**INTRODUZIONE**

Ci sono tre tipologie di prove sul nucleo:

* Sistema;
* I/O;
* Memoria virtuale.

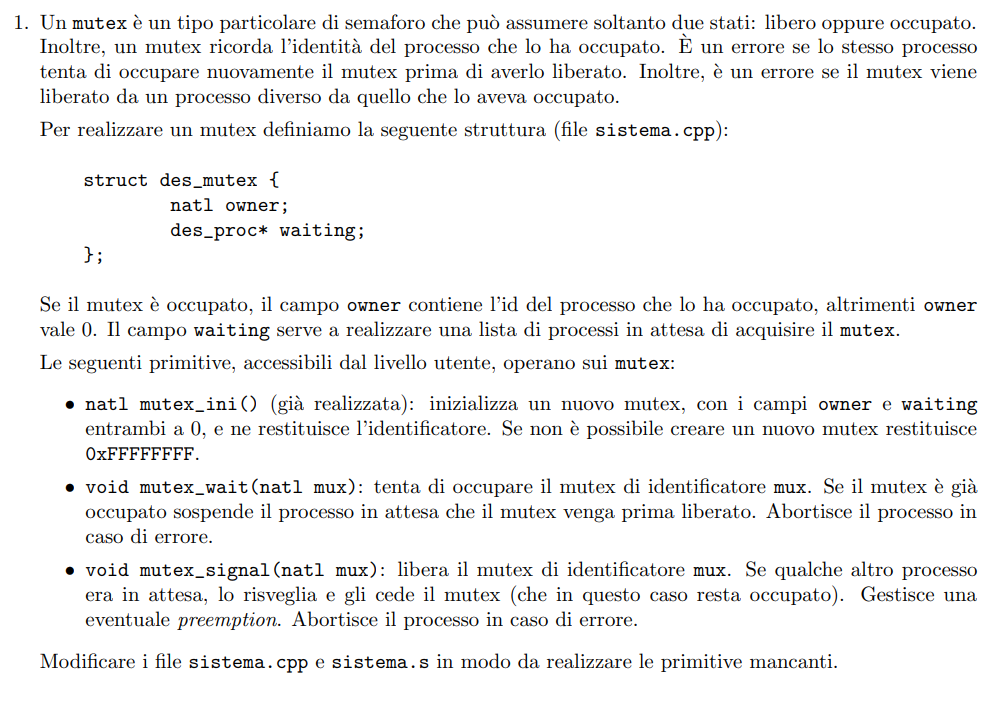
Tra i tag “esame” ci sono tutte le aggiunte che sono state fatte;

Tra i tag “soluzione” dobbiamo scrivere noi qualcosa. Dove non ci sono questi tag dunque non va scritto nulla.

Utente.cpp trascuriamolo sempre, è di difficile comprensione.

**ESERCITAZIONI SISTEMA**

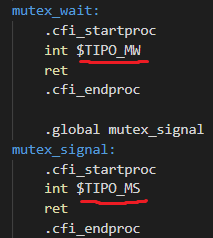
**PROVA D’ESAME 05/10/10**



**SOLUZIONE**

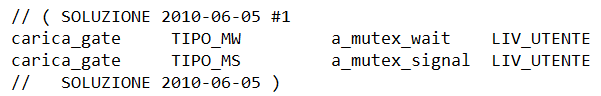
**Partiamo dal modificare sistema.s.** Innanzitutto, **vedendo i tag “soluzione”, i primi compaiono dove bisogna caricare il gate relativo al tipo di interruzione associato alla routine *a\_nomeprimitiva*.**

* Per recuperare il tipo basta andare a vedere il file utente.s, dove è stata definita la funzione di supporto che fa INT $TIPO, e recuperare il tipo associato alla due routine;

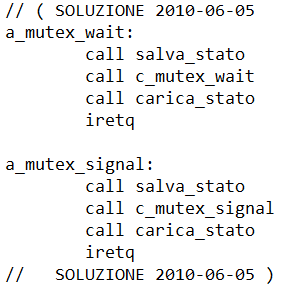


* Le routine le chiameremo sempre *a\_nomeprimitiva* (la parte Assembly) e *c\_nomeprimitiva* (la parte C++), quindi ad esempio *a\_mutex\_wait* e *a\_mutex\_signal*;
* Il DPL necessario si legge dal testo: visto che dice che si vuole fornire le primitive all’utente, allora è LIV\_UTENTE.

Allora carichiamo i gate:

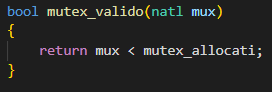


**Andando a vedere dove sono i prossimi tag “soluzione”, questi compaiono dove bisogna scrivere la parte Assembly della primitiva.** Senza sapere né leggere né scrivere, basta copiare la struttura delle altre routine pari pari mettendo al posto di c\_*primitiva* il nome delle parti C++ delle primitive che dobbiamo implementare:



**Modifichiamo ora sistema.cpp.**

Innanzitutto, può essere utile vedere quali altre aggiunte sono state fatte al file sistema.cpp. Oltre a quelle già scritte nel testo d’esame, ci interessa una funzione *mutex\_valido*, che verifica, dato un id, se è valido o meno (e cioè se è l’id di un mutex effettivo o meno):

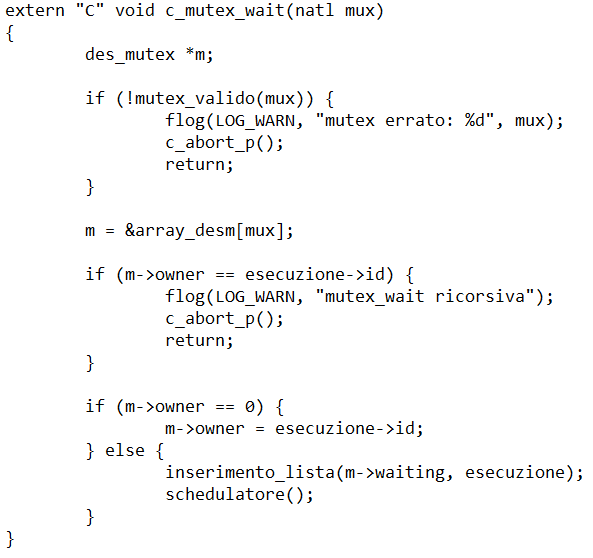


Lo spazio per tutti i mutex è stato già allocato in un array *array\_desm*, e c’è una variabile *mutex\_allocati* che tiene conto di quanti sono i mutex effettivamente allocati. I mutex vengono allocati in maniera sequenziale nell’array, e l’indice all’interno dell’array corrisponde all’identificatore del semaforo, per cui sono validi (e cioè fanno riferimento a effettivi mutex) tutti gli id che vanno da 0 a mutex\_allocati escluso.

Allora, se mux < mutex\_allocati, l’identificatore è valido; altrimenti, no.

Vedendo ora dove sono i tag “soluzione”, questi si trovano dove bisogna implementare la parte C++ delle primitive. **NB:** poiché in sistema.s chiameremo sempre la parte C++ scrivendo semplicemente call c\_*nomeprimitiva,* laparte C++ di ogni primitiva va dichiarata mettendo prima **extern “C”.**

Partiamo dalla c\_mutex\_wait:



Innanzitutto, **ogni volta che si ha una primitiva che deve essere chiamata dall’utente, poiché di lui non ci si può fidare, si fa il controllo sull’input.**

Come prima cosa dobbiamo controllare l’id del mutex passato dall’utente mutex sia valido, e quindi usiamo la funzione mutex\_valido. Se non è valido, la traccia ci dice che in caso di errore si deve abortire, e quindi facciamo c\_abort\_p() e poi ritorniamo al chiamante facendo return.

**NB1**: nelle tracce in cui si modifica il sistema si chiama c\_abort\_p(), e non abort\_p() (che tra l’altro, il sistema non può neanche chiamarla, perché non è stata definita una funzione di supporto abort\_(), ma se esistesse, questa invocherebbe proprio la primitiva abort\_p(), e quindi si avrebbero due salva\_stato consecutive, e si avrebbero problemi).

**NB2:** ricorda sempre di mettere return dopo aver abortito. Non è che dopo la abort muore tutto: questa termina il processo che prima era in esecuzione e poi ne schedula un altro, e affinché vada in esecuzione, c’è bisogno di ritornare al chiamante, e cioè alla parte Assembly della primitiva, che facendo carica\_stato e iretq metterà in esecuzione un nuovo processo.

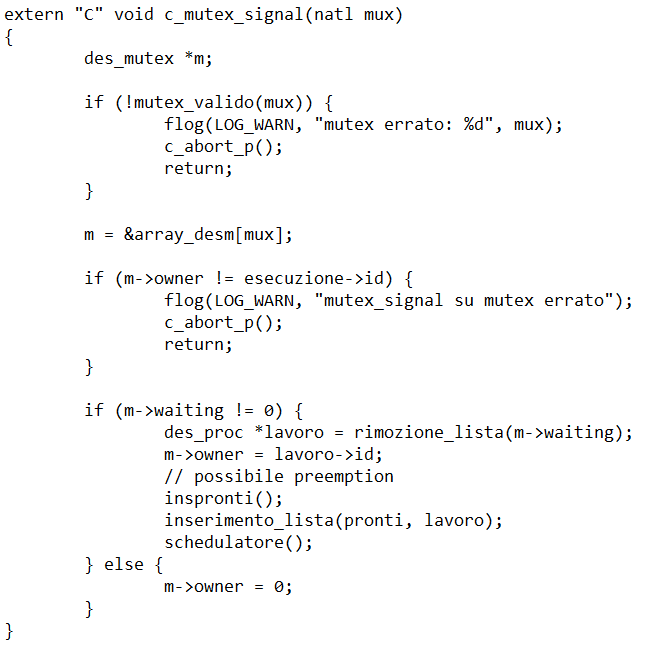
Superato questo primo controllo, significa che l’id del mutex passato dall’utente mutex è valido, e dunque recuperiamo il des\_mutex del mutex che ha id == mux. In particolare, mettiamo il suo indirizzo dentro una variabile puntatore a des\_mutex, che chiamiamo *m,* e quindi facciamo m = &array\_desm[mux] (prendiamo l’indirizzo, così se dobbiamo fare modifiche, facendo m->… modifichiamo proprio il des\_mutex all’interno dell’array).

