2023-06-07

Memoria Copy-On-Write

Aggiungiamo una nuova zona "utente/cow" alla memoria virtuale dei nuovi processi utente. Il contenuto di questa zona è inizializzato all'avvio del sistema e ogni processo ne possiede una copia privata.

Invece di copiare l'intera zona ogni volta che creiamo un nuovo processo, adottiamo la tecnica del *copy on write* (abbreviato in *cow*): tutti i processi condividono inizialmente la stessa zona, ma in solo lettura, e copiamo le singole pagine se e solo se un processo tenta di scrivervi.

Più in dettaglio:

- all'avvio del sistema creiamo e inizializziamo la zona cow, allocando e inizializzando i frame e creando le opportune traduzioni partendo da una tabella di livello 4 che chiamiamo `cow_root`, avendo cura di vietare le scritture in tutti gli indirizzi della zona;
- alla creazione di ogni processo utente copiamo le entrate opportune di `cow_root` nella tabella radice del nuovo processo;
- se un processo genera un page fault per accesso in scrittura su un indirizzo appartenente alla zona utente/cow, invece di terminare il processo copiamo la corrispondente pagina, abilitiamo l'accesso in scrittura e facciamo ripartire il processo.

Attenzione: l'accesso in scrittura deve essere abilitato solo per il processo che ha generato il fault, e solo sulla pagina che contiene l'indirizzo cha ha causato il fault. Inoltre, se l'accesso *non* appartiene alla zona utente/cow del processo, il processo deve essere abortito come al solito.

```
natl pf_counter = 0;    ///< numero di page fault
vaddr last_pf_v;    ///< indirizzo virtuale dell'ultimo page fault
vaddr last_pf_rip;  ///< istruzione che ha causato l'ultimo page fault
natq  last_pf_error;    ///< codice di errore dell'ultimo page fault
extern "C" void gestore_eccezioni(int tipo, natq errore, vaddr rip)
{
    ...
    if (tipo == 14)
    {
        vaddr v = readCR2();
        pf_counter++; // solo per debug
        if (pf_counter) {
            if (last_pf_v == v && last_pf_rip == rip && last_pf_error == errore) {
                panic("PAGE FAULT NON RISOLTO");
            }
            last_pf_v = v;
```

```
            last_pf_rip = rip;
            last_pf_error = errore;
        }
        if (aggiorna_cow_privata(v))
            return;
    }
    ...
}
void init_frame()
{
    ...
    // Solo per debug
    memset(voidptr_cast(primo_frame_libero * DIM_PAGINA), 0xaa, N_M2 * DIM_PAGINA);
    ...
}

const vaddr ini_utn_w = norm(I_UTN_W * PART_SIZE); ///< base di della zona cow
const vaddr fin_utn_w = ini_utn_w + PART_SIZE * N_UTN_W; ///< limite della zona cow

des_proc* crea_processo(void f(natq), natq a, int prio, char liv)
{
    ...
    copia_cow_condivisa(p->cr3);
}
void distruggi_processo(des_proc* p)
{
    distruggi_cow_privata();
    ...
}
```

Per realizzare il meccanismo appena descritto sono state definite le nuove costanti `ini_utn_w` e `fin_utn_w`, che delimitano gli indirizzi riservati alla nuova zona, e la costante `DIM_USR_COW` che ne specifica la dimensione effettiva. Inoltre sono state aggiunte le seguenti funzioni, chiamate nei punti opportuni:

- `bool crea_cow_condivisa()` (chiamata durante l'inizializzazione del sistema): crea e inizializza la zona cow iniziale, con dimensione pari a `DIM_USR_COW`, visibile a partire dall'indirizzo `ini_utn_w`; la zona deve inizialmente contenere solo byte nulli; restituisce `false` se non è stato possibile creare la zona, `true` altrimenti;
- `void copia_cow_condivisa()` (chiamata durante la creazione di un processo): copia le entrate opportune di `cow_root` nella tabella radice del nuovo processo;
- `aggiorna_cow_privata(vaddr v)` (chiamata durante un page fault): se `v` cade nella zona utente/cow effettua la copia, aggiorna la traduzione come descritto e restituisce `true`; se `v` non cade nella zona utente/cow, o se

l'aggiornamento fallisce per qualche motivo, restiuisce `false`;

- `void distruggi_cow_privata()` (chiamata durante la distruzione di un processo): disfa quanto eventualmente fatto nelle precedenti `copia_cow_condivisa()` e `aggiorna_cow_privata()`.

```
paddr cow_root;
void copia_cow_condivisa(paddr dest)
{
    copy_des(cow_root, dest, I_UTN_W, N_UTN_W);
}
void distruggi_cow_privata()
{
    tab_iter it(esecuzione->cr3, ini_utn_w, DIM_USR_COW);
    for (it.post(); it; it.next_post()) {
        tab_entry& e = it.get_e();
        if (!(e & BIT_P))
            continue;
        int liv = it.get_l();
        // set il bit RW è settato il frame puntato
        // deve essere stato allocato da aggiorna_cow_privata,
        // quindi dobbiamo deallocarlo
        if (e & BIT_RW) {
            paddr f = extr_IND_FISICO(e);
            if (liv > 1) {
                set_des(f, 0, 512, 0);
                rilascia_tab(f);
            } else {
                rilascia_frame(f);
            }
        }
        // azzeriamo l'entrata corrente. Se la tabella corrente
        // è la radice dobbiamo farlo in ogni caso, per disfare
        // quanto fatto in copia_cow_condivisa()
        if (liv == MAX_LIV || (e & BIT_RW)) {
            e = 0;
            dec_ref(it.get_tab());
        }
    }
}
```

