Archivos y Entrada Salida

Gestión de archivos

- Campo
- Registro
- Archivo
- Bases de datos

Operaciones típicas

- Leer todo
- Leer uno
- Leer siguiente
- Leer previo
- Insertar uno
- Borrar uno
- Actualizar uno.
- Recuperar varios

Sistema de gestión de ficheros

- ...
- Optimizar rendimiento en productividad global y en términos de tiempos de respuesta.
- Variedad de dispositivos.
- Minimizar la pérdida de datos.
- Conjunto estándar de rutinas de acceso.
- Múltiples usuarios.

Tipos de organización y accesos

- Pilas
- Secuenciales
- Secuenciales indexados
- Indexados
- Acceso directo o de dispersión

Organización por directorios

Gestión del almacenamiento secundario

Asignación previa versus dinámica.

Fragmentación del disco, interna y externa.

Asignación contigua

Es una técnica de asignación previa, aunque se puede hacer dinámicamente asignando más secciones. Problemas de fragmentación interna y externa, necesita métodos de compactación

Asignación encadenada o enlazada

Se ajusta mejor a ficheros que serán accedidos secuencialmente. No hay principios de cercanía, hay que “consolidar”.

Asignación indexada

Un nivel y varios niveles, i-nodos en Unix.

Gestión de espacio libre

Vector de bits

Un bit por cada bloque del disco. Relativamente sencillo para encontrar bloques libres u ocupados consecutivos. Si el disco es grande no se puede tener todo en memoria, incrementando los accesos a discos.

Lista de libres

Cada bloque tiene un número secuencial (como vimos antes) y a lista de bloques libres en esa sección

Lista de bloques libres con recuento

Lista de secciones libres enlazada

Puntero más el tamaño. Si se necesita un bloque se puede seleccionar el primero de la lista. Si hay bloques de distintos tamaños se puede hacer una búsque por el “best fit”.

Después de un tiempo el disco estará fragmentado lo que dificulta encontrar varios consecutivos. La asignación de varios bloques individuales y el borrado de ficheros es poco óptimo.

Indexación (recuento)

Trata a los bloque libres de forma similar a los archivos. Hay un índice por cada sección libre.

Gestión E/S y Planificación de discos

- Tipos de dispositivos
- legibles por humanos
- Legibles por la máquina

Existen diferencias entre estas clases de dispositivos

- Velocidad de datos
- Aplicaciones
- Complejidad de control
- Unidad de transferencia
- Representación de datos
- Codificación de error

Organización de entrada/salida

- E/S Programada
- E/S dirigida por interrupciones
- Acceso directo a memoria
-

....

Aspectos de diseño

Objetivos

Eficiencia: las E/S constituyen el cuello de botella.

Generalidad: simplicidad y tratamiento de errores.

Estructura lógica

- E/S lógica
- E/S con dispositivos
- Planificación y control

Sistemas de ficheros

- Gestión de directorios
- Gestión de ficheros
- Organización física

Buffer y cache

Memoria simple

Doble

Circular

Dispositivos de Disco

Pistas, sectores, bloques.

Velocidad angular constante

Velocidad lineal constante

Cabezas fijas y cabezas móviles

Cara simple y doble cara

Particiones

MBR...

Partition Boot record

Platos múltiples

Cilindros

```
s = sectores/pista
p = pista/cilindro
i = cilindro
j = superficie
k = sector en el disco

l = k + s * (j + i * p)
```

Parámetros de rendimiento

Tiempo de búsqueda (*seek time*)

Tiempo que tarda en ubicarse sobre la pista

Retardo de giro (o latencia de giro)

Tiempo que tarda el sector en llega abajo de la cabeza.

T = 1/2r

Tiempo de acceso

suma del tiempo de búsqueda más la latencia.