Ora dobbiamo vedere se lo stesso processo sta tentando di occupare nuovamente il mutex prima di averlo liberato. Il processo che ha chiamato la primitiva è quello il cui des\_proc è puntato dal puntatore *esecuzione*, e il suo id è esecuzione->id; il campo owner del des\_mutex contiene l’id del processo che lo ha occupato. Allora, se lo stesso processo sta tentando di occupare nuovamente il mutex prima di averlo liberato, si avrebbe m->owner == esecuzione->id. Allora, in questo caso, poiché è errore, si abortisce e si fa return.

Superato questo controllo, bisogna vedere se effettivamente si può occupare il mutex, e cioè se è libero o meno.

* Se m->owner == 0, allora è libero, e quindi si può occupare. Per indicare che è occupato, basta mettere dentro m->owner l’id del processo che ha richiesto di occupare il mutex, e cioè   
  esecuzione->id;
* Se m->owner != 0, significa che il mutex non è libero, e quindi il processo deve essere messo in attesa. Inseriamo allora il des\_proc del processo che ha richiesto di occupare il mutex, e cioè quello puntato da *esecuzione,* nella lista m->waiting, che è la lista dei processi in attesa di acquisire il mutex. Chiamiamo poi schedulatore() per schedulare un nuovo processo (e cioè, togliere il des\_proc in testa alla lista *pronti* e assegnare l’indirizzo del suo des\_proc al puntatore esecuzione).

Facciamo ora la c\_mutex\_signal:



Anche in questo caso, poiché la primitiva va invocata dall’utente, si fa il controllo sull’input.

Come prima cosa verifichiamo che l’id passato dall’utente sia valido o meno.

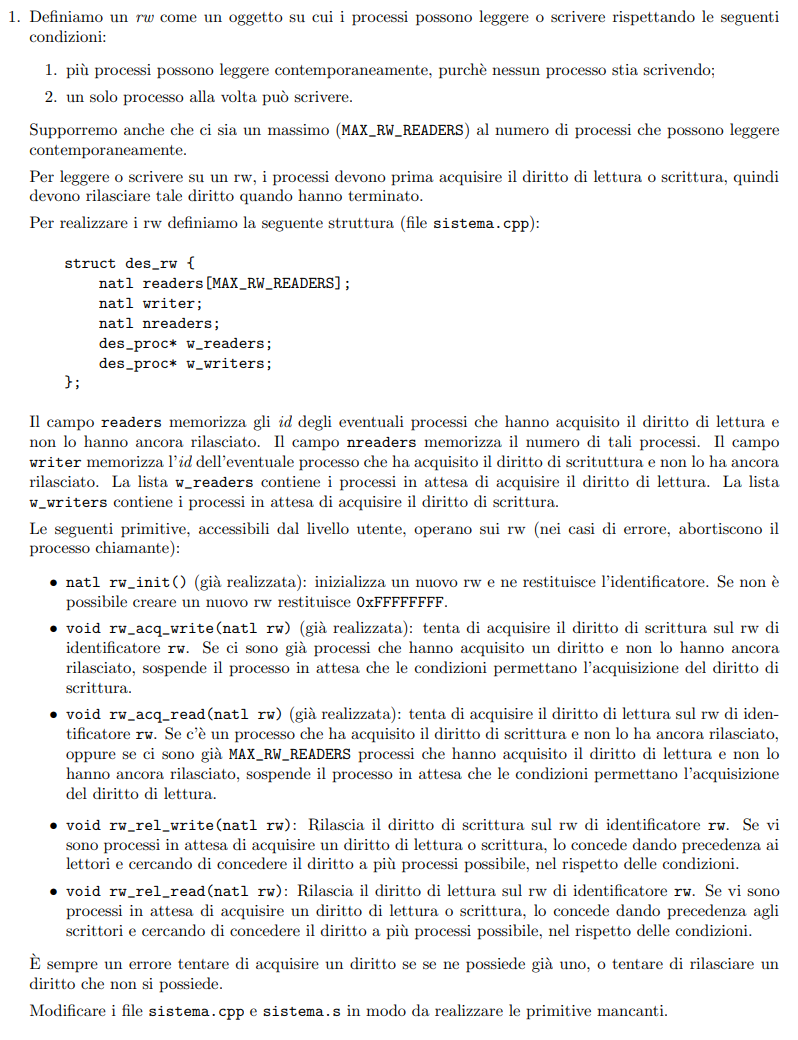
Se è valido, recuperiamo il des\_sem del mutex che ha tale id. Poiché è un errore se il mutex viene liberato da un processo diverso da quello che lo aveva occupato, si vede se m->owner != esecuzione->id e, se sì, si abortisce. Questo serve anche a bloccare il caso in cui si voglia liberare un semaforo già libero, perché in tal caso m->owner == 0, e sicuramente sarà != esecuzione->id.

Superato anche questo controllo, significa che si può liberare il semaforo. Se m->waiting != 0, significa che c’è qualche processo in attesa del mutex, e si risveglia quello di maggiore priorità, che è in testa alla lista. Quello che facciamo allora è togliere dalla lista m->waiting il des\_proc in testa alla lista e assegnare l’indirizzo ad un nostro puntatore a des\_proc, che chiamiamo lavoro. Mettiamo poi in m->owner l’id del processo appena tolto dalla lista, quindi m->owner = lavoro->id. A questo punto, **per gestire una possibile preemption, si fa così:**

* **inseriamo il des\_proc puntato da *esecuzione* in testa alla lista pronti con inspronti();**
* **inseriamo il des\_proc appena tolto dalla lista m->waiting nella lista pronti usando inserimento\_lista(…, …);**
* **chiamiamo schedulatore().**

**In questo modo, verrà correttamente schedulato il processo a maggiore priorità tra tutti, che è o quello precedentemente in esecuzione, o quello che è stato appena risvegliato solo se ha precedenza maggiore del processo precedentemente in esecuzione** (infatti, perché la inspronti() mette il des\_proc del processo precedentemente in esecuzione in testa alla lista. Questo è corretto, perché se un processo era in esecuzione significa che aveva priorità maggiore fra tutti quelli che si trovavano in lista pronti. Il des\_proc del processo risvegliato finirà in testa alla lista con la inserimento\_lista quindi solo se ha priorità strettamente maggiore di tutti, compreso quello precedentemente in esecuzione. Dopo schedulatore() il des\_proc in testa alla lista pronti verrà tolto da lì e messo in *esecuzione*).

**PROVA D’ESAME 05/02/11**

****

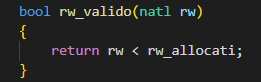
**SOLUZIONE**

**Partiamo dal modificare sistema.s.**

Andando a cercare il tag “soluzione”, vediamo che non troviamo nulla. Infatti, se cerchiamo il tag “esame”, vediamo che sono stati già caricati i gate relativi ai tipi di interruzioni associati alle routine *a\_nomeprimitiva*, e sono state già scritte le routine in Assembly. Non dobbiamo fare nulla.

**Modifichiamo ora sistema.cpp.**

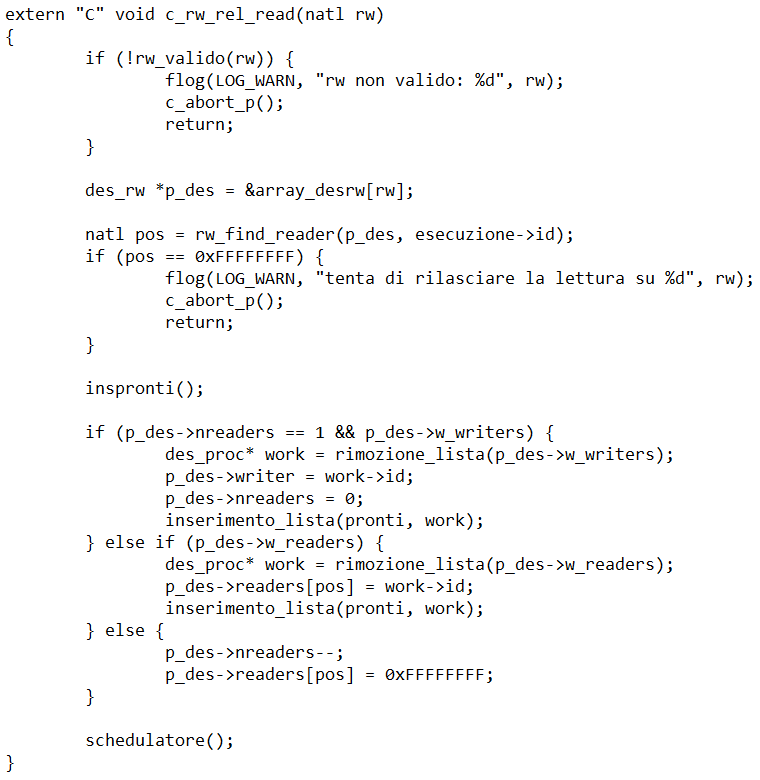
Andiamo a vedere quali aggiunte sono state fatte al file sistema.cpp. Oltre a quelle già descritte nel testo d’esame, c’è anche una funzione *rw\_valido* che verifica, dato un id, se è valido o meno (e cioè se è l’id di un rw effettivo o meno):



Lo spazio per tutti i rw è stato già allocato in un array *array\_desrw*, e c’è una variabile *rw\_allocati* che tiene conto di quanti sono i mutex effettivamente allocati. I mutex vengono allocati in maniera sequenziale nell’array, e l’indice all’interno dell’array corrisponde all’identificatore dell’rw, per cui sono validi (e cioè fanno riferimento effettivi rw) tutti gli id che vanno da da 0 a rw\_allocati escluso.

