2025-02-12 - Pipeline di comunicazione

Due processi possono comunicare tramite una `pipe`, un canale con una estremità di scrittura e una di lettura attraverso il quale viaggia una sequenza di caratteri. I caratteri inviati dall'estremità di scrittura possono essere letti dall'estremità di lettura. Il sistema contiene un numero prefissato `pipe`, numerate da zero a `MAX_PIPE` meno uno. Un processo che voglia leggere o scrivere in una pipe deve prima *aprire* la corrispondente estremità, bloccandosi (se necessario) in attesa che venga aperta anche l'altra estremità. All'apertura, i processi ricevono un *identificatore privato* da zero a `MAX_OPEN_PIPES` meno uno poi lo utilizzano per riferisi all'estremità della pipe durante le successive operazioni di lettura o scrittura. Un processo può aprire più di una estremità di qualunque pipe, fino ad un massimo di `MAX_OPEN_PIPES`. Le estremità devono essere *chiuse* quando non servono più. Quando un processo termina, il sistema provvede comunque a chiudere tutte le estremità che risultavano ancora aperte dal processo. Le operazioni di lettura o scrittura sono sincrone: lo scrittore di blocca in attesa di aver trasferito tutti i byte, e il lettore si blocca in attesa di averli ricevuti tutti. Se una delle due estremità viene chiusa mentre un processo è in attesa sull'altra, il processo si risveglia dalla primitiva di lettura o scrittura e riceve il valore `false`.

Per realizzare le `pipe` aggiungiamo le seguenti primitive (abortiscono il processo in caso di errore):

- `natl openpipe(natl pipeid, bool writer)` (tipo 0x2a, già realizzata): Apre l'estremità di lettura (se `writer` è `false`) o di scrittura (se `writer` è `true`) della pipe con identificatore `pipeid`. È un errore se la pipe non esiste, o se il processo ha troppe pipe aperte. Restituisce l'identificatore privato della pipe, o `0xFFFFFFFF` se l'estremità della pipe è già aperta (dallo stesso o da un altro processo).
- `bool writepipe(natl slotid, const char *buf, natl n)` (tipo 0x2b, realizzara in parte): Invia `n` caratteri dal buffer `buf` sulla pipe con identificatore privato `slotid`. È un errore se la pipe `slotid` non è l'identificatore valido di una pipe aperta in scrittura, o se il buffer non è accessibile in lettura a tutti i processi. Restituisce `false` se non è stato possibile trasferire tutti i byte.
- `bool readpipe(natl slotid, char *buf, natl n)` (tipo 0x2c, realizzata in parte): Riceve `n` caratteri dal dalla pipe di identificatore privato `slotid` e li scrive nel buffer `buf`. È un errore se la pipe `slotid` non è l'identificatore valido di una pipe aperta in lettura, o se il buffer non è accessibile in scrittura da tutti i processi. Restituisce `false` se non è stato possibile ricevere tutti i byte.
- `void closepipe(natl slotid)` (tipo 0x2d, da realizzare): Chiude una estremità di una pipe. È un errore se `slotid` non è l'indentificatore privato valido di una estremità di pipe ancora aperta.

Per descrivere una `pipe` aggiungiamo al nucleo la seguente struttura dati:

```
struct des_pipe {
    natl writer;
    natl reader;
    natl w_pending;
    const char *w_buf;
    natl r_pending;
    char *r_buf;
};
des_pipe array_despipe[MAX_PIPES];
```

Il campo `writer` identifica il processo che ha aperto l'estremità di scrittura, e il campo `reader` identifica il processo che ha aperto l'estremità di lettura. La pipe è libera se entrambi sono zero. I campi assumono il valore `0xFFFFFFFF` se la corrispondente è estremità è stata chiusa, ma l'altra è ancora aperta. Il campo `w_buf` punta al prossimo byte da trasferire dal buffer dello scrittore, ed è diverso da `nullptr` solo se lo scrittore è bloccato in attesa di completare il trasferimento. Analogamente, il capo `r_buf` punta alla prossima locazione da riempire nel buffer del ricevitore, ed è diverso da `nullptr` solo se il lettore è bloccato in attesa di ricevere tutti i byte. I campi `w_pending` e `r_pending` contengono il numero di byte ancora da scrivere o leggere, rispettivamente.