Tiempo de transferencia

Depende la velocidad del disco T=b/rN

T= tiempo de transferencia

b= bytes a transferir

N=número de bytes por pista

r=velocidad de rotación en revoluciones por segundo

Tiempo medio de acceso total

Ta = Ts + 1/2r + b/rN

Comparación de tiempos

Ts = 10ms, 10.000 rpm, sectores de 512 bytes, 320 sectores por pista

Leer un fichero de 2560 sectores (1.3 Mbytes)

Forma compacta, uno tras otro, consecutivos, 8 pistas

Primera pista: Ts=10ms, Tr= 3 ms, Lectura completa pista= 6 ms. Total: 19 ms

Se leen las otras pistas si retardo de búsqueda, sólo de giro;

19 + 7 x 9 = 92 ms = **0.082**

Acceso aleatorio y/o sectores distribuidos de forma aleatoria

Ts = 10

Retardo de giro = 3 ms

Leer **un** sector: 0.01875

Tiempo medio total un sector: 13,01875

Tiempo total = 2560 x 13,01875 = 33328 ms = 33,328 segundos

Planificación de disco

Estrategias fundamentales:

- Unificación de peticiones (*request merging*)
- Elevador/ascensor

Algortimo básicos

- FIFO
- SSTF
- SCAN
- C-SCAN

Compararlos con un disco de 200 pistas, las cabezas están posicionadas en la 100 inicialmente y la cola de peticiones es: 55, 58, 39, 18, 90, 160, 150, 38, 184

Planificadores en Linux

- ***Noop scheduler*:** Implementa sólo la unificación de peticiones.
- ***Deadline scheduler*:** Implementa unificación más elevador y caducidad de operaciones.
- ***Anticipatory scheduler*** **`("as scheduler"):`** además de unificación minimiza el movimiento de las cabezas esperando por un tiempo determinado antes de pasar a atender peticiones de escritura (no es bueno para servidores de almacenamiento).
- ***Complete fair queuing scheduler*** **`("cfq scheduler"):`** implementa unificación, elevador más estrategia de round robin por proceso y dispositivo.

Gestión de memoria

Enlace de direcciones

1. Compilación: se genera el “código absoluto” en el momento de la compilación.
2. Carga: Se posterga el enlace al momento de la carga.
3. Ejecución: Se debe tener hardware especial.

Carga Dinámica

Las rutinas no se cargan hasta que son necesarias. No requiere apoyo especial del sistema operativo, es responsabilidad de los programadores.

Enlace dinámico

Bibliotecas enlazadas dinámicamente o bibliotecas compartidas (shared object, DLL). Con cada referencia a una librería externa se agrega un pequeño fragmento de código (stub) para localizar la rutina de la librería.

Superposiciones (overlays)

Editor de enlaces

Requisitos

1. Realojamiento
2. Protección: necesita ayuda del hardware.
3. Compartición
4. Organización lógica
5. Organización física: hay dos capas, la memoria RAM y la secundaria.

Carga de programas en memoria

Particiones fijas

Gran fragmentación interna y externa

Asignación: simple.

Particiones dinámicas

Por primera vez en el OS/MVT de IBM. Genera fragmentación externa. Se usa la compactación o defragmentación.

Asignación: mejor ajuste, primer ajuste, siguiente ajuste

Reemplazo: expulsión

Sistema de colegas (buddies)

Particiones de tamaño 2^K tal que L <= K <= U. 2^L es el tamaño más pequeño asignable, 2^U es el más grande (normalmente toda la memoria).

Si se hace una petición de tamaño s y 2^U-1 < s <= 2^U entonces se asigna el bloque entero, sino se divide el bloque en dos y se continúa el proceso.

Reubicación

Direcciones lógicas

Direcciones físicas

Paginación

Políticas de reemplazo

Algoritmos básicos

- Óptima
- LRU
- FIFO
- Reloj

Serie 2 3 2 1 5 2 4 5 3 2 5 2 (con 3 marcos)

Reloj

Bits de acceso (a) y modificación (m) para el reloj

1. Se busca con a=0 y m=0 No se cambia ningún bit.
2. Si falla 1 se busca con a=0 y m=1, se pone a=0.
3. Si falla 2 se repite 1 y si hace falta 2.

Gestión conjunto residente

Tamaño del conjunto residente

- Asignación fija
- Asignación variable

Alcance

- Global
- Local

Asignación variable y alcance local: Conjuntos de Trabajo

Page fault frequency (PFF)

Se define un umbral **F**.

1. Si el tiempo transcurrido desde el último fallo de página es menor que F entonces se agrega

una página al conjunto.

2. Sino se descartan todas las páginas con el bit de uso en cero y se pone a cero el bit de uso de las que quedan.

Se puede mejorar la estrategia con dos umbrales, uno para incrementar y otro para decrementar el tamaño del conjunto.

El principal problema es que no libera páginas durante los procesos de transición.