Allora, se rw < rw\_allocati, l’identificatore è valido; altrimenti, no.

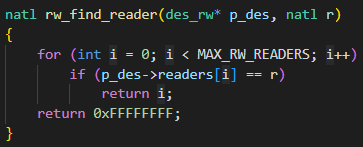
Vedendo ora dove sono i tag “soluzione”, questi si trovano dove bisogna implementare la parte C++ delle primitive. Partiamo dalla c\_rw\_rel\_read:



Prima controlliamo che l’input sia valido con rw\_valido: se non è valido, c\_abort\_p e return.

Se è valido, recuperiamo il des\_rw di indice rw all’interno dell’array, assegnando il suo indirizzo ad un puntatore *p\_des* (prendiamo l’indirizzo, così se dobbiamo fare modifiche, facendo *p\_des*->… modifichiamo proprio il des\_rw all’interno dell’array).

A questo punto, la traccia ci dice che è un errore tentare di rilasciare un diritto che non si possiede. Dobbiamo allora vedere se dentro l’array p\_des->readers[] c’è l’id del processo che ha invocato la primitiva, e cioè esecuzione->id. Questo controllo è fatto dalla funzione *rw\_find\_reader* già implementata:



Passando un puntatore a des\_rw *p\_des* e l’id di un processo *r*, vede se *r* compare all’interno dell’array   
p\_des->readers[]; se sì, restituisce l’indice all’interno dell’array in cui si trova *r*; se no, restituisce 0xFFFFFFFF.

Chiamiamo allora la funzione rw\_find\_reader passando p\_des e esecuzione->id e restituiamo il risultato in *pos:* se *pos* == 0xFFFFFFFF, significa che l’id del processo che ha invocato la primitiva non c’è tra i readers, ed essendo un errore, c\_abort\_p() e return. Altrimenti, in *pos* abbiamo l’indice all’interno dell’array in cui si trova l’id del processo che ha invocato la primitiva.

A questo punto, sono stati fatti tutti i controlli, e quindi si può rilasciare il diritto di lettura. Innanzitutto, poiché qualche processo si può potenzialmente risvegliare, all’inizio mettiamo il processo attualmente in *esecuzione* in testa alla lista pronti usando inspronti(), e poi alla fine fine chiamiamo schedulatore(). In questo modo gestiamo anche un’eventuale preemption.

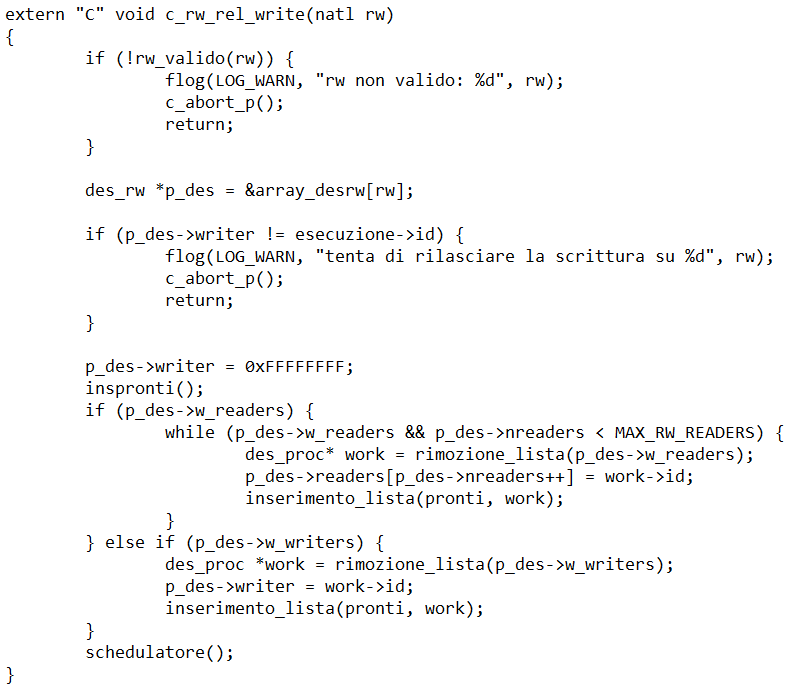
La traccia inoltre mi dice che se ci sono altri processi in attesa di acquisire un diritto di lettura o scrittura, va data precedenza agli scrittori e cercando di concedere il diritto a più processi possibile, nel rispetto delle condizioni. In particolare, ricordiamo che :

* Più processi per volta possono scrivere, purché nessun processo stia scrivendo;
* Un solo processo per volta può scrivere.

Quindi vanno distinti vari casi:

* Primo caso: si possono risvegliare degli scrittori in attesa. Se fino ad adesso c’era un solo lettore, che è quello a cui stiamo togliendo il diritto di leggere, e ci sono dei processi in attesa di scrivere, allora poiché poi non ci sarà più alcun lettore, possiamo effettivamente risvegliare un processo che era in attesa per scrivere (uno solo e solo lui, poiché quando un processo scrive, nessuno può leggere, e può scrivere un solo processo per volta). Allora risvegliamo il processo che è in testa alla lista in attesa per scrivere, togliendolo dalla lista p\_des->w\_writers e assegnando l’indirizzo del suo des\_proc ad una variabile puntatore a des\_proc ausiliaria *work*. Inseriamo poi il des\_proc puntato da *work* (e quindi il des\_proc del processo da risvegliare) nella lista pronti. Azzeriamo poi   
  p\_des->nreaders, in quanto ora non si ha più alcun lettore, e mettiamo in p\_des->writer l’id del processo che si è appena risvegliato, e quindi work->id.
* Secondo caso: non si possono risvegliare scrittori e ci sono lettori in attesa. Se i lettori sono più di 1 (sicuro non 0, sennò significa che nessuno aveva il diritto di leggere, e quindi si è cercato di togliere il diritto di lettura ad un processo che sicuramente non aveva tale diritto, ma un tentativo del genere è stato intercettato dai controlli precedenti), oppure non si hanno scrittori in attesa, in entrambi i casi togliendo il diritto di lettura ad un processo non si può risvegliare alcuno scrittore. Allora, se si hanno dei lettori in attesa, si possono risvegliare questi, e se ne risveglia quanti più possibile. Quanti? Prima c’era un processo che stava leggendo e comunque c’erano lettori in attesa, e dunque questo significa che prima tutti i posti da reader erano occupati. Dunque si può risvegliare un solo processo, che prende il posto di quello che prima stava leggendo e a cui stiamo togliendo il diritto. Allora risvegliamo il processo che è in testa alla lista in attesa per leggere, togliendolo dalla lista   
  p\_des->w\_readers e assegnando l’indirizzo del suo des\_proc alla variabile ausiliaria *work*. Inseriamo poi il des\_proc puntato da *work* (e quindi il des\_proc del processo da risvegliare) in lista pronti. Mettiamo poi nell’array p\_des\_readers[] l’id del processo che si è appena risvegliato, e cioè   
  work->id, inserendolo alla posizione *pos*, che era la posizione in cui si trovava l’id del processo a cui abbiamo tolto il diritto di lettura (e quindi è una posizione libera). p\_des->nreaders non si aggiorna, poiché complessivamente il numero di lettori è rimasto uguale;
* Terzo caso: non si possono risvegliare né lettori, né scrittori. Se si finisce in questo caso, significa che i due casi precedenti non sono rispettati, e dunque non si può risvegliare niente. Allora a questo punto semplicemente decrementiamo p\_des->nreaders e mettiamo in p\_des->reader[pos] il valore 0xFFFFFFFF, che è il valore per indicare che la posizione a quell’indice dell’array è vuota (si vede guardando la primitiva rw\_init, che inizializza tutti i campi vuoti con 0xFFFFFFFF, e quindi rispettiamo anche noi questo).

Facciamo ora la c\_rw\_rel\_write:



Prima controlliamo che l’input sia valido con rw\_valido: se non è valido, c\_abort\_p e return.

Se è valido, recuperiamo il des\_rw di indice rw all’interno dell’array, assegnando il suo indirizzo ad un puntatore *p\_des* (prendiamo l’indirizzo, così se dobbiamo fare modifiche, facendo *p\_des*->… modifichiamo proprio il des\_rw all’interno dell’array).

A questo punto, la traccia ci dice che è un errore tentare di rilasciare un diritto che non si possiede. Dobbiamo allora vedere se in p\_des->writer c’è proprio l’id del processo che ha invocato la primitiva, e cioè   
esecuzione->id. Se sì, controllo superato; se no, c\_abort\_p e return.

