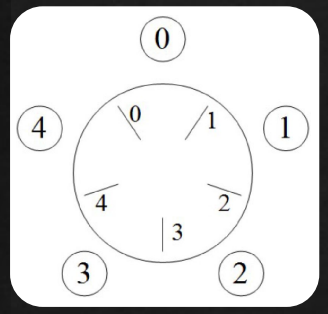
Problemas

Problema 1: Spinlocks

A la continuación se muestra una solución parcial de la cena de filósofos, en donde 5 filósofos numerados de 0 a 4 cenan juntos en una mesa redonda en donde hay 5 palitos, numerados de 0 a 4.

```
void pedir(int id);
void devolver(int id);
void init( );
void filosofo(int id) {
  for (;;) {
    pedir(id);
    comer(id,(id+1)%5);
    devolver(id);
    pensar();
  }
}
```

Cada filósofo alterna entre comer y pensar. Para poder comer el filósofo id necesita los palitos id y (id+1)%5. No necesita ningún palito para pensar. Los filósofos deben comer en paralelo cuando sea posible pero no deben comer con el mismo palito simultáneamente, y tampoco pueden sufrir hambruna.

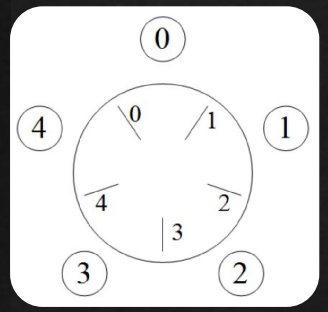


Problema 1: Spinlocks

Ud. debe programar las funciones pedir y devolver. Programe también init para inicializar las variables globales que necesite. Para evitar la hambruna se exige que los filósofos coman por orden de llegada. Use la cola Queue para hacer esperar a los filósofos. Recuerde que la función peek le permite obtener el primer elemento de la cola, sin extraerlo.

```
void pedir(int id);
void devolver(int id);
void init( );
void filosofo(int id) {
  for (;;) {
    pedir(id);
    comer(id,(id+1)%5);
    devolver(id);
    pensar();
  }
}
```

Por ejemplo si el filósofo 1 está comiendo y 0 llama a pedir, 0 deberá esperar porque el palito 1 está ocupado. Si a continuación 3 llama a pedir también deberá esperar porque llegó después que 0. Finalmente, el filósofo 1 termina de comer e invoca devolver. Entonces 0 y 3 deberán comer en paralelo. Observe que la función devolver puede despertar 0, 1 o 2 filósofos.



Restricción: Ud. debe usar el patrón request para evitar cambios de contexto innecesarios. Para la sincronización debe usar spinlocks

P1 (ii) - EX - 2023O

Resumen

Páginamiento

Páginamiento

Conceptos

1. MMU: “Memory Management Unit”, estrategia para manejar el páginamiento
2. Page fault: Cuando se intenta acceder a una página que no está en memoria, causa que el MMU deba obtener la página del disco y llevarla a memoria
3. Copy-on-Write: Cualidad a páginas protegidas contra modificaciones, indica que para modificar la página se debe primero hacer una copia para el proceso que quiere escribir.
4. Swapping: Pasar todas las páginas de un proceso al disco para traer páginas para otro (o el mismo) proceso. Es el último recurso en páginamiento, es muy lento en particular cuando se hace repetidas veces.

Páginamiento

Conceptos

5. Thrashing: Cuando ocurren demasiados page-faults. Lleva al computador a dedicar demasiado tiempo al disco y no a la CPU. Esto es lo que nos “congela el computador”.
6. Working Set: Estrategia para evitar el swapping, maneja conjuntos de páginas recientemente accedidas para los procesos para determinar cuáles se pueden llevar a disco.
    a. Tiene el sobrecosto de instanciar y mantener los conjuntos para cada proceso.
    b. Evita el thrashing y reduce cantidad de page-faults al utilizar swapping de vez en cuando.
7. Estrategia del reloj: Estrategia para evitar el swapping, es un algoritmo que cicla entre punteros a páginas marcando el bit R para evitar mover páginas a memoria.
    a. Tiende a buenos resultados al mover páginas que no se han utilizado en harto tiempo, pero hace muchos page-faults.
    b. Tiene 0 sobrecosto si sobra memoria.

