2024-01-10

Colleghiamo al sistema delle periferiche PCI di tipo `ce`, con vendorID `0xedce` e deviceID `0x1234`. Ogni periferica `ce` usa 16 byte nello spazio di I/O a partire dall'indirizzo base specificato nel registro di configurazione BAR0, sia b.

Le periferiche `ce` sono semplici periferiche con un registro RBR di ingresso, tramite il quale è possibile leggere il prossimo byte disponibile. Se abilitata scrivendo 1 nel registro CTL, la perifica invia una richiesta di interruzione quando il registro RBR contiene un nuovo byte. La periferica non invia nuove richieste di interruzione fino a quando il registro RBR non viene letto.

Vogliamo realizzare una primitiva `ceread_n_to()` che permetta di leggere n byte da una periferica `ce`, ma specificando un *timeout*: se l'operazione non si conclude entro il timeout, la primitiva annulla il trasferimento e restituisce soltanto i byte ricevuti fino a quel punto (eventualmente nessuno).

Per evitare che il vincolo temporale venga violato a causa dell'esecuzione di processi a più alta priorità, realizziamo la primitiva `ceread_n_to()` con un driver, invece di usare handler e processo esterno: in questo modo sia il driver che la primitiva si trovano nel modulo sistema, il driver è atomico e la primitiva può essere interrotta solo quando chiama le primitive semaforiche. Si noti che la primitiva potrebbe comunque dover attendere a causa della mutua esclusione sull'accesso alla periferica, e questo tempo entra nel conteggio del timeout.

Per descrivere le periferiche `ce` aggiungiamo le seguenti strutture dati al modulo sistema:

```
struct des_ce {
    /// @name Indirizzi dei registri
    /// @{
    ioaddr iCTL,    ///< registro di controllo
           iSTS,    ///< registro di stato
           iRBR;    ///< registro di ingresso
    /// @}

    /// true se è in corso una operazione di trasferimento
    bool enabled;
    /// indirizzo a cui scrivere il prossimo byte letto
    char *buf;
    /// numero di byte ancora da leggere
    natl quanti;
    /// indice del semaforo di sincronizzazione
    natl sync;
    /// indice del semaforo di mutua esclusione
    natl mutex;
};
```

```
/// Array dei descrittori di periferiche CE
des_ce array_ce[MAX_CE];
/// Numero dei descrittori di periferiche CE utilizzati
natl next_ce;
```

Ogni `des_ce` descrive una periferica `ce`. I campi `iCTL`, `iSTS` e `iRBR` contengono gli indirizzi nello spazio di I/O dei registri dell'interfaccia (il registro STS può essere ignorato); il campo `enabled` vale `true` solo se è in corso una operazione di trasferimento, e dunque l'interfaccia è abilitata a inviare richieste di interruzione; se è in corso un trasferimento, `quanti` conta quanti byte restano da leggere e `buf` dice dove andrà scritto il prossimo byte; `sync` è l'indice di un semaforo di sincronizzazione e `mutex` è l'indice di un semaforo di mutua esclusione. Le periferiche `ce` installate sono numerate da 0 a `next_ce` meno uno. Il descrittore della periferica numero `x` si trova in `array_ce[x]`.

La primitiva da aggiungere è

- `bool cread_n_to(natl id, char *buf, natl& quanti, natl to)` (tipo 0x2a): avvia una operazione di trasferimento di `quanti` byte dalla periferica `ce` numero `id` verso il buffer `buf`, con timeout `to`. Restituisce `true` se l'operazione si è conclusa entro il timeout, altrimenti restituisce `false` e scrive in `quanti` il numero di byte effettivamente trasferiti; abortisce il processo in caso di errore (`id` non valido, Cavallo di Troia, timeout pari a zero).

