2023-02-15

Letture e scritture non allineate a blocco

Vogliamo aggiungere al nucleo delle primitive che ci permettano di leggere e scrivere dall'hard disk come se questo fosse un unico array di byte, invece che di blocchi. In particolare, il blocco 0 conterrà i byte nell'intervallo $[0, 512)$, il blocco 1 i byte nell'intervallo $[512, 1024)$, e così via. Supponiamo che l'utente chieda di leggere n byte a partire dal byte b. Le primitive dovranno internamente accedere ai blocchi che contengono tutto l'intervallo $[b, b + n)$ e poi restituire o modificare solo i byte richiesti dall'utente.

Per rendere efficienti queste operazioni le nuove primitive usano una buffer cache che mantiene in memoria i blocchi acceduti più di recente, in modo che eventuali letture di byte contenuti in blocchi che si trovino nella buffer cache possano essere realizzate con una semplice copia da memoria a memoria, senza ulteriori operazioni di I/O. Per quanto riguarda le scritture adottiamo una politica write-back/write-allocate. Per il rimpiazzamento adottiamo la politica LRU (Least Recently Used): se la cache è piena e deve essere caricato un blocco non in cache, si rimpiazza il blocco a cui non si accede da più tempo.

Per realizzare la buffer cache definiamo la seguente struttura dati nel modulo I/O:

```
struct buf_des {
    /// id del blocco contenuto in buf_des::buf (se buf_des::full è true)
    natl block;
    /// true se buf_des::buf contiene dati validi
    bool full;
    int next,   ///< indice del prossimo buffer nella coda LRU
        prev;   ///< indice del buffer precedente nella coda LRU
    /// copia del blocco con id buf_des::block (se buf_des::full è true)
    natb buf[DIM_BLOCK];
    /// true se il buffer è stato modificato rispetto alla copia sull hard disk
    bool dirty;
};
```

La struttura rappresenta un singolo elemento della buffer cache. I campi sono significativi solo se `full` è `true`. In quel caso `buf` contiene una copia del blocco `block`. I campi `next` e `prev` servono a realizzare la coda LRU come una lista doppia (si veda più avanti). Il campo `dirty` è `true` se (e solo se) il blocco è stato modificato e il suo contenuto non è stato ancora ricopiato sull'hard disk. Aggiungiamo poi i seguenti campi alla struttura `des_ata` (che è il descrittore dell'hard disk):

```
struct des_ata {
    ...
    buf_des bufcache[MAX_BUF_DES];
```

```
    int lru,    ///< indice del prossimo buffer da rimpiazzare
        mru;    ///< indice del buffer acceduto più di recente
    natl rd_count;  ///< non usare: solo per debug e test
    natl wr_count;  ///< non usare: solo per debug e test
};
```

Il campo `bufcache` è la buffer cache vera e propria; il campo `lru` è l'indice in `bufcache` del prossimo buffer da rimpiazzare (testa della coda LRU) e il campo `mru` è l'indice del buffer acceduto più di recente (ultimo elemento della coda LRU). I campi `next` e `prev` in ogni elemento di `bufcache` sono gli indici del prossimo e del precedente buffer nella coda LRU.

```
void bufcache_promote(buf_des *b)
{
    des_ata *d = &hard_disk;
    int i = b - d->bufcache;

    if (b->next >= 0 || b->prev >= 0) {
        // d era già in lista: estraiamolo
        if (b->next >= 0)
            d->bufcache[b->next].prev = b->prev;
        else
            d->mru = b->prev;
        if (b->prev >= 0)
            d->bufcache[b->prev].next = b->next;
        else
            d->lru = b->next;
    }
    if (d->mru >= 0)
        d->bufcache[d->mru].next = i;
    else
        d->lru = i;
    b->prev = d->mru;
    b->next = -1;
    d->mru = i;
}
buf_des* bufcache_search(natl block)
{
    des_ata *d = &hard_disk;

    for (int i = 0; i < MAX_BUF_DES; i++) {
        buf_des *b = &d->bufcache[i];
        if (b->full && b->block == block)
            return b;
    }
    return nullptr;
}
```

```
void estern_hd(natq)
{
    des_ata* d = &hard_disk;
    for(;;) {
        d->cont--;
        hd::ack();
        switch (d->comando) {
        case hd::READ_SECT:
            d->rd_count++;
            hd::input_sect(d->punt);
            d->punt += DIM_BLOCK;
            break;
        case hd::WRITE_SECT:
            d->wr_count++;
            if (d->cont != 0) {
                hd::output_sect(d->punt);
                d->punt += DIM_BLOCK;
            }
            break;
        case hd::READ_DMA:
            d->rd_count++;
            bm::ack();
            break;
        case hd::WRITE_DMA:
            d->wr_count++;
            bm::ack();
            break;
        }
        if (d->cont == 0)
            sem_signal(d->sincr);
        wfi();
    }
}

bool hd_init()
{
    ...
    // conviene inserire anche i buffer vuoti nella coda LRU (in un ordine
    // qualsiasi), in modo da non dover considerare a parte il caso di buffer
    // vuoto.
    d->lru = d->mru = -1;
    for (int i = 0; i < MAX_BUF_DES; i++) {
        buf_des *b = &d->bufcache[i];
        b->full = false;
        b->next = b->prev = -1;
```

```
        bufcache_promote(b);
        b->dirty = false;
    }
    return true;
}
```

```
.extern c_bufcache_status
a_bufcache_status:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_bufcache_status
    iretq
    .cfi_endproc
```