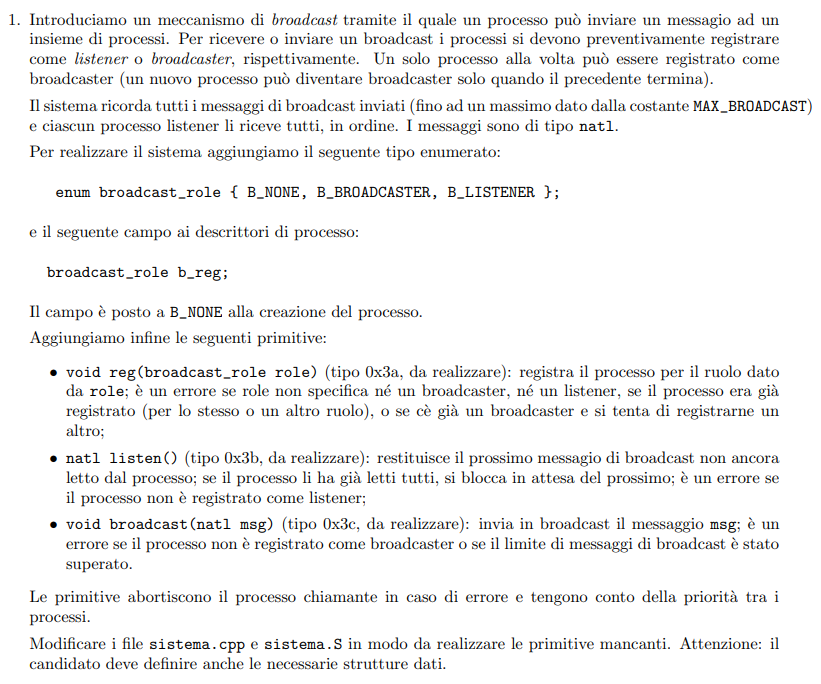
Arrivati qui, sono stati superati tutti i controlli, e quindi si può rilasciare il diritto di scrittura. Questo si fa mettendo in p\_des->writer il valore 0xFFFFFFFF, ad indicare che non c’è più nessuno scrittore. Inoltre, poiché qualche processo si può potenzialmente risvegliare, mettiamo il processo attualmente in *esecuzione* in testa alla lista pronti usando inspronti(), e poi alla fine fine chiamiamo schedulatore(). In questo modo gestiamo anche un’eventuale preemption.

La traccia inoltre mi dice che se ci sono altri processi in attesa di acquisire un diritto di lettura o scrittura, va data precedenza ai lettori, cercando di concedere il diritto a più processi possibile, nel rispetto delle condizioni già viste prima.

Quindi vanno distinti vari casi:

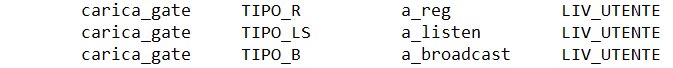
* Primo caso: si possono risvegliare dei lettori in attesa. Ora che non c’è più nessuno scrittore, ci possono essere più lettori, e quindi ne possiamo risvegliare quanti più possibile in attesa. Allora, se effettivamente ci sono lettori in attesa, finché ce ne sono e finché il numero di lettori è minore di MAX\_RW\_READERS, togliamo il *des\_proc* dalla testa della lista p\_des->w\_readers, mettendo il suo indirizzo in una variabile puntatore a des\_proc *work*; inseriamo poi il des\_proc puntato da *work* (e quindi il des\_proc del processo da risvegliare) nella lista pronti; mettiamo nell’array   
  p\_des->readers[] l’id del processo che si è appena risvegliato, e cioè work->id. Per capire in che posizione, ragioniamo: se stiamo rilasciando un diritto di scrittura, significa all’inizio ci sono lettori, e quindi nreaders = 0 e l’array readers è totalmente vuoto (cioè ogni elemento contiene 0xFFFFFFFF). Allora la prima volta possiamo metterlo in readers[0], e incrementare n\_readers (abbiamo un lettore in più); la seconda volta in readers[1], e incrementare n\_reader… Allora come posizione scegliamo proprio p\_des->nreaders; poi incrementiamo p\_des->nreaders (abbiamo un lettore in più).
* Secondo caso: non ci sono lettori in attesa ma ci sono scrittori in attesa. Se non ci sono lettori in attesa ma ci sono scrittori in attesa, poiché ora non c’è più nessuno scrittore e sicuramente non ci sono altri lettori (mentre sta uno scrittore non ci possono essere lettori), possiamo risvegliare uno scrittore (e uno solo, perché ci può essere max uno scrittore).
* Terzo caso: non ci sono né lettori in attesa, né scrittori in attesa. In questo caso allora non si fa altro.

**PROVA D’ESAME 18/01/17**



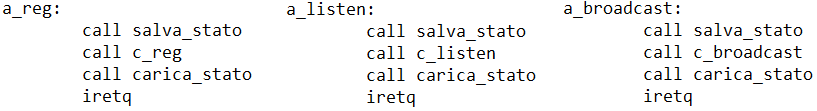
**SOLUZIONE**

**Partiamo dal modificare sistema.s.** Innanzitutto, vedendo i tag “soluzione”, i primi compaiono dove bisogna caricare il gate relativo al tipo di interruzione associato alla routine *a\_nomeprimitiva,* e lo facciamo:



“LIV\_UTENTE” perché devono essere chiamate dall’utente.

Andando a vedere dove sono i prossimi tag “soluzione”, questi compaiono dove bisogna scrivere la parte Assembly della primitiva. Basta copiare la struttura delle altre routine pari pari mettendo al posto di c\_*primitiva* il nome delle parti C++ delle primitive che dobbiamo implementare:



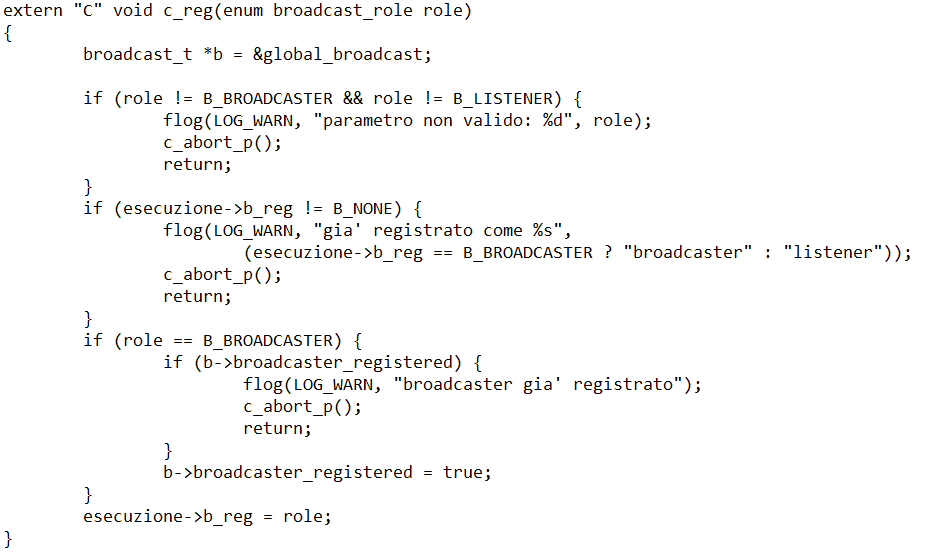
**Modifichiamo ora sistema.cpp.**

Andiamo a vedere quali aggiunte sono state fatte al file sistema.cpp. Come dice la traccia, la struttura di ogni descrittore di processo è stata modificata aggiungendo il campo “broadcast\_role b\_reg”, che specifica il ruolo del processo. Abbiamo anche dei tag “soluzione”, quindi dovremo poi aggiungere qualcosa.

Abbiamo poi una struttura broadcast\_t con un campo broadcaster\_registered, che vale true o false in base se c’è un broadcaster o meno. Abbiamo anche dei tag “soluzione”, quindi dovremo poi aggiungere qualcosa. E’ stata dichiarata anche una variabile di tipo broadcast\_t, di nome global\_broadcast, che probabilmente sarà usata da quello che sarà poi l’unico broadcaster.

Abbiamo poi broadcast\_init, che inizializza la variabile global\_broadcast. Questa imposta broadcaster\_registered a false, poiché all’inizio non ci sono broadcaster, e poi ci sono dei tag “soluzione”, e quindi dovremo inizializzare anche i campi che aggiungeremo alla struttura.

Iniziamo ora a scrivere le primitive, e poi vediamo che aggiunte fare dentro i tag “soluzione” di volta in volta. Partiamo dalla c\_reg:



Innanzitutto, recuperiamoci global\_broadcast, assegnando il suo indirizzo ad un puntatore a broadcast\_t di nome *b* (prendiamo l’indirizzo, così possiamo eventualmente modificare i suoi campi, facendo b->… = … ). Servirà per dopo.

Poiché queste primitive vengono chiamate dall’utente, come prima cosa controlliamo che l’input sia valido.

Innanzitutto, è un errore se role non specifica né un broadcaster, né un listener. In caso di errore, la traccia dice di abortire, e quindi facciamo c\_abort\_p() e return.

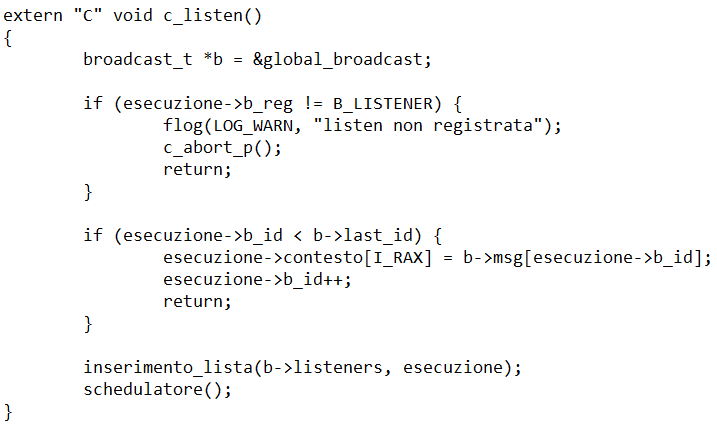
Dice poi che è un errore se il processo che ha chiamato la primitiva (che è il processo il cui des\_proc è puntato dal puntatore *esecuzione*) era già registrato (per lo stesso o un altro ruolo). Allora, se esecuzione->b\_reg è diverso da B\_NONE, significa che il processo è già registrato, e quindi c\_abort\_p() e return.