Variable-interval sampled working set (VSWS)

Se ponen a cero los bit de uso al inicio de cada intervalo. Se mantienen en memoria los que tengan el bit de uso en uno y se descartan las demás. El conjunto de trabajo sólo puede disminuir al final del período, durante el intervalo las páginas se agregan al conjunto.

M: Mínima duración del intervalo

L: Máxima duración del intervalo

Q: Número de fallos de páginas permitidas

1. Si el tiempo transcurrido alcanza L, suspender los procesos y analizar los bit de uso.
2. Si ocurren Q fallos de página antes que se alcance L:
   a) Si t < M esperar por M para suspender procesos y analizar bits de uso.
   b) Si t >= M suspender procesos y analizar bit de uso.

Concurrencia, Bloqueos mutuos y espera infinita

RCU: Red-Copy-Update

So the typical RCU update sequence goes something like the following:

1. Remove pointers to a data structure, so that subsequent readers cannot gain a reference to it.
2. Wait for all previous readers to complete their RCU read-side critical sections.
3. At this point, there cannot be any readers who hold references to the data structure, so it now may safely be reclaimed (e.g., freed, perhaps using `kfree` in the Linux kernel).

Example API

The implementation of RCU in version 2.6 of the Linux kernel is among the better-known RCU implementations, and will be used as an example RCU API in the remainder of this article. The core Linux kernel RCU API (Application Programming Interface) is quite small:

1. rcu_read_lock()
2. rcu_read_unlock()
3. synchronize_rcu() / call_rcu()
4. rcu_assign_pointer()
5. rcu_dereference()

The Linux kernel contains numerous other RCU-related APIs, and these may be found elsewhere [1].

rcu_read_lock

Used by a reader to inform the reclaimer that the reader is entering an RCU read-side critical section. Any RCU-protected data structure accessed during an RCU read-side critical section is guaranteed to remain unreclaimed for the full duration of that critical section. Other schemes, such as reference counts, may be used in conjunction with RCU to maintain longer-term references to data structures.

rcu_read_unlock

Used by a reader to inform the reclaimer that the reader is exiting an RCU read-side critical section. Note that RCU read-side critical sections may be nested and/or overlapping.

synchronize_rcu / call_rcu

Marks the end of updater code and the beginning of reclaimer code. It does this by blocking until all pre-existing RCU read-side critical sections on all CPUs have completed. Note that `synchronize_rcu` will *not* necessarily wait for any subsequent RCU read-side critical sections to complete. For example, consider the following sequence of events:

```
         CPU 0                  CPU 1                 CPU 2
    ----------------- ------------------------ ---------------
1.  rcu_read_lock()
2.                    enters synchronize_rcu()
3.                                              rcu_read_lock()
4.  rcu_read_unlock()
5.                     exits synchronize_rcu()
6.                                             rcu_read_unlock()
```

To reiterate, `synchronize_rcu` waits only for ongoing RCU read-side critical sections to complete, not necessarily for any that begin after `synchronize_rcu` is invoked.

Of course, synchronize_rcu does not necessarily return immediately after the last pre-existing RCU read-side critical section completes. For one thing, there might well be scheduling delays. For

another thing, many RCU implementations process requests in batches in order to improve efficiencies, which can further delay `synchronize_rcu`.

Since `synchronize_rcu` is the API that must figure out when readers are done, its implementation is key to RCU. For RCU to be useful in all but the most read-intensive situations, `synchronize_rcu`'s overhead must also be quite small.

The `call_rcu` API is a callback form of `synchronize_rcu`, and is described in more detail in a later section. Instead of blocking, it registers a function and argument which are invoked after all ongoing RCU read-side critical sections have completed. This callback variant is particularly useful in situations where it is illegal to block.

rcu_assign_pointer

The updater uses this function to assign a new value to an RCU-protected pointer, in order to safely communicate the change in value from the updater to the reader. This function returns the new value, and also executes any memory barrier instructions required for a given CPU architecture. Perhaps more importantly, it serves to document which pointers are protected by RCU.

rcu_dereference

The reader uses `rcu_dereference` to fetch an RCU-protected pointer, which returns a value that may then be safely dereferenced. Note that `rcu_deference` does not actually dereference the pointer, instead, it protects the pointer for later dereferencing. It also executes any needed memory-barrier instructions for a given CPU architecture. Currently, only DEC Alpha needs memory barriers within `rcu_dereference`; on other CPUs, it compiles to nothing, not even a compiler directive.