Aggiungiamo infine le seguenti primitive che realizzano il meccanismo descritto:

- `void bufread_n(natb* vetti, natl first, natl nbytes)` (tipo 0x48, da realizzare): legge dall'hard disk i byte nell'intervallo [`first`, `first` + `nbytes`) e li copia in `vetti`, usando la buffer cache per minimizzare il numero di operazioni di I/O;
- `void bufwrite_n(natb* vetto, natl first, natl nbytes)`(tipo 0x49, da realizzare): scrive i byte contenuti in `vetto` nell'intervallo [`first`, `first` + `nbytes`) di byte dell'hard disk, usando la buffer cache per minimizzare il numero di operazioni di I/O.

Nota: in nessun caso la cache deve eseguire operazioni di lettura/scrittura dall'hard disk che non siano strettamente necessarie.

Controllare eventuali problemi di Cavallo di Troia e abortire il processo se necessario.

Modificare i file `io.cpp` `io.s` in modo da realizzare il meccanismo descritto.

```
extern "C" void c_bufread_n(natb *vetti, natl first_byte, natl nbytes)
{
    if (!access(vetti, nbytes, true)) {
        flog(LOG_WARN, "bufread_n: parametri non validi: %p, %d", vetti, nbytes);
        abort_p();
    }

    des_ata *d = &hard_disk;

    natl block = first_byte / DIM_BLOCK;
    natl offset = first_byte % DIM_BLOCK;
    natl toread = DIM_BLOCK - offset < nbytes ? DIM_BLOCK - offset : nbytes;
    sem_wait(d->mutex);
    while (nbytes > 0) {
```

```
        buf_des *b = bufcache_search(block);
        if (!b) {
            b = &d->bufcache[d->lru];
            if (b->dirty) {
                starthd_out(d, b->buf, b->block, 1);
                sem_wait(d->sincr);
            }
            starthd_in(d, b->buf, block, 1);
            sem_wait(d->sincr);
            b->block = block;
            b->full = true;
            b->dirty = false;
        }
        memcpy(vetti, b->buf + offset, toread);
        bufcache_promote(b);
        vetti += toread;
        nbytes -= toread;
        offset = 0;
        toread = DIM_BLOCK < nbytes ? DIM_BLOCK : nbytes;
        block++;
    }
    sem_signal(d->mutex);
}

extern "C" void c_bufwrite_n(natb *vetto, natq first_byte, natq nbytes)
{
    if (!access(vetto, nbytes, false)) {
        flog(LOG_WARN, "bufwrite_n: parametri non validi: %p, %lu", vetto, nbytes);
        abort_p();
    }

    des_ata *d = &hard_disk;

    natl block = first_byte / DIM_BLOCK;
    natl offset = first_byte % DIM_BLOCK;
    natl towrite = DIM_BLOCK - offset < nbytes ? DIM_BLOCK - offset : nbytes;
    sem_wait(d->mutex);
    while (nbytes > 0) {
        buf_des *b = bufcache_search(block);
        if (!b) {
            b = &d->bufcache[d->lru];
            if (b->dirty) {
                starthd_out(d, b->buf, b->block, 1);
                sem_wait(d->sincr);
            }
            if (towrite < DIM_BLOCK) {
```

```
                starthd_in(d, b->buf, block, 1);
                sem_wait(d->sincr);
            }
            b->block = block;
            b->full = true;
        }
        memcpy(b->buf + offset, vetto, towrite);
        b->dirty = true;
        bufcache_promote(b);
        vetto += towrite;
        nbytes -= towrite;
        offset = 0;
        towrite = DIM_BLOCK < nbytes ? DIM_BLOCK : nbytes;
        block++;
    }
    sem_signal(d->mutex);
}
```

```
fill_io_gates:
    ...
    fill_io_gate    IO_TIPO_BRD a_bufread_n
    fill_io_gate    IO_TIPO_BWR a_bufwrite_n
    fill_io_gate    IO_TIPO_BCS a_bufcache_status
    ...
```

```
.extern c_bufread_n
a_bufread_n:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_bufread_n
    iretq
    .cfi_endproc

.extern c_bufwrite_n
a_bufwrite_n:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_bufwrite_n
    iretq
    .cfi_endproc
```