Dice poi che è un errore se c’è già un broadcaster e si tenta di registrarne un altro. Allora, se   
role == B\_BROADCASTER:

* se b->broadcaster\_registered == true, dunque c’è già un broadcaster, c\_abort\_p() e return;
* se invece b->broadcaster\_registered == false, quindi non c’è già un broadcaster, abbiamo superato tutti i controlli e quindi mettiamo b->broadcaster\_registered = true, perché stiamo registrando un nuovo broadcaster.

In ogni caso poi, superati tutti i controlli, mettiamo esecuzione->b\_reg = role, indicando quindi qual è adesso il ruolo del processo.

Facciamo ora la c\_listen:

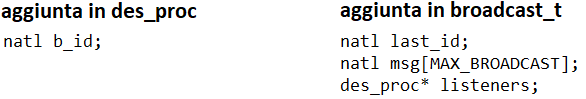


Recuperiamo global\_broadcast.

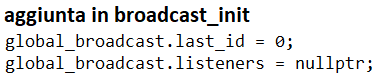
A questo punto, controllo input (indicando con input anche come l’utente chiama la primitiva, visto che sta primitiva non ha parametri d’ingresso). E’ un errore se il processo non è registrato come listener, quindi se esecuzione->b\_reg != B\_LISTENER, c\_abort\_p() e return.

La traccia dice che questa primitiva restituisce il prossimo messaggio di broadcast non ancora letto dal processo; se il processo li ha già letti tutti, si blocca in attesa del prossimo. Allora significa che dobbiamo fare delle aggiunte alle strutture dati viste prima:

* Nella struttura broadcast\_t, dobbiamo aggiungere un campo che è un array di msg di dimensione MAX\_BROADCAST. La traccia dice che i messaggi sono natl, quindi il campo da aggiungere è   
  “natl msg[MAX\_BROADCAST]”;
* Serve un campo all’interno della struttura broadcast\_t per indicare qual è il prossimo id dell’array destinato a contenere un messaggio. Sia allora “natl last\_id”;
* Serve implementare una lista di des\_proc relativi ai processi che sono in attesa di leggere messaggi, dunque aggiungiamo un campo “des\_proc\* listeners”;
* Serve un campo all’interno della struttura des\_proc per indicare qual è l’id del prossimo messaggio da leggere. Aggiungiamo allora un campo “natl b\_id”.



A questo punto dobbiamo andare anche in broadcast\_init e inizializzare i campi appena aggiunti, in particolare last\_id e listeners (l’array no, perché tanto lo scorreremo solo basandoci su last\_id).



* Last\_id a 0, poiché il primo id dell’array destinato a contenere un messaggio è proprio il primo elemento dell’array, di indice 0;
* Listeners a nullptr, perché all’inizio la lista di processi in attesa di leggere messaggi è vuota.

Tornando alla c\_listen, se il processo ha ancora da leggere messaggi, quindi esecuzione->b\_id < b->last\_id, allora bisogna restituirlo. **NB: se la primitiva presenta in Assembly salva/carica\_stato, per restituire una cosa non si fa “*return cosa*”. Ricordiamo infatti che “*return cosa*” mette *cosa* dentro RAX, ma poi c’è la carica\_stato, che sovrascrive tutti i registri, compreso RAX. Allora, per restituire una cosa, la si mette in   
esecuzione->contesto[I\_RAX], in modo che dopo la carica\_stato finisca in RAX proprio la *cosa* da restituire, e quindi il processo chiamante si trova effettivamente in RAX la roba restituita.**

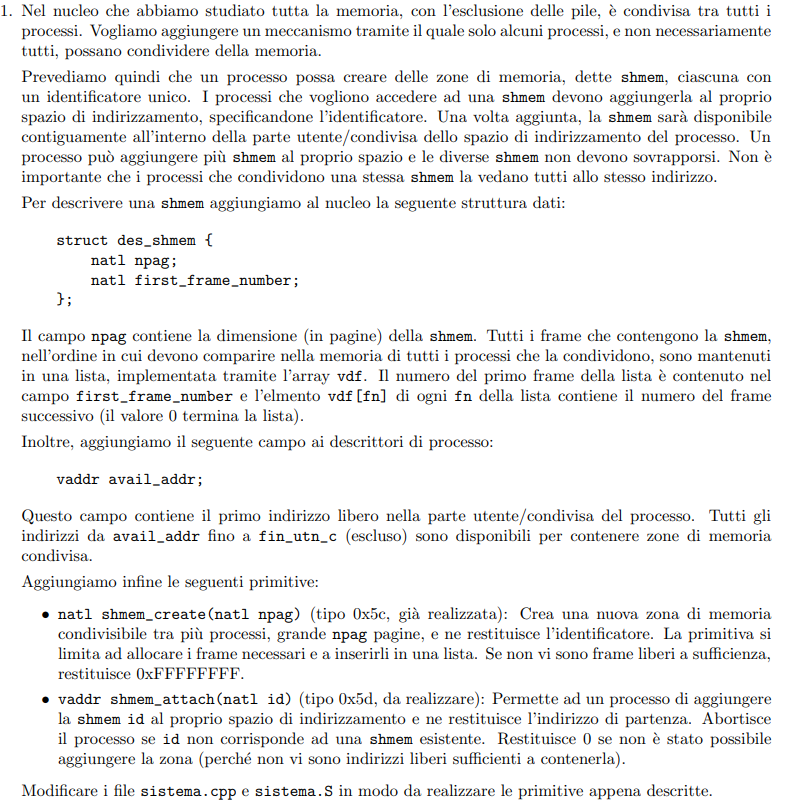
**Questo è il motivo per cui come tipo di ritorno si mette *void*: non deve ritornare nulla facendo “return …”.** Allora facciamo “esecuzione->contesto[I\_RAX] = b->msg[esecuzione->b\_id]”, quindi leggiamo il primo messaggio da leggere, e poi facciamo “esecuzione->b\_id++”, visto che ora bisogna leggere il prox messaggio. Infine facciamo “return” (non te ne dimenticare!), per tornare al chiamante, e quindi alla routine in Assembly.

Se invece il processo non ha da leggere messaggi, va messo in attesa. Allora mettiamo il processo il cui des\_proc è puntato da *esecuzione* dentro la lista b->listeners, e poi chiamiamo schedulatore(), così da schedulare un nuovo processo, che andrà in esecuzione dopo carica\_stato e IRETQ.

La c\_broadcast non la vediamo.

**ESERCITAZIONI MEMORIA VIRTUALE**

**PROVA D’ESAME 02/07/15**

****

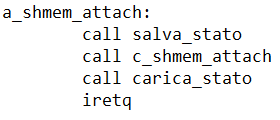
**SOLUZIONE**

**Partiamo dal modificare sistema.s.** Innanzitutto, vedendo i tag “soluzione”, i primi compaiono dove bisogna caricare il gate relativo al tipo di interruzione associato alla routine *a\_nomeprimitiva,* e lo facciamo:



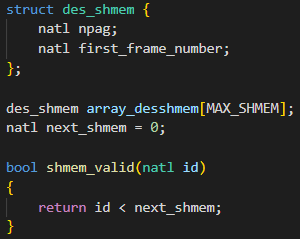
“LIV\_UTENTE” perché devono essere chiamate dall’utente.

Andando a vedere dove sono i prossimi tag “soluzione”, questi compaiono dove bisogna scrivere la parte Assembly della primitiva. Basta copiare la struttura delle altre routine pari pari mettendo al posto di c\_*primitiva* il nome della parte C++ della primitiva che dobbiamo implementare:



**Modifichiamo ora sistema.cpp.**

Andiamo a vedere quali aggiunte sono state fatte al file sistema.cpp:



Per descrivere una shmem si è aggiunta la struct des\_shmem. Il campo *npag* contiene la dimensione (in pagine) della shmem, mentre first\_frame\_number contiene il numero del primo frame della shmem.

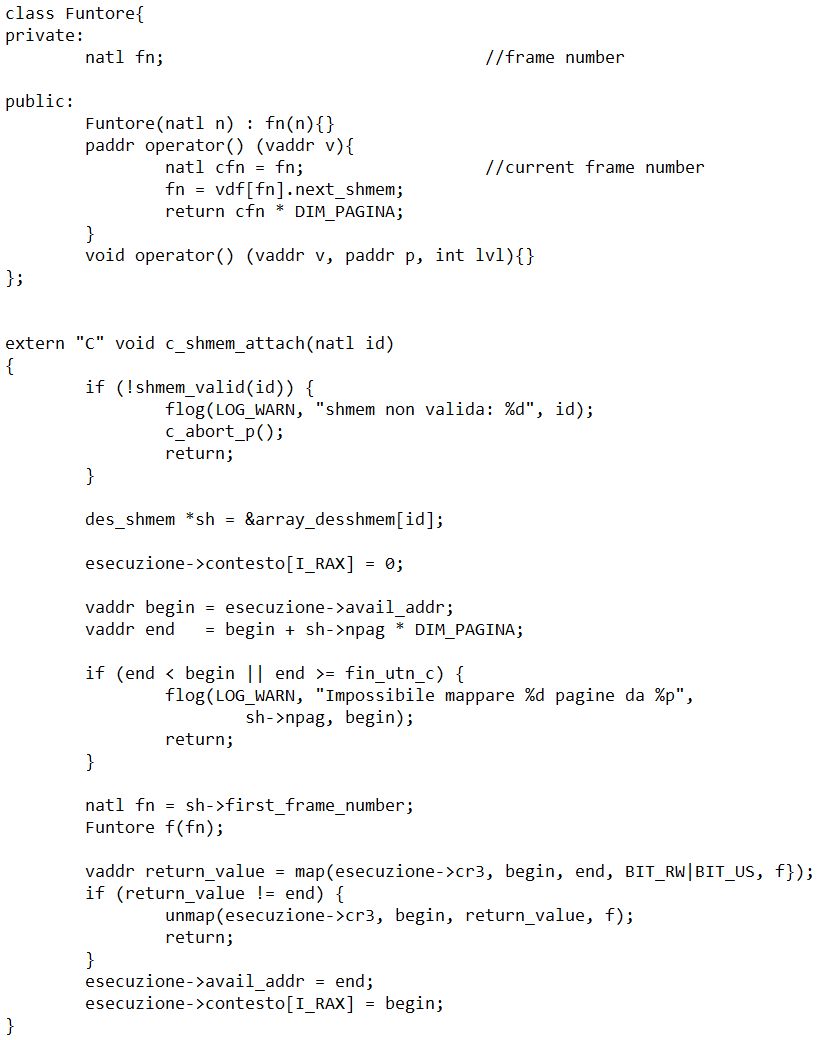
Tutti i frame che contengono la shmem, nell’ordine in cui devono comparire nella memoria di tutti i processi che la condividono, sono mantenuti in una lista, implementata tramite l’array *vdf*, che è l’array di des\_frame.

