Sistemas Operativos
Control 2

2 horas

Mayo de 2009

Pregunta 1 (memoria: conceptos)

Parte I

Un ingeniero opina que hoy en día, con tanta memoria disponible, ya no es buena idea paginar la memoria. Mejor sería volver al esquema de los segmentos de memoria contigua dinámicos porque no se necesita copiar páginas a disco al tener tanta RAM. Discuta esta afirmación revisando lo bueno y lo malo de los dos esquemas. ¿Cuál sería mejor hoy día?

Al no paginar la memoria, se ahorra algo de fragmentación interna y las enormes tablas de página para traducir. La indirección, al pasar por la tabla de segmentos, se paga igual. Lo más caro de un sistema de tamaños variables es la asignación y la compactación de la memoria. El hecho que no haya paginamiento a disco no es una razón importante para abandonar el paginamiento, es más bien el costo de la asignación.

Parte II

El tamaño de página en los computadores ha aumentado de 512 bytes a unos 4 Kbytes hoy en día, mientras la RAM total disponible ha aumentado, en el mismo periodo, de 1 Mbyte a 1 Gbyte. Discuta porqué no ha seguido el mismo ritmo en ambos casos, tratando de buscar razones tecnológicas y analizando los costos y beneficios de cada caso.

Esta diferencia parecería indicar que los programas no han crecido tanto en su consumo total de memoria y que probablemente se corren más programas en total en cada computador. El tener páginas más grandes haría más pequeñas las tablas de páginas y nos ahorraríamos las indirecciones. Pero tendríamos mayor fragmentación interna y menor precisión al traerlas a memoria.

Parte III

En un S.O. que implementa LRU como reemplazo de página, explique cual sería el patrón de acceso a su RAM que representa el peor caso para el algoritmo. ¿Cómo

podría defenderse un S.O. de este caso particular?

El peor caso es un acceso secuencial de las páginas, una detrás de otra y luego volver a la más antigua. De esa forma, la página más antigua siempre tiene mayor probabilidad de ser referenciada, al revés de lo que espera LRU. Un S.O. podría monitorear la tasa de falta de páginas de los procesos y detectar estos casos, anulando LRU para ellos.

Pregunta 2: memoria (implementación)

El código del kernel que atiende una falta de página es:

```
PageFault(int pid, int page)
{
    PAGETABLE *pagetable = procs[pid].pagetable;

    if(pagetable == NULL) error(BADPID); /* pid inexistente */

    if(!pagetable->page_array[page].invalid)
        return;

    if(pagetable->mempages < pagetable->wss) {
        fr = FindFreeFrame();
        pagetable->mempages++;
    }
    else {
        p = FindLRU(pagetable);
        if(pagetable->page_array[p].dirty) {
            pagetable->page_array[p].disk = swapout(pagetable[p].frame);
            pagetable->page_array[p].invalid = TRUE;
        }
        fr = pagetable->page_array[p].frame;
    }

    swapin(pagetable->page_array[page].disk, fr);
    pagetable->page_array[page].invalid = FALSE;
    pagetable->page_array[page].frame = fr;
    pagetable->page_array[page].dirty = FALSE;
    return;
}
```

Parte I

Escriba la función `FindFreeFrame()` de modo que retorne siempre un frame disponible. Suponga que la sumatoria de los WSS de los procesos siempre es menor o igual que los frames de la memoria RAM (NFRAMES). Si no hay frames desocupados, debe sacarle frames a otros procesos.

```
FindFreeFrame() {
```

```
for(i=0; i < NFRAMES; i++)
if(frame[i].free) {
    frame[i].free = FALSE;
         return(i);
}

/* Ningun marco libre */
for(pid=0; pid < MAXPROCS; pid++) {
    pagetable = procs[pid].pagetable;
    if(pagetable == NULL) continue;
    if(pagetable->wss < pagetable->mempages) {
        p = FindLRU(pagetable);
    fr = pagetable[p].frame;
    if(pagetable->page_array[p].dirty) {
                 pagetable->page_array[p].disk = swapout(fr);
    }
            pagetable->page_array[p].invalid = TRUE;
        pagetable->mempages--;
        return fr;
}

/* No puede ocurrir! */
return (out of mem);
```

Parte II

En el código anterior, buscamos una página candidata con LRU y luego vemos si está sucia o no para copiarla. Si consideramos que reemplazar una página sucia me cuesta el doble que una limpia, podríamos buscar primero con LRU en las limpias y solo si no tengo, buscar con LRU en las sucias.

¿Quedaría un buen algoritmo? Discuta.

El problema de esta solución es que no sería realmente LRU: podríamos sacar una página que no corresponde limpia, cuando hay páginas sucias que ya no serán usadas. Mejor sería elegir páginas limpias cuando LRU me indica que están en igualdad (o casi igualdad) de condiciones para ser elegidas.

Pregunta 3 (drivers)

Queremos hacer un device que sirva de semáforo. Para eso, la función read sirve como wait y la función write como signal. Suponga que siempre inicializamos el contador del semáforo en 1 al hacer open la primera vez, y luego lo incrementamos haciendo varios signals seguidos.

Se propone la siguiente implementación del driver, escrita por un muy mal alumno del curso:

```
ssize_t jbox_read(struct file *filp, char __user *buf, size_t count,
```

```
                loff_t *f_pos)
{
    struct jbox_dev *dev = filp->private_data;

    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;
    if (dev->sem_count == 0) {
        if (wait_event_interruptible(dev->p_queue, (dev->sem_count > 0)))
            return -ERESTARTSYS;
    }
    dev->sem_count--;
    up(&dev->sem);
    return 1;
}

ssize_t jbox_write(struct file *filp, const char __user *buf, size_t count,
                loff_t *f_pos)
{
    struct jbox_dev *dev = filp->private_data;

    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;
    dev->sem_count++;
    up (&dev->sem);
    wake_up_interruptible(&dev->p_queue);
    return count;
}
```

Revísela y corrija los errores.

```
ssize_t jbox_read(struct file *filp, char __user *buf, size_t count,
                 loff_t *f_pos)
{
    struct jbox_dev *dev = filp->private_data;

    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;
    while (dev->sem_count == 0) { // Cambio
up(&dev->sem); // Cambio
        if (wait_event_interruptible(dev->p_queue, (dev->sem_count > 0)))
            return -ERESTARTSYS;
if(down_interruptible(&dev->sem)) // Cambio
    return -ERESTARTSYS; // Cambio
    }
    dev->sem_count--;
    up(&dev->sem);
    return 1;
}
```

```
ssize_t jbox_write(struct file *filp, const char __user *buf, size_t count,
                loff_t *f_pos)
{
    struct jbox_dev *dev = filp->private_data;

    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;
    dev->sem_count++;
    up (&dev->sem);
    wake_up_interruptible(&dev->p_queue);
    return count;
}
```