2024-01-29 - Barriera con timeout

Sincronizzazione

Aggiungiamo al nucleo il meccanismo delle *barriere con timeout*.

Una barriera *senza* timeout serve a sincronizzare un certo numero di processi e funziona nel modo seguente: la barriera è normalmente chiusa; un processo che arriva alla barriera si blocca; la barriera si apre solo quando sono arrivati tutti i processi attesi, che a quel punto si sbloccano; una volta aperta e sbloccati tutti i processi, la barriera si richiude e il meccanismo si ripete.

Il timeout cambia le cose nel seguente modo:

- il primo processo che arriva alla barriera dopo una chiusura fa partire il timeout;
- se tutti gli altri processi attesi arrivano prima dello scatto del timeout, la barriera si comporta normalmente (si apre, tutti i processi si sbloccano e poi la barriera si richiude);
- altrimenti la barriera entra in uno stato "erroneo": si apre (quindi, processi già arrivati si risvegliano) e resta aperta fino a quando non sono arrivati tutti i processi attesi, ma tutti i processi la attraversano ricevendo un errore; quando arriva l'ultimo processo, la barriera esce dallo stato erroneo e si richiude.

Nota: sopra e nel seguito, dove diciamo "dall ultima chiusura", intendiamo anche l'istante in cui la barriera è stata creata.

Per rappresentare una barriera introduciamo la seguente struttura dati:

```
struct barrier_t {
    /// Numero di processi che devono sincronizzarsi
    natl nproc;
    /// Numero di processi già arrivati
    natl narrived;
    /// Timeout richiesto per questa barriera
    natl timeout;
    /// True se la barriera è nello stato erroneo
    bool bad;
    /// Coda dei processi che attendono l'apertura della barriera
    des_proc *waiting;
    /// Puntatore al primo processo arrivato alla barriera
    des_proc *first;
} barriers[MAX_BARRIERS];

/// Numero di descrittori di barriera utilizzati
natl barrier_nextid = 0;
```

Dove: `nproc` è il numero di processi che devono sincronizzarsi sulla barriera; `narrived` conta i processi arrivati alla barriera dall'ultima chiusura; `timeout` è il timeout che regola l'entrata nello stato erroneo; `bad` è true se e solo se la barriera si trova nello stato erroneo; `waiting` è la coda dei processi che attendono l'apertura della barriera; `first` punta al descrittore del primo processo arrivato dall'ultima chiusura (quello che ha avviato il timeout corrente).

Aggiungiamo anche il seguente campo ai descrittori di processo:

```
struct des_proc {
    ...
    /// Se questo campo è diverso da 0xFFFFFFFF, vuol dire che il processo
    /// è bloccato sulla barriera con questo identificatore, ed è il primo
    /// processo ad essere arrivato su questa barriera dall'ultima chiusura.
    natl barrier_id;
};

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

extern "C" void c_driver_td(void)
{
    ...
    while (sospesi != nullptr && sospesi->d_attesa == 0)
    {
        check_barrier(sospesi->pp);
        ...
    }
    schedulatore();
}

void rimozione_lista_attesa(des_proc* p)
{
    richiesta** r;

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

Se questo campo è diverso da 0xFFFFFFFF, vuol dire che questo processo è bloccato sulla barriera con identificatore `barrier_id`, ed è il primo processo ad essere arrivato su quella barriera dall'ultima chiusura.

Aggiungiamo inoltre le seguenti primitive:

- `natl barrier_create(natl nproc, natl timeout)` (già realizzata): crea una nuova barriera che sincronizza `nproc` processi con timeout `timeout` e ne restituisce l'identificatore (`0xFFFFFFFF` se non è stato possibile completare l'operazione).
- `bool barrier(natl id)` (da realizzare): fa giungere il processo corrente alla barriera di identificatore `id`. È un errore se tale barriera non esiste. Restituisce `true` quando termina normalmente, e `false` quando termina perchè la barriera è stata attraversata nello stato erroneo.

Le primitive abortiscono il processo chiamante in caso di errore e tengono conto della priorità tra i processi.

```
extern "C" void c_barrier_create(natl nproc, natl to)
{
    if (!to  || !nproc) {
        flog(LOG_WARN, "parametri non validi");
        c_abort_p();
        return;
    }

    if (barrier_nextid >= MAX_BARRIERS) {
        flog(LOG_WARN, "create troppe barriere");
        esecuzione->contesto[I_RAX] = 0xFFFFFFFF;
        return;
    }

    esecuzione->contesto[I_RAX] = barrier_nextid;
    barrier_t *b = &barriers[barrier_nextid];
    barrier_nextid++;

    b->nproc = nproc;
    b->narrived = 0;
    b->timeout = to;
    b->bad = false;
    b->first = nullptr;
    b->waiting = nullptr;
}
```

Modificare i file `sistema.cpp` e `sistema.s` in modo da realizzare le primitive mancanti.

```
void check_barrier(des_proc *p)
```

```
{
    if (p->barrier_id == 0xFFFFFFFF)
        return;

    barrier_t *b = &barriers[p->barrier_id];
    p->barrier_id = 0xFFFFFFFF;

    while (b->waiting) {
        des_proc *work = rimozione_lista(b->waiting);
        work->contesto[I_RAX] = false;
        // il processo corrente verrà inserito in lista
        // pronti dal driver del timer, quindi dobbiamo
        // stare attendi a non inserirlo due volte
        if (work != p)
            inserimento_lista(pronti, work);
    }
    if (b->narrived == b->nproc) {
        b->first = nullptr;
        b->narrived = 0;
    } else {
        b->bad = true;
    }
}

/// Parte C++ della primitiva barrier()
extern "C" void c_barrier(natl id)
{
    if (id >= barrier_nextid) {
        flog(LOG_WARN, "id non valido: %d", id);
        c_abort_p();
        return;
    }

    barrier_t *b = &barriers[id];

    b->narrived++;

    if (b->bad) {
        esecuzione->contesto[I_RAX] = false;
        if (b->narrived == b->nproc) {
            b->bad = false;
            b->narrived = 0;
        }
        return;
    }
```

```
    if (!b->first) {
        b->first = esecuzione;
        esecuzione->barrier_id = id;
        richiesta* p = new richiesta;
        p->d_attesa = b->timeout;
        p->pp = esecuzione;
        inserimento_lista_attesa(p);
    }

    inserimento_lista(b->waiting, esecuzione);
    if (b->narrived == b->nproc) {
        rimozione_lista_attesa(b->first);
        b->first->barrier_id = 0xFFFFFFFF;
        b->first = nullptr;
        while (b->waiting) {
            des_proc *work = rimozione_lista(b->waiting);
            work->contesto[I_RAX] = true;
            inserimento_lista(pronti, work);
        }
        b->narrived = 0;
    }
    schedulatore();
}
```