In particolare, nel des\_frame è stato aggiunto un campo *next\_shmem*, che contiene il numero del frame successivo della shmem. In questo modo, facendo vdf[first\_frame\_number] ottengo il descrittore del primo frame della shmem; se poi voglio recuperare il descrittore del frame successivo della shmem, basta fare vdf[next\_shmem], e così di volta in volta recupero i des\_frame successivi, e il valore 0 indica che la lista è terminata termina la lista (ricordiamo poi che l’indice i-esimo dell’array vdf[] contiene il des\_frame   
dell’i-esimo frame, quindi questa deferenziazione funziona).

Si ha anche un array di des\_shmem, e l’elemento *i*-esimo dell’array contiene il des\_shmem relativo alla shmem di identificatore *i.* C’è poi anche una variabile next\_shmem (da non confondere con il campo next\_shmem all’interno del des\_frame) che contiene qual è il prossimo indice dell’array (e quindi il prossimo identificatore) da riservare ad un des\_shmem, e quindi sono id validi tutti quelli che vanno da 0 a next\_shmem escluso.

Nel des\_proc di ogni processo poi è stato aggiunto un campo avail\_addr, che contiene il valore del primo indirizzo virtuale libero nella parte utente/condivisa del processo. Tutti gli indirizzi da avail\_addr fino a fin\_utn\_c escluso sono disponibili per mapparci delle shmem.

Vedendo ora dove sono i tag “soluzione”, questi si trovano dove bisogna implementare la parte C++ delle primitive. Facciamo ora la c\_shmem\_attach:



Ricordiamo anche qua come tipo di ritorno “void”, poiché la primitiva presenta in Assembly “salva/carica\_stato”, e quindi per ritornare qualcosa non si fa “return *qualcosa”,* ma metteremo dentro esecuzione->contesto[I\_RAX].

Come prima cosa, controlliamo l’input. Vediamo se l’id è valido e, se non lo è, c\_abort\_p() e return.

Se è valido, recuperiamo il des\_shmem relativo allo shmem che ha identificatore == *id,* salvando il suo indirizzo in una variabile *sh* puntatore a des\_shmem (prendiamo l’indirizzo, così se dobbiamo fare modifiche, facendo *sh*->… = … modifichiamo proprio il des\_shmem all’interno dell’array).

La primitiva restituisce 0 se non è stato possibile aggiungere la zona (perché non vi sono indirizzi liberi sufficienti a contenerla). Allora, per semplicità, a prescindere mettiamo in esecuzione->contesto[I\_RAX] il valore 0. Se poi tutto va a buon fine, metteremo poi l’indirizzo virtuale di partenza della shmem; altrimenti lasciamo 0.

Adesso, dobbiamo mappare la shmem nello spazio di indirizzamento virtuale del processo che prima era in esecuzione.L’indirizzo virtuale iniziale su cui mappare la shmem è *start* = esecuzione->avail\_addr. La traccia dice che questa shmem va mappata su pagine contigue, per cui il primo indirizzo virtuale su cui non verrà mappata la shmem è *end* = start + sh->npag \* DIM\_PAGINA. Si hanno allora due controlli da fare:

* Bisogna assicurarsi che *end* non sia >fin\_utn\_c, e quindi significherebbe che non vi sono indirizzi virtuali liberi sufficienti a contenere la shmem;
* Il controllo sopra non basta, perché **bisogna anche controllare che non si faccia wrap-around**: può succedere che la shmem sia talmente grande che facendo end = start + sh->npag \* DIM\_PAGINA non solo si supera fin\_utn\_c, ma si supera anche il valore 264 – 1. Vaddr è un natq, cioè un numero a 64 bit, come sono effettivamente gli indirizzi virtuali, e allora succede che arrivati a 264 – 1 si riparte da 0. Bisogna allora controllare anche che end non sia minore di start.

Se si verifica uno di questi casi, la traccia dice di restituire 0. 0 sta già in esecuzione->contesto[I\_RAX], allora basta fare return, così da tornare al chiamante (e cioè alla parte Assembly della primitiva).

A questo punto, fatti tutti i controlli, si può mappare la shmem nell’intervallo [start, end), **e per mappare frame fisici su tutte le pagine che fanno parte di un intervallo di indirizzi virtuali si usa la funzione map.**

In particolare, come parametri vanno passati:

* Come indirizzo fisico della tabella di livello 4 del trie su cui si vuole creare la traduzione, passiamo l’indirizzo della tabella di livello 4 del processo che ha invocato la primitiva, quindi esecuzione->cr3;
* Come *begin*, passiamo begin;
* Come *end*, passiamo end (NB: ricordiamo che sia come *begin*, che come *end*, va passato un indirizzo allineato alla pagina. Andando a vedere i tag esame, si vede che in fase di inizializzazione ogni avail\_addr è ini\_utn\_c + dim\_region(MAX\_LIV - 1)), dunque avail\_addr è allineato alla pagina. Allora anche begin ed end saranno sempre allineati alla pagina, visto che mappiamo sempre e solo delle pagine);
* Come bit da settare sui descrittori di pagina e delle regioni che si occupano di tradurre le pagine, mettiamo “BIT R/W | BIT U/S” (l’intervallo di indirizzi virtuali deve essere accessibile da livello utente, e permettiamo che sia scrivibile).
* Come parametro getpaddr, un qualcosa tale che, dato l’indirizzo virtuale *v* dell’inizio di una pagina che fa parte dell’intervallo [begin, end)*,* facendo *getpaddr(v)* deve restituire l’indirizzo fisico dell’inizio del frame da far corrispondere a *v.*

**In particolare, come getpaddrpassiamo un oggetto classe di un funtore, e cioè di una classe di cui è stato ridefinito l’operatore parentesi con un solo parametro (visto che dobbiamo pare getpaddr(v)) e che restituisce un paddr (ovvero l’indirizzo fisico del frame da far corrispondere a v).**

**Creiamo allora la classe funtore, definendo il costruttore, ridefinendo l’operatore () a un solo parametro che deve restituire un paddr, e creiamo un oggetto classe di tale classe.** Nel nostro caso, la facciamo così: dato un frame number *fn*, l’indirizzo fisico dell’inizio del frame è *fn \** DIM\_PAGINA. Allora aggiungo alla classe un campo dati *fn* che inizialmente conterrà il first frame number, e poi nel corpo dell’operator() mi salvo il current frame number in *cfn*, faccio avanzare *fn* al frame number successivo della shmem e poi restituisco   
fn \* DIM\_PAGINA. In questo modo, quando getpaddr(v) verrà chiamato pagina per pagina, restituirà correttamente l’indirizzo fisico del frame da mappare sulla pagina, e questo per ogni frame della shmem. Così, i frame della shmem verranno mappati consecutivamente nell’intervallo [begin, end).

**La funzione map restituisce poi il primo indirizzo virtuale su cui la map non ha mappato nulla. Si fa allora un ulteriore controllo: salvandoci il valore restituito da map in una variabile *return\_value,* se   
return\_value != end, allora significa che la mappatura non è andata a buon fine** (probabilmente in memoria non c’era spazio per allocare nuove tabelle del trie per creare la traduzione). **Allora quello che si fa è unmappare ciò che è stato mappato finora, usando la funzione unmap.**

Come parametri alla unmap vanno passati:

* Come indirizzo fisico della tabella di livello 4 del trie da cui vanno eliminate le traduzioni, l’indirizzo su cui prima le abbiamo create e che vogliamo togliere, quindi esecuzione->cr3;
* Come *begin*, passiamo begin;
* Come *end*, passiamo *return\_value* (anche in questo caso, return\_value sarà allineato alla pagina, visto che la map mappa pagine, e quindi return\_value, che è il primo indirizzo virtuale su cui la map non ha mappato nulla, sarà il primo indirizzo di una pagina);
* Come parametro putpaddr, un qualcosa tale che, dato l’indirizzo virtuale dell’inizio di una pagina *v*, l’indirizzo fisico *p* dell’inizio del frame che è mappato su tale pagina e da cui si vuole togliere tale mappatura e il livello della pagina *lvl*,facendo *putpaddr(v, p, lvl),* decide cosa fare sul frame stesso + altre cose.