Note that the value returned by `rcu_dereference` is valid only within the enclosing RCU read-side critical section. For example, the following is not legal:

```
rcu_read_lock();
p = rcu_dereference(head.next);
rcu_read_unlock();
x = p->address;
rcu_read_lock();
y = p->data;
rcu_read_unlock();
```

Holding a reference from one RCU read-side critical section to another is just as illegal as holding a reference from one lock-based critical section to another! Similarly, using a reference outside of the critical section in which it was acquired is just as illegal as doing so with normal locking.

As with `rcu_assign_pointer`, an important function of `rcu_dereference` is to document which pointers are protected by RCU.

Proceso en estado D

Son procesos en estado “ininterrumpible”, si falla el dispositivo o el gestor del mismo.

Inversión de prioridades

Lo sufrió el Mars Pathfinder.

Tres procesos con prioridades distintas A es la de mayor, C la de menor. C hace un wait sobre un semáforo, A luego hace un wait sobre el mismo semáforo. B se ejecuta al no haber otro proceso listo de mayor prioridad y no deja que C salga de su sección crítica.

Lo mismo puede pasar con mensajes.

Principios de Deadlocks

Bloqueo permanente de un conjunto de procesos compitiendo por recursos del sistema o por comunicarse entre ellos. No hay solución general eficiente.

Reusables

Los recursos pueden ser reusables o consumibles. Los reusables son recursos que pueden ser usados de forma segura por sólo un proceso a la vez y que no desaparecen luego del dicho uso, Ejemplos son canales de entrada-salida, memoria principal y secundaria, dispositivos y estructuras de datos como ficheros, bases de datos, semáforos.

P1	***P2***
solicitar(D)	solicitar(T)
solicitar(T)	solicitar(D)
liberar(T)	liberar(D)
liberar(D)	liberar(T)

Otro ejemplo es con memoria, suponiendo que hay 200 MB disponibles

P1	***P2***
solicitar 80 MB	solicitar 70 MB
solicitar 60 MB	solicitar 80 MB

Consumibles

Son recursos que son creados y destruidos. Típicamente no hay limites en el número de un tipo de recursos. Cuando un recurso es usado por un proceso deja de existir. Ejemplos son: interrupciones, señales, mensajes e información de de los buffers de E/S

P1	***P2***
Recibir(P2, M)	Recibir(P1, Q)
Enviar(P2, N)	Enviar(P1, R)

Condiciones para el deadlock

1. Exclusión mutua
2. Retención y espera
3. No apropiación
4. Espera circular

Prevención de Deadlocks

Se trata de evitar que se presenten una o varias de las condiciones anteriores

Detección de deadlocks

Se asigna los recursos a los procesos y el SO periódicamente verifica el estado. Se puede usar cualquier algoritmo de detección de ciclos en un grafo. Una vez que ha sido detectado se puede hacer:

1. Abortar todos los procesos bloqueados
2. Volver los procesos involucrados a un estado previo predefinido (checkpoint) Requiere que el sistema incluya mecanismos de *rollback* y de reinicio. El *deadlock* puede volver ocurrir.

3. Abortar de forma sucesiva hasta que el deadlock desaprezca. El orden es importante.
4. Apropiarse de forma sucesiva de los recursos hasta que el deadlock desaparezca. Un proceso que pierda el recurso debe ser vuelto atrás a un punto predeterminado.

Prevención dinámica // Predicción

Con la prevención estática se restringe la asignación de recursos para evitar que se produzcan las 4 condiciones. Se hace indirectamente mediante políticas de evitar que ocurran las tres primeras o directamente que no ocurra la espera circular. Es ineficiente a la hora de asignar recursos.

En cambio con la dinámica se permiten las tres primeras pero se hacen selecciones de asignación cuidadosas para evitar que se llegue a una situación de posible deadlock.

Hay dos posibilidades:

- No iniciar un proceso que puede llevar a una situación de deadlock.
- No garantizar la asignación incrementar de recursos si esa asignación puede generar una situación de deadlock.

Denegación de inicio de procesos

Sólo se inicia un nuevo proceso si sus necesidades máximas de éste más la suma de todos los que están ejecutando no superan las disponibilidades del sistema. Es muy ineficiente.