È possibile utilizzare la primitiva `natl sem_wait_to(natl sem, natl to)` che esegue una wait sul semaforo `sem` con timeout `to` (che deve essere maggiore di zero). La primitiva si comporta come una normale `sem_wait(sem)`, ma se il processo non viene risvegliato entro `to` termina l'attesa e restituisce 0; altrimenti restituisce il tempo che mancava allo scatto del timeout. Si consideri trascurabile il tempo speso eseguendo istruzioni non interrompibili (in altre parole, l'unico tempo che conta ai fini del timeout è quello speso mentre il processo è bloccato su qualche semaforo).

```
struct des_proc {
    ...
    /// indice del semaforo su cui il processo è in attesa con timeout (0xffffffff se nessu
    natl waiting_on;
};

des_proc* crea_processo(void f(natq), natq a, int prio, char liv)
{
    ...
    p->waiting_on = 0xFFFFFFFF;
}

extern "C" void c_sem_signal(natl sem)
{
```

```
    ...
    if (s->counter <= 0) {
        des_proc* lavoro = rimozione_lista(s->pointer);
        if (lavoro->waiting_on != 0xFFFFFFFF) {
            lavoro->contesto[I_RAX] = rimozione_lista_attesa(lavoro);
            lavoro->waiting_on = 0xFFFFFFFF;
        }
        ...
    }
}

natl rimozione_lista_attesa(des_proc* p)
{
    richiesta** r;
    natl rem = 0;

    for (r = &sospesi; *r && (*r)->pp != p; r = &(*r)->p_rich)
        rem += (*r)->d_attesa;
    if (richiesta* t = *r) {
        rem += t->d_attesa;
        if ( (*r = t->p_rich) )
            (*r)->d_attesa += t->d_attesa;
        delete t;
    }
    return rem;
}

extern "C" void c_driver_td(void)
{
    ...
    while (sospesi != nullptr && sospesi->d_attesa == 0)
    {
        if (sospesi->pp->waiting_on != 0xFFFFFFFF)
        {
            des_sem *s = &array_dess[sospesi->pp->waiting_on];
            s->counter++;
            des_proc **p = &s->pointer;
            while (*p != sospesi->pp)
                p = &(*p)->puntatore;
            *p = (*p)->puntatore;
            sospesi->pp->contesto[I_RAX] = 0;
            sospesi->pp->waiting_on = 0xFFFFFFFF;
        }
        ...
    }
    schedulatore();
```

```
}
```

Modificare i file `sistema.s` e `sistema.cpp` in modo da realizzare le parti mancanti (incluso il driver `void c_driver_ce(natl id)`).

```
extern "C" void c_sem_wait_to(natl sem, natl to)
{
    // una primitiva non deve mai fidarsi dei parametri
    if (!sem_valido(sem)) {
        flog(LOG_WARN, "semaforo errato: %u", sem);
        c_abort_p();
        return;
    }

    if (!to) {
        flog(LOG_WARN, "timeout non valido");
        c_abort_p();
        return;
    }

    des_sem *s = &array_dess[sem];
    s->counter--;
    if (s->counter < 0) {
        inserimento_lista(s->pointer, esecuzione);
        esecuzione->waiting_on = sem;
        c_delay(to);
    } else {
        esecuzione->contesto[I_RAX] = to;
    }
}

extern "C" void main(natq)
{
    ...
    ce_init();
}
bool ce_init()
{

    for (natb bus = 0, dev = 0, fun = 0;
         pci::find_dev(bus, dev, fun, 0xedce, 0x1234);
         pci::next(bus, dev, fun))
    {
        if (next_ce >= MAX_CE) {
            flog(LOG_WARN, "troppi dispositivi ce");
            break;
        }
```

```
        des_ce *ce = &array_ce[next_ce];
        ioaddr base = pci::read_confl(bus, dev, fun, 0x10);
        base &= ~0x1;
        ce->iCTL = base;
        ce->iSTS = base + 4;
        ce->iRBR = base + 8;
        if ( (ce->sync = sem_ini(0)) == 0xFFFFFFFF ) {
            flog(LOG_WARN, "impossibile allocare semaforo");
            return false;
        }
        if ( (ce->mutex = sem_ini(1)) == 0xFFFFFFFF ) {
            flog(LOG_WARN, "impossibile allocare semaforo");
            return false;
        }
        natb irq = pci::read_confb(bus, dev, fun, 0x3c);
        apic::set_VECT(irq, INTR_TIPO_CE + next_ce);
        apic::set_MIRQ(irq, false);
        apic::set_TRGM(irq, false);
        flog(LOG_INFO, "ce%d: %02x:%02x.%1x base=%04x IRQ=%u", next_ce, bus, dev, fun, ce->i
        next_ce++;
    }
    return true;
}
```

```
a_driver_ce_0:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    movq $0, %rdi
    call c_driver_ce
    call apic_send_EOI
    call carica_stato
    iretq
    .cfi_endproc

/// Driver associato alla periferica CE1
a_driver_ce_1:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    movq $1, %rdi
    call c_driver_ce
```

```
    call apic_send_EOI
    call carica_stato
    iretq

    .cfi_endproc
/// Driver associato alla periferica CE2
a_driver_ce_2:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    movq $2, %rdi
    call c_driver_ce
    call apic_send_EOI
    call carica_stato
    iretq

    .cfi_endproc
/// Driver associato alla periferica CE3
a_driver_ce_3:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    movq $3, %rdi
    call c_driver_ce
    call apic_send_EOI
    call carica_stato
    iretq
    .cfi_endproc

/// @cond
    .global sem_ini
sem_ini:
    .cfi_startproc
    int $TIPO_SI
    ret
    .cfi_endproc

    // non strettamente necessaria
    .global access
access:
    .cfi_startproc
    int $TIPO_ACC
```

```
    ret
    .cfi_endproc

    .global sem_wait_to
sem_wait_to:
    .cfi_startproc
    int $TIPO_WTO
    ret
    .cfi_endproc

    .global sem_signal
sem_signal:
    .cfi_startproc
    int $TIPO_S
    ret
    .cfi_endproc

    .global abort_p
abort_p:
    .cfi_startproc
    int $TIPO_AB
    ret
    .cfi_endproc
```