**Anche in questo caso, come getpaddrpassiamo lo stesso oggetto classe della classe Funtore, ridefinendo però l’operatore parentesi nel caso in cui si hanno 3 parametri (visto che dobbiamo fare putpaddr(v, p, lvl) e che ha tipo di ritorno void (visto che non deve restituire nulla).**

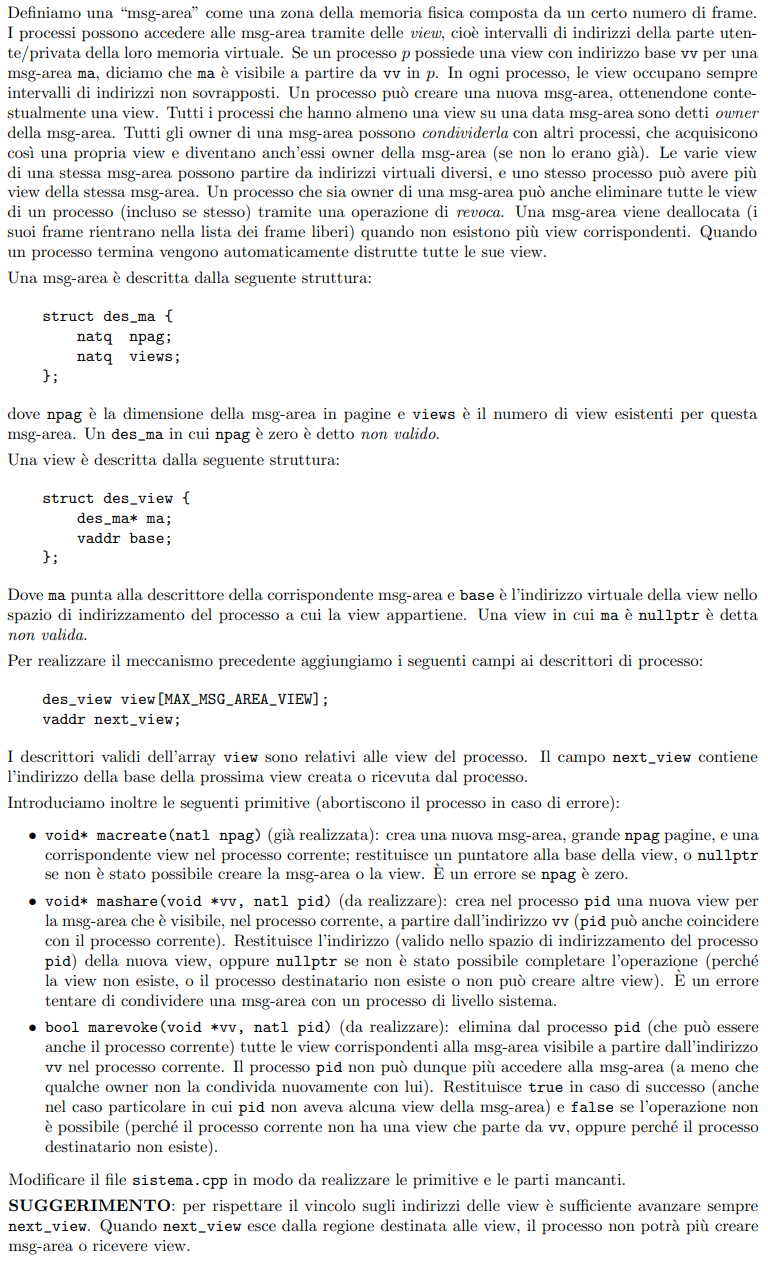
Ragioniamo su come dobbiamo ridefinirlo: sul frame fisico non dobbiamo fare nulla, perché dobbiamo solo togliere la mappatura dallo spazio di indirizzamento virtuale. Dobbiamo fare altre cose? Tra le possibili “altre cose”, ci sarebbe invalidare ogni volta l’entrata del TLB che contiene la traduzione della pagina di cui fa parte l’indirizzo virtuale *v*, altrimenti se il processo provasse ad accedere alla pagina, anche se abbiamo fatto l’unmap, poiché la copia della vecchia traduzione della pagina è ancora salvata nel TLB, il processo accederebbe comunque al frame che prima era mappato su tale pagina (ricordiamo che la MMU, prima di andare in memoria a consultare le tabelle per recuperare la traduzione di una pagina, guarda sempre prima il TLB. **La unmap non fa nulla sul TLB**, e quindi se nel TLB c’era memorizzata la traduzione precedente della pagina, anche dopo che una pagina viene unmappata, nel TLB continua ad esserci questa traduzione. Quindi il processo, accedendo ad una pagina unmappata, non provoca un’eccezione ha il page fault come dovrebbe esserci, ma accede invece al frame corrispondente precedente).

Ma serve farlo? Non viene mai fatto l’accesso a queste pagine, quindi nel TLB non abbiamo le traduzioni di queste pagine. Allora il corpo dell’operator() nel caso in cui si hanno tre parametri lo facciamo vuoto, visto che non deve fare nulla. In questo modo, quando putpaddr(v, p, lvl) verrà chiamato pagina per pagina, non farà nulla.

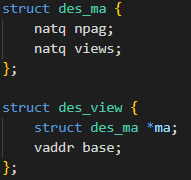
Se invece la map ha fatto tutto bene, bisogna aggiornare avail\_addr mettendo il valore *end* e restituire l’indirizzo virtuale di partenza della shmem, quindi mettiamo in esecuzione->contesto[I\_RAX] il valore *begin.*

La continuazione di questa è 23-07-15, ma la soluzione è un modo che non ho mai visto e ho saltato.

**PROVA D’ESAME 15/09/21**

****

**Modifichiamo sistema.cpp, visto che la traccia ci dice che sistema.s non va modificato, quindi significa che è già tutto pronto.** Andiamo a vedere quali aggiunte sono state fatte al file sistema.cpp:

****

Una msg-area è descritta dalla struttura des\_ma. *npag* è la dimensione della msg-area in pagine e *views* e il numero di view esistenti per questa msg-area. Un des\_ma in cui npag è zero è detto non valido. Abbiamo anche un array di des\_ma, chiamato *array\_des\_ma.*

Una view è descritta dalla struttura des\_view. *ma* punta al descrittore della corrispondente msg-area e *base* è l’indirizzo virtuale della view nello spazio di indirizzamento del processo a cui la view appartiene. Una view in cui *ma* è nullptr è detta non valida.

Si aggiungono poi i seguenti campi ai descrittori di processo:



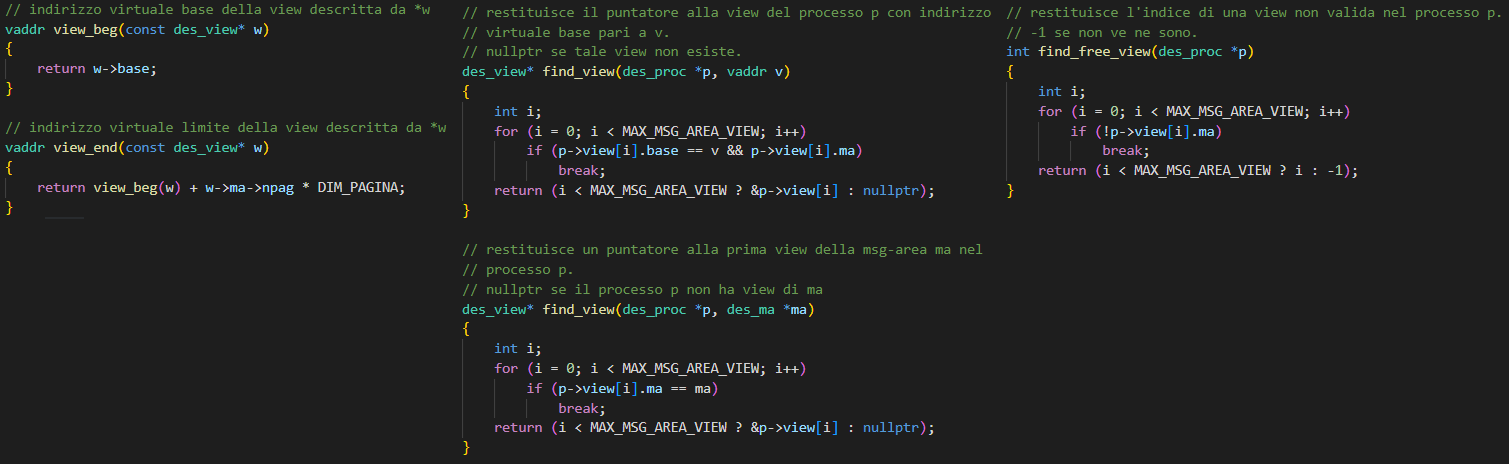
I descrittori validi dell’array *view* (che hanno *ma* != nullptr) sono relativi alle view del processo. Si vede quindi che un processo può avere massimo MAX\_MSG\_AREA\_VIEW view.

Il campo *next\_view* contiene il primo indirizzo libero in cui si può avere una view.