Bits de páginas

bit V	
bit R	
bit W	
bit D	

Bits de páginas

bit V	“Valid” : La página está en memoria, si es 0 está en el disco.
bit R	
bit W	
bit D	

Bits de páginas

bit V	“Valid” : La página está en memoria, si es 0 está en el disco.
bit R	“Referenced” : La página ha sido recientemente utilizada, si es 0 es candidata a ser sacada de memoria.
bit W	
bit D	

Bits de páginas

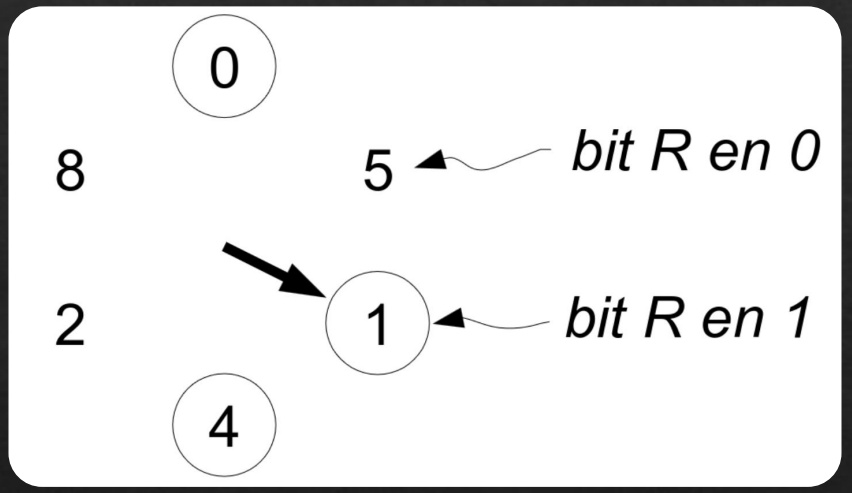
bit V	“Valid” : La página está en memoria, si es 0 está en el disco.
bit R	“Referenced” : La página ha sido recientemente utilizada, si es 0 es candidata a ser sacada de memoria.
bit W	“Write” : La página puede ser modificada por el proceso, si es 0 debe crearse una copia y escribir en esa.
bit D	

Bits de páginas

bit V	“Valid” : La página está en memoria, si es 0 está en el disco.
bit R	“Referenced” : La página ha sido recientemente utilizada, si es 0 es candidata a ser sacada de memoria.
bit W	“Write” : La página puede ser modificada por el proceso, si es 0 debe crearse una copia y escribir en esa.
bit D	“Dirty” : La ha sufrido modificaciones, si es 0 puede eliminarse de la memoria, si es 1 debe ser guardada antes de salir de la memoria.

Problema 2: Estrategia del reloj

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



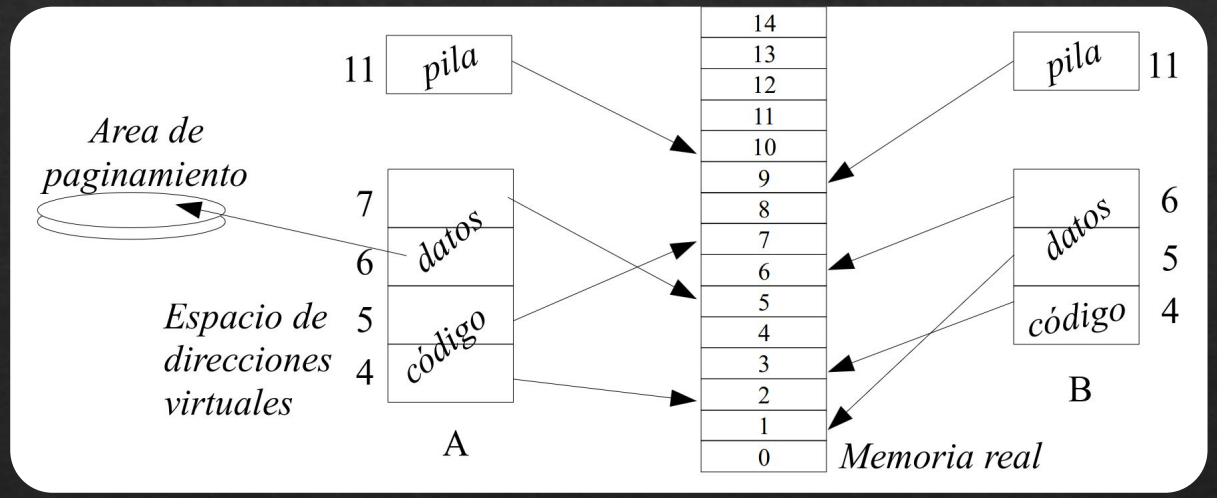
Dibuje los estados por los que pasa la memoria para la siguiente traza de accesos a páginas virtuales:

8 6 0 1 7 8

P3 (IV) - EX - 2023O

Problema 3: Diagramas de páginas

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.

P1 (II) - C3 - 2017P

Problema 4: MMU

En las siguientes diapositivas se muestra la implementación de la estrategia del working set.
Reimplemente la misma estrategia considerando una MMU que sí implementa el bit D (dirty) de manera que no se grabe una página en disco si esa página ya había sido grabada previamente.