Suggerimento: quando si annulla l'operazione `ceread_n_to()` si faccia attenzione a non lasciare la periferica con le interruzioni abilitate, e ci si assicuri che il driver gestisca le richieste solo se è ancora in corso una operazione di trasferimento.

```
extern "C" bool c_ceread_n_to(natl id, char *buf, natl& quanti, natl to)
{
    natl rem;

    if (id >= next_ce) {
        flog(LOG_WARN, "ce non riconosciuto: %d", id);
        abort_p();
    }

    if (!c_access(reinterpret_cast<vaddr>(&quanti), sizeof(quanti), true, false)) {
        flog(LOG_WARN, "quanti non valido");
        abort_p();
    }

    if (!c_access(reinterpret_cast<vaddr>(buf), quanti, true)) {
        flog(LOG_WARN, "buf non valido");
        abort_p();
    }
```

```
    if (!quanti)
        return true;

    des_ce *ce = &array_ce[id];
    to = sem_wait_to(ce->mutex, to);
    if (!to) {
        quanti = 0;
        return false;
    }
    ce->buf = buf;
    ce->quanti = quanti;
    ce->enabled = true;
    outputb(1, ce->iCTL);
    to = sem_wait_to(ce->sync, to);
    if (ce->enabled) {
        outputb(0, ce->iCTL);
        ce->enabled = false;
    }
    rem = ce->quanti;
    sem_signal(ce->mutex);
    if (!to) {
        quanti -= rem;
        return false;
    }
    return true;
}

extern "C" void c_driver_ce(int id)
{
    des_ce *ce = &array_ce[id];

    if (!ce->enabled)
        return;

    ce->quanti--;
    if (ce->quanti == 0) {
        outputb(0, ce->iCTL);
        ce->enabled = false;
        des_sem *s = &array_dess[ce->sync];
        s->counter++;

        if (s->counter <= 0) {
            des_proc* lavoro = rimozione_lista(s->pointer);
            if (lavoro->waiting_on != 0xFFFFFFFF) {
                lavoro->contesto[I_RAX] = rimozione_lista_attesa(lavoro);
```

```
                lavoro->waiting_on = 0xFFFFFFFF;
            }
            inspronti();    // preemption
            inserimento_lista(pronti, lavoro);
            schedulatore(); // preemption
        }
    }
    char b = inputb(ce->iRBR);
    *ce->buf = b;
    ce->buf++;
}

.global init_idt
init_idt:
    .cfi_startproc
    ...
    carica_gate TIPO_WTO    a_sem_wait_to   LIV_UTENTE
    carica_gate TIPO_CEDEBUG    a_cedebug   LIV_UTENTE
    carica_gate TIPO_CEREADTO   a_ceread_n_to   LIV_UTENTE
    carica_gate INTR_TIPO_CE    a_driver_ce_0   LIV_SISTEMA

    carica_gate INTR_TIPO_CE+1  a_driver_ce_1   LIV_SISTEMA
    carica_gate INTR_TIPO_CE+2  a_driver_ce_2   LIV_SISTEMA
    carica_gate INTR_TIPO_CE+3  a_driver_ce_3   LIV_SISTEMA

.extern c_sem_wait_to
a_sem_wait_to:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_sem_wait_to
    call carica_stato
    iretq
    .cfi_endproc

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