Inoltre, aggiungiamo il seguente campo ai descrittori di processo:

```
struct des_proc {
    ...
    natl mypipes[MAX_OPEN_PIPES];
}
```

L'array è indicizzato tramite gli identificatori privati. Ogni entrata contiene `0xFFFFFFFF` se l'identificatore non è usato, oppure contiene l'identificatore della pipe di cui il processo ha aperto una estremità.

```
des_proc* crea_processo(void f(natq), natq a, int prio, char liv)
{
    des_proc*   p;
    ...
    // Mettere tutte le pipes di un processo a 0xFFFFFFFF di default
    for (natl s = 0; s < MAX_OPEN_PIPES; s++) p->mypipes[s] = 0xFFFFFFFF;
    ...
}
```

Modificare il file `sistema.cpp` in modo da realizzare le parti mancanti.

```
void distruggi_processo(des_proc* p)
{
    // Chiude le pipe ancora aperte associate al progetto
    for (natl s = 0; s < MAX_OPEN_PIPES; s++)
    {
```

```
        natl pi = p->mypipes[s];
        if (pi == 0xFFFFFFFF)
            continue; // Pipe già chiusa
        do_closepipe(pi);
    }
    ...
}

natl find_free_slot()
{
    // Trovo la prima pipe libera del processo corrente o 0x
    for (natl i = 0; i < MAX_OPEN_PIPES; i++)
        if (esecuzione->mypipes[i] == 0xFFFFFFFF)
            return i;
    return 0xFFFFFFFF;
}
extern "C" void c_openpipe(natl p, bool writer)
{
    if (p >= MAX_PIPES) {
        flog(LOG_WARN, "openpipe: id '%u' non valido", p);
        c_abort_p();
        return;
    }

    natl slot = find_free_slot();
    if (slot == 0xFFFFFFFF) {
        flog(LOG_WARN, "openpipe: troppe pipe aperte");
        c_abort_p();
        return;
    }

    des_pipe *dp = &array_despipe[p];
    natl otherproc, *myrole;
    if (writer) {
        myrole = &dp->writer;
        otherproc = dp->reader;
    } else {
        myrole = &dp->reader;
        otherproc = dp->writer;
    }
    esecuzione->contesto[I_RAX] = 0xFFFFFFFF;
    if (*myrole)
        return;
    if (otherproc) {
        inspronti();
        inserimento_lista(pronti, proc_table[otherproc]);
```

```
    }
    *myrole = esecuzione->id;
    esecuzione->mypipes[slot] = p;
    esecuzione->contesto[I_RAX] = slot;
    schedulatore();
}

extern "C" void c_writepipe(natl s, const char *buf, natl n)
{
    if (s >= MAX_OPEN_PIPES) {
        flog(LOG_WARN, "writepipe: slot '%u' non valido", s);
        c_abort_p();
        return;
    }
    if (!c_access(int_cast<vaddr>(buf), n, false)) {
        flog(LOG_WARN, "writepipe: buf non valido");
        c_abort_p();
        return;
    }
    natl p = esecuzione->mypipes[s];
    if (p == 0xFFFFFFFF) {
        flog(LOG_WARN, "writepipe: slot '%u' non aperto", s);
        c_abort_p();
        return;
    }
    struct des_pipe *dp = &array_despipe[p];

    if (dp->writer != esecuzione->id) {
        flog(LOG_WARN, "writepipe: pipe non aperta in scrittura");
        c_abort_p();
        return;
    }

    if (dp->reader == 0xFFFFFFFF) {
        esecuzione->contesto[I_RAX] = false;
        return;
    }

    for ( ; dp->r_pending && n; dp->r_pending--, n--)
        *dp->r_buf++ = *buf++;

    if (n) {
        dp->w_pending = n;