Warning!
Implementacion incomming...

```c
// 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()
```

```c
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;
  }
}
```

Problema 5: Módulos

La siguiente es la parte relevante de la implementación del driver del dispositivo incógnito **`/dev/inc`**:

```
static char inc_buf, ult= 0;
static struct semaphore lleno;
static struct semaphore vacio;
static ssize_t inc_read(
    struct file *filp, char *buf,
    size_t count, loff_t *f_pos) {
  if (ult=='\n') {
    ult= 0;
    return 0;
  }
  down(&lleno);
  copy_to_user(buf, &inc_buf, 1);
  copy_to_user(buf+1, &inc_buf, 1);
  ult= inc_buf;
  up(&vacio);
  return 2;
}
```

```
int inc_init(void) {
  …
  sema_init(&lleno, 0);
  sema_init(&vacio, 1);
  …
}

static ssize_t inc_write(
    struct file *filp, char *buf,
    size_t count, loff_t *f_pos) {
  for (size_t i= 0; i<count; i++) {
    down(&vacio);
    copy_from_user(&inc_buf, buf, 1);
    up(&lleno);
  }
  return count;
}
```

Resumen

Módulos de Linux

Módulos de Linux

Los módulos son la interfaz por la cual los dispositivos se comunican con los sistemas operativos tal que se utilicen unas funciones básicas que logran abstraer el sistema operativo de la implementación del dispositivo.

Módulos de Linux

API

```
struct file_operations DRIVERNAME_fops = {
  read: DRIVERNAME_read,
  write: DRIVERNAME_write,
  open: DRIVERNAME_open,
  release: DRIVERNAME_release
};
```

La API mapea a esas 4 instrucciones básicas las funciones respectivas. Cuyas firmas están abajo.

Hay 2 funciones fundamentales más, estas son las de init y exit.

```
int DRIVERNAME_open(struct inode *inode, struct file *filp);
int DRIVERNAME_release(struct inode *inode, struct file *filp);
ssize_t DRIVERNAME_read(struct file *filp, char *buf, size_t count, loff_t *f_pos);
ssize_t DRIVERNAME_write(struct file *filp, const char *buf, size_t count, loff_t *f_pos);
void DRIVERNAME_exit(void);
int DRIVERNAME_init(void);
```

Módulos de Linux

Mapeo de init y exit

```
module_init(DRIVERNAME_init);
module_exit(DRIVERNAME_exit);
```

Print de mensajes

```
printk("<DRIVERNAME> algo ocurre\n");
```

Herramientas de sincronización

```
// Declaraciones e inicialización
static KMutex mutex;
static KCondition cond;
m_init(&mutex);
c_init(&cond);
// Funciones Mutex
m_lock(&mutex);
m_unlock(&mutex);
// Espera
c_wait(&cond, &mutex); // Retorna 0 si fue despertado, y != 0 si
                       // hubo un error
c_broadcast(&cond);
```

Módulos de Linux

Registro y termino de módulo

```
register_chrdev(DRIVERNAME_major, "DRIVERNAME", &DRIVERNAME_fops);
unregister_chrdev(DRIVERNAME_major, "DRIVERNAME");
```

Manejo de memoria

```
// equivalente a malloc, "queremos MAX_SIZE bytes desde el heap”
a_buffer = kmalloc(MAX_SIZE, GFP_KERNEL);  // a_buffer == NULL, si hay error
kfree(a_buffer);
/* Transfiriendo datos hacia el espacio del usuario: copia al buffer del usuario los datos del driver
desde la posición *f_pos, “count” bytes  */
copy_to_user(buf, a_buffer+*f_pos, count);  // retorna 0 si todo sale bien
/* Transfiriendo datos desde el espacio del usuario: desde la posición *f_pos,
al buffer del driver, “count” bytes */
copy_from_user(a_buffer+*f_pos, buf, count); // retorna 0 si todo sale bien
```

Módulos de Linux

Desarrollo de un dispositivo

Crear dispositivo	sudo mknod /dev/DRIVERNAME <type> <major> <minor>
Definir permisos	sudo chmod <permisos> DRIVERNAME.ko
Conectar al SO	sudo insmod $(PWD)/DRIVERNAME.ko
Printear mensajes	sudo dmseg | grep <string> | tail -n <N de msgs a ver>
Desconectar del SO	sudo rmmod DRIVERNAME.ko
Borrar dispositivo	sudo rm dev/DRIVERNAME

Problema 5: Módulos

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 > /dev/inc (1)`	
`$ echo abcd > /dev/inc (2)`	
	`$ cat /dev/inc (3)`
	`$ cat /dev/inc`
	`$ cat /dev/inc`
`$ echo ef > /dev/inc`	
`$ echo ghi > /dev/inc` *<control-C>*	

(1) El comando **echo** sin argumentos escribe solo un byte en su salida estándar: el newline (\n). En este caso su salida estándar se redirigió a **/dev/inc**.
(2) Se escriben 5 bytes: a, b, c, d y el newline.
(3) El comando **cat** abre su único parámetro (**/dev/inc**) con **open** y lee repetidamente con **read** arrojando lo leído a la salida estándar (el terminal en este caso) hasta que **read** retorne 0. Luego invoca **close** y termina.

P2 (c) - C3 - 2018O

Fin