Denegación de asignación: algoritmo del banquero

- multiple instances of resource types IMPLIES cannot use resource-allocation graph
- banks do not allocate cash unless they can satisfy customer needs when a new process enters the system
- declare in advance maximum need for each resource type
- cannot exceed the total resources of that type
- later, processes make actual request for some resources
- if the the allocation leaves system in safe state grant the resources
- otherwise, suspend process until other processes release enough resources

Estructuras de datos

```
#define MAXN 10             /* maximum number of processes                  */
#define MAXM 10             /* maximum number of resource types             */
int Available[MAXM];        /* Available[j] = current # of unused resource j */
int Max[MAXN][MAXM];        /* Max[i][j] = max demand of i for resource j    */
int Allocation[MAXN][MAXM]; /* Allocation[i][j] = i's current allocation of j*/
int Need[MAXN][MAXM];       /* Need[i][j] = i's potential for more j         */
                            /* Need[i][j] = Max[i][j] - Allocation[i][j]     */
```

Notation:

```
 X <= Y iff X[i] <= Y[i] for all i
```

(0,3,2,1) is less than (1,7,3,2)

(1,7,3,2) is NOT less than (0,8,2,1)

Each row of *Allocation* and *Need* are vectors: Allocation_i and Need_i

Ejemplo

Initially:

```
Available
  A B C
 10 5 7
```

Later Snapshot:

```
      Max   -  Allocation  =  Need   Available
     A B C       A B C       A B C     A B C
P0   7 5 3       0 1 0       7 4 3     3 3 2
P1   3 2 2       2 0 0       1 2 2
P2   9 0 2       3 0 2       6 0 0
P3   2 2 2       2 1 1       0 1 1
P4   4 3 3       0 0 2       4 3 1
```

Estado seguro

- consider some sequence of processes
- if the first process has Need less than Available
- it can run until done
- then release all of its allocated resources
- allocation is increased for next process
- if the second process has Need less than Available
- ...
- then all of the processes will be able to run eventually
- IMPLIES system is in a **safe state**

Algoritmo de estado seguro

```
STEP 1: initialize
  Work := Available;
  for i = 1,2,...,n
    Finish[i] = false
STEP 2: find i such that both
  a. Finish[i] is false
  b. Need_i <= Work
  if no such i, goto STEP 4
STEP 3:
  Work := Work + Allocation_i
  Finish[i] = true
  goto STEP 2
STEP 4:
  if Finish[i] = true for all i, system is in safe state
```

Ejemplo

Using the previous example, P1,P3,P4,P2,P0 satisfies criteria.

```
      Max   -  Allocation  =  Need <= Work  Available
     A B C       A B C       A B C           A B C
P1   3 2 2       2 0 0       1 2 2    3 3 2   3 3 2
P3   2 2 2       2 1 1       0 1 1    5 3 2
P4   4 3 3       0 0 2       4 3 1    7 4 3
P2   9 0 2       3 0 2       6 0 0    7 4 5
P0   7 5 3       0 1 0       7 4 3   10 4 7
                                     10 5 7<<< initial system
```

Solicitud de recursos

```
STEP 1: if Request_i <= Need_i
```

```
          goto STEP 2
        else ERROR
STEP 2: if Request_i <= Available
          goto STEP 3
        else suspend P_i
STEP 3: pretend to allocate requested resources
          Available := Available - Request_i
          Allocation_i := Allocation_i + Request_i;
          Need_i := Need_i - Request_i
STEP 4: if pretend state is SAFE
          then do a real allocation and P_i proceeds
        else
          restore the original state and suspend P_i
```

Algortimo de solicitud de recursos

Say P1 requests (1,0,2)

Compare to Need_1: (1,0,2) <= (1,2,2)

Compare to Available: (1,0,2) <= (3 3 2)

Pretend to allocate resources:

```
      Max    -  Allocation  =  Need   Available
     A B C       A B C       A B C     A B C
P0   7 5 3       0 1 0       7 4 3     2 3 0<<<
P1   3 2 2       3 0 2<<<    0 2 0<<<
P2   9 0 2       3 0 2       6 0 0
P3   2 2 2       2 1 1       0 1 1
P4   4 3 3       0 0 2       4 3 1
```

Is this safe? Yes: P1, P3, P4, P0, P2

Can P4 get (3,3,0)? No, (3,3,0) > (2,3,0) Available

Can P0 get (0,2,0)? (0,2,0) < (2,3,0) Available

Pretend: Available goes to (2,1,0)

But ALL Needs are greater than Available IMPLIES NOT SAFE