        dp->w_buf = buf;
    } else {
        dp->w_buf = nullptr;
```

```
        esecuzione->contesto[I_RAX] = true;
        inspronti();
    }

    if (!dp->r_pending && dp->r_buf) {
        dp->r_buf = nullptr;
        des_proc *reader = proc_table[dp->reader];
        reader->contesto[I_RAX] = true;
        inserimento_lista(pronti, reader);
    }

    schedulatore();
}

extern "C" void c_readpipe(natl s, char *buf, natl n)
{
    if (s >= MAX_OPEN_PIPES) {
        flog(LOG_WARN, "readpipe: slot '%u' non valido", s);
        c_abort_p();
        return;
    }

    if (!c_access(int_cast<vaddr>(buf), n, true)) {
        flog(LOG_WARN, "readpipe: buf non valido");
        c_abort_p();
        return;
    }

    natl p = esecuzione->mypipes[s];
    if (p == 0xFFFFFFFF) {
        flog(LOG_WARN, "readpipe: slot '%u' non aperto", s);
        c_abort_p();
        return;
    }
    struct des_pipe *dp = &array_despipe[p];

    if (dp->reader != esecuzione->id) {
        flog(LOG_WARN, "readpipe: pipe non aperta in lettura");
        c_abort_p();
        return;
    }

    if (dp->writer == 0xFFFFFFFF) {
        esecuzione->contesto[I_RAX] = false;
        return;
    }
```

```
    for ( ; dp->w_pending && n; dp->w_pending--, n--)
        *buf++ = *dp->w_buf++;

    if (n) {
        dp->r_pending = n;
        dp->r_buf = buf;
    } else {
        dp->r_buf = nullptr;
        esecuzione->contesto[I_RAX] = true;
        inspronti();
    }

    if (!dp->w_pending && dp->w_buf) {
        dp->w_buf = nullptr;
        des_proc *writer = proc_table[dp->writer];
        writer->contesto[I_RAX] = true;
        inserimento_lista(pronti, writer);
    }

    schedulatore();
}

// Chiude la pipe p
void do_closepipe(natl p)
{
    struct des_pipe *dp = &array_despipe[p];
    natl otherproc, *myrole;
    const char* otherbuf;

    if (esecuzione->id == dp->writer) {
        myrole = &dp->writer;
        otherproc = dp->reader;
        otherbuf = dp->r_buf;
    } else {
        myrole = &dp->reader;
        otherproc = dp->writer;
        otherbuf = dp->w_buf;
    }

    // Chiude pipe dal lato dell'altro processo
    if (otherproc == 0xFFFFFFFF) {
        dp->writer = dp->reader = 0;
        dp->w_pending = dp->r_pending = 0;
        dp->w_buf = dp->r_buf = nullptr;
        return;
```

```
    }
    *myrole = 0xFFFFFFFF;
    if (otherbuf) {
        des_proc *op = proc_table[otherproc];
        if (otherproc == dp->reader)
            dp->r_buf = nullptr;
        else
            dp->w_buf = nullptr;
        op->contesto[I_RAX] = false;
        inserimento_lista(pronti, op);
    }
}

extern "C" void c_closepipe(natl s)
{
    if (s >= MAX_OPEN_PIPES) {
        flog(LOG_WARN, "closepipe: slot '%u' non valido", s);
        c_abort_p();
        return;
    }
    natl p = esecuzione->mypipes[s];
    if (p == 0xFFFFFFFF) {
        flog(LOG_WARN, "closepipe: pipe non aperta");
        c_abort_p();
        return;
    }
    inspronti();
    do_closepipe(p);
    esecuzione->mypipes[s] = 0xFFFFFFFF;
    schedulatore();
}
```