Modificare il file `sistema.cpp` scrivendo le parti mancanti tra i tag `SOLUZIONE`.

```
bool crea_cow_condivisa()
{
    cow_root = alloca_tab();
    if (!cow_root)
        return false;
```

```
    vaddr v = map(cow_root, ini_utn_w, ini_utn_w + DIM_USR_COW, BIT_US,
            [](vaddr v) {
                paddr f = alloca_frame();
                memset(voidptr_cast(f), 0, DIM_PAGINA);
                return f;
            });
    if (v != ini_utn_w + DIM_USR_COW) {
        unmap(cow_root, ini_utn_w, v,
            [](vaddr, paddr p, int) {
                rilascia_frame(p);
            });
        rilascia_tab(cow_root);
        cow_root = 0;
        return false;
    }
    return true;
}

// aggiorna la cow privata in modo che l'indirizzo v sia scrivibile
// dal processo corrente
bool aggiorna_cow_privata(vaddr v)
{
    if (esecuzione->livello != LIV_UTENTE || v < ini_utn_w || v - ini_utn_w >= DIM_USR_COW)
        return false;

// Per rendere l'indirizzo scrivibile dobbiamo settare BIT_RW su tutto il
// percorso di traduzione, ma siccome dobbiamo farlo solo per il processo
// corrente, dobbiamo creare una copia privata di tutte le tabelle sul percorso
// e del frame finale.  Dobbiamo anche stare attenti a non creare copie di cose
// che avevamo già copiato. Per fortuna possiamo riconoscere le entità già
// copiate osservando il BIT_RW del descrittore che le punta: se è già settato
// deve per forza trattarsi di una tabella o frame privato.
//
// Operativamente, usiamo un tab_iter per discendere lungo il percorso di
// traduzione partendo dal livello 4. Ad ogni passo osserviamo il tab_entry su
// cui si è fermato l'iteratore: se BIT_RW è già settato non abbiamo altro da
// fare a questo livello e possiamo proseguire. Altrimenti dobbiamo creare una
// copia dell'entità di livello inferiore, redirigere il puntatore nel
// tab_entry e settare BIT_RW. Notare che, ad ogni passo, il tab_entry su cui
// si ferma l'iteratore è sicuramente privato: la priva volta perché si trova
// nella tabella radice, che é già privata per ogni processo, e nei passi
// successivi perché abbiamo creato una tabella privata al passo precedente (se
// non lo era già).
    for (tab_iter it(esecuzione->cr3, v, 1); it; it.next()) {
        // prendiamo un riferimento all'entrata corrente
        tab_entry& e = it.get_e();
```

```
        // aggiorna_cow_privata() è chiamata solo per gli indirizzi
        // della zona cow, che dovrebbero essere tutti validi. Se
        // troviamo P==0 ci deve essere un errore.
        if (!(e & BIT_P))
            fpanic("indirizzo cow %lx non mappato", v);
        if (e & BIT_RW)
            continue;
        paddr new_frame;
        // estraiamo il puntatore alla entità (tabella o frame)
        // corrente
        paddr old_frame = extr_IND_FISICO(e);
        if (it.get_l() > 1) {
            // se siamo ad un livello maggiore di 1, l'entità
            // puntata è una tabella. Per copiarla usiamo
            // copy_des(), che aggiorna correttamente il
            // contatore delle entrate valide
            new_frame = alloca_tab();
            if (!new_frame)
                return false;
            copy_des(old_frame, new_frame, 0, 512);
        } else {
            // se il livello è 1, l'entità puntata è
            // un frame. Possiamo copiarla con una semplice
            // memcpy()
            new_frame = alloca_frame();
            if (!new_frame)
                return false;
            memcpy(voidptr_cast(new_frame),
                   voidptr_cast(old_frame), DIM_PAGINA);
            // visto che modificheremo il permesso RW per questo
            // indirizzo, invalidiamo la corrispondente entrata nel TLB
            //
            // Nota: in questo caso l'invalidazione non è
            // strettamente necessaria. Visto che l'indirizzo era
            // non scrivibile, il bit D copiato nel TLB sarà
            // sicuramente 0. Questo vuol dire che il TLB genererà
            // un miss forzato al primo tentativo di scrittura, e
            // quindi la MMU sarà costretta a percorrere il TRIE e
            // vedere il nuovo valore dei bit RW.
            invalida_entrata_TLB(it.get_v());
        }
        // redirigiamo l'entrata corrente e settiamo RW
        set_IND_FISICO(e, new_frame);
        e |= BIT_RW;
    }
    return true;
```

```
}
```