Abbiamo poi queste due costanti:

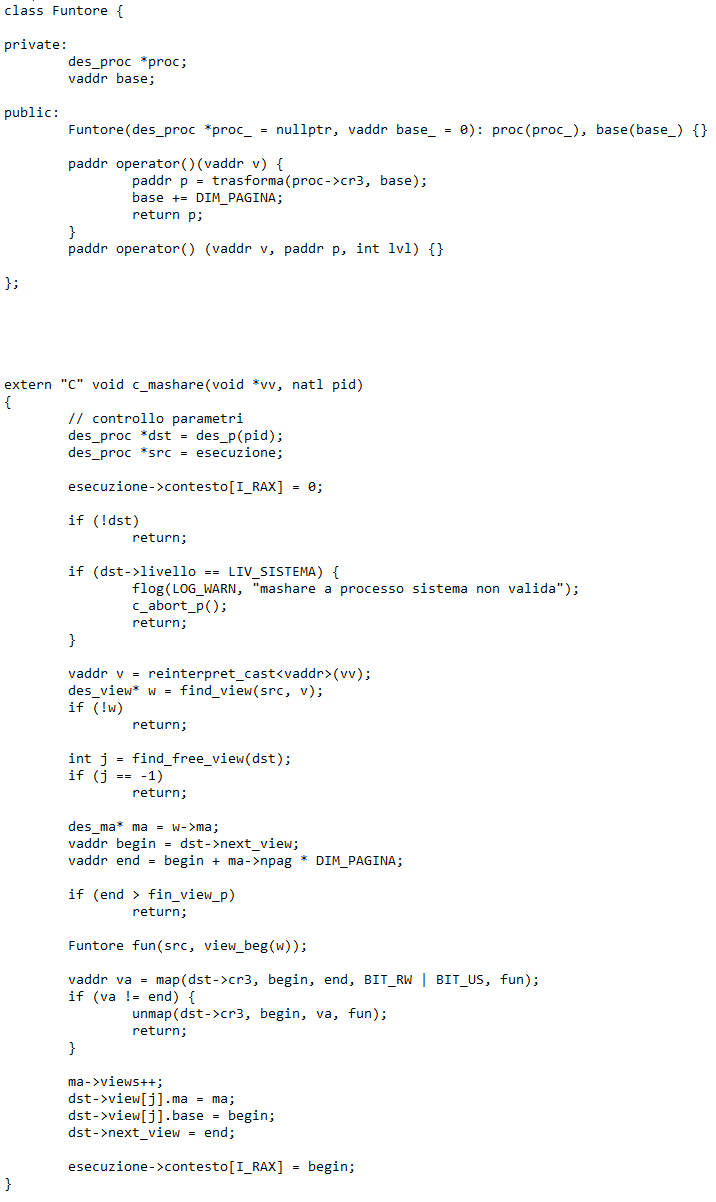


In particolare, fin\_view\_p è il primo indirizzo virtuale in cui non si può avere una view **(quindi è importante leggere come è stato modificato sistema.cpp, sennò questa cosa non l’avresti mai vista).**

Abbiamo poi delle funzioni di utilità:

Vedendo ora dove sono i tag “soluzione”, questi si trovano dove bisogna implementare la parte C++ delle primitive e anche nella distruggi\_processo(), in cui dovremo distruggere le view che si sono create.

Facciamo ora la c\_mashare:



Come prima cosa, controlliamo l’input. Serve recuperare il des\_proc sia del processo che ha invocato la primitiva (visto che dobbiamo condividere una sua view, e i des\_view si trovano nell’array del suo des\_proc), sia del processo di id == pid, su cui dobbiamo condividere la view. Il des\_proc del processo che ha invocato la primitiva è puntato da *esecuzione*, e quindi assegniamo il suo indirizzo ad un puntatore a des\_proc *src.* Per recuperare il des\_proc del processo su cui si vuole condividere la view, **si può usare la funzione des\_p che, dato l’id del processo, restituisce il des\_proc di tale processo se esiste, nullptr se non esiste o fpanic se pid > MAX\_PROC, quindi l’utente passa un input sbagliato.** Allora assegniamo il risultato restituito da des\_p(pid) ad un puntatore a des\_proc *dst.*

La primitiva restituisce l’indirizzo virtuale (valido nello spazio di indirizzamento del processo *pid*) della nuova view, oppure nullptr se non è stato possibile completare l’operazione. Allora direttamente mettiamo in   
esecuzione->contesto[I\_RAX] il valore 0, così se è stato possibile completare l’operazione mettiamo il valore corretto, altrimenti rimane 0 (ricorda di che si mette in esecuzione->contesto[I\_RAX], visto che la primitiva presenta salva/carica\_stato. Per lo stesso motivo, il tipo di ritorno della parte C++ è void, visto che non ritorna nulla facendo “return …”).

Continuiamo con i controlli. Se il processo non esiste (dst == nullptr), si restituisce 0, quindi basta tornare al chiamante (0 sta già in esecuzione->contesto[I\_RAX], e ricorda sempre di tornare al chiamante, cioè alla parte Assembly della primitiva).

Se si sta tendando di condividere una msg-area con un processo di livello sistema   
(dst->livello == LIV\_SISTEMA), è un errore, e in caso di errore la traccia mi dice che si fa c\_abort\_p() e return.

Ora dobbiamo recuperare la view da condividere. Questa view è del processo sorgente (e la stiamo condividendo al processo destinatario). Per recuperarla, usiamo la prima funzione *find\_view*, che restituisce il puntatore al des\_view del processo p con indirizzo virtuale base pari a v, o nullptr se tale view non esiste. Allora convertiamo vv in un vaddr, assegnando il valore a v (sennò il compilatore si lamenta se proviamo a chiamare la funzione passando vv), e poi chiamiamo la funzione find\_view(src, v), assegnando il valore ad un puntatore a des\_view che chiamiamo w. Se w == nullptr, allora la view non esiste, si restituisce 0 e si ritorna al chiamante. Altrimenti, in w abbiamo il puntatore al des\_view relativo alla view da condividere.

Superato questo controllo, la view esiste, e quindi creiamola anche nel processo destinatario. Innanzitutto, dobbiamo vedere se ne può effettivamente crearne un’altra, e cioè se non si è raggiunto il numero massimo di view. Questo si controlla vedendo c’è un elemento libero all’interno dell’array dst->view[], dove per “libero” si intende “non valido”, e cioè se ha *ma* == nullptr. Di questo se ne occupa la funzione *find\_free\_view* cui passiamo *dst,* che restituisce l’id del primo elemento libero dell’array se ce n’è uno, -1 altrimenti. Assegniamo allora questo valore ad una variabile j: se j == -1, allora non si può creare un’altra view, e quindi restituiamo 0 e torniamo al chiamante. Altrimenti, in j abbiamo l’indice dell’array dst->view[] da destinare al des\_view della view da creare.

Ora dobbiamo vedere quale intervallo di indirizzi virtuali deve occupare tale view, e quindi sapere quanto deve essere grande la msg-area da mappare. In w abbiamo il puntatore al descrittore della view da condividere, e tra i suoi campi ha anche il puntatore al descrittore della corrispondente msg\_area. Ci salviamo allora questo valore in un puntatore a des\_ma, che chiamiamo *ma.* Tra i campi di un des\_ma c’è anche la dimensione della corrispondente msg\_area espressa in numero di pagine. Allora l’indirizzo virtuale di partenza dello spazio di indirizzamento virtuale del processo destinatario in cui va mappata la view è   
*begin* = dst->next\_view; il primo indirizzo virtuale in cui non va mappato nulla è   
*end* = *begin* + ma->npag \* DIM\_PAGINA.

Ora dobbiamo vedere se c’è lo spazio: se end > fin\_view\_p, significa che non c’è spazio, e quindi si restituisce 0 e si ritorna al chiamante. Il problema del wrap-around non si pone, poiché ini\_view\_p corrisponde con l’inizio della sezione utente/privata; fin\_view\_p = ini\_view\_p + dim\_region(0) \* MSG\_AREA\_PAGES, dove MSG\_AREA\_PAGES == 10. Questo allora significa che una view può essere grande massimo 10 pagine, e quindi anche se begin fosse vicino a fin\_view\_p, sommandoci nel caso peggiore + 10\* DIM\_PAGINA non si fa wrap-around, visto che la sezione utente privata è grande 512 \* 128 GiB.

Superato questo controllo, andiamo ora a mappare la view sul processo destinazione, e usiamo la map, passando come parametri:

* Come indirizzo fisico della tabella di livello 4 del trie su cui si vuole creare la traduzione, passiamo l’indirizzo della tabella di livello 4 del processo destinatario, quindi dst->cr3;
* Come *begin*, passiamo begin;
* Come *end*, passiamo end (entrambi sono allineati alla pagina, visto che mappiamo pagine e basta);
* Come bit da settare sui descrittori di pagina e delle regioni che si occupano di tradurre le pagine, mettiamo “BIT R/W | BIT U/S” (l’intervallo di indirizzi virtuali deve essere accessibile da livello utente, e permettiamo anche che sia scrivibile).
* Come parametro getpaddr, un qualcosa tale che, dato l’indirizzo virtuale *v* dell’inizio di una pagina che fa parte dell’intervallo [begin, end)*,* facendo *getpaddr(v)* deve restituire l’indirizzo fisico dell’inizio del frame da far corrispondere a *v.*

Come parametro getpaddr passiamo un oggetto classe di un funtore, e quindi l’oggetto classe di una classe di cui è stato ridefinito l’operatore parentesi. Dichiariamo allora la classe Funtore. La situazione è questa: ad ogni pagina del processo destinatario vogliamo associare i frame della msg\_area. Sappiamo che questa msg\_area è stata già mappata nello spazio di indirizzamento virtuale del processo sorgente a partire dall’indirizzo virtuale view\_beg(w). Salviamoci allora questo valore in un campo dati *base* della classe. L’idea ora è questa: ridefiniamo l’operatore () ad un parametro, e vogliamo che la prima volta che verrà fatto getpaddr(v) per mappare la prima pagina del processo destinatario, *base* venga trasformato in indirizzo fisico secondo la traduzione del processo sorgente, e così si recupera l’indirizzo fisico dell’inizio del primo frame della msg\_area. Salviamoci questo indirizzo fisico in una variabile paddr *p*, ed è *p* il valore da restituire. Prima di restituirlo, però, sommiamo a *base* il valore DIM\_PAGINA, così che ora contenga l’indirizzo virtuale della pagina successiva della view del processo sorgente. In questo modo, quando verrà rifatto getpaddr(v), questa restituirà l’indirizzo fisico del secondo frame della msg\_area, e quindi sulla seconda pagina del processo destinatario verrà mappato questo secondo frame; quando verrà rifatto di nuovo getpaddr(v), restituirà l’indirizzo fisico del terzo frame della msg\_area, e quindi sulla terza pagina del processo destinatario verrà mappato questo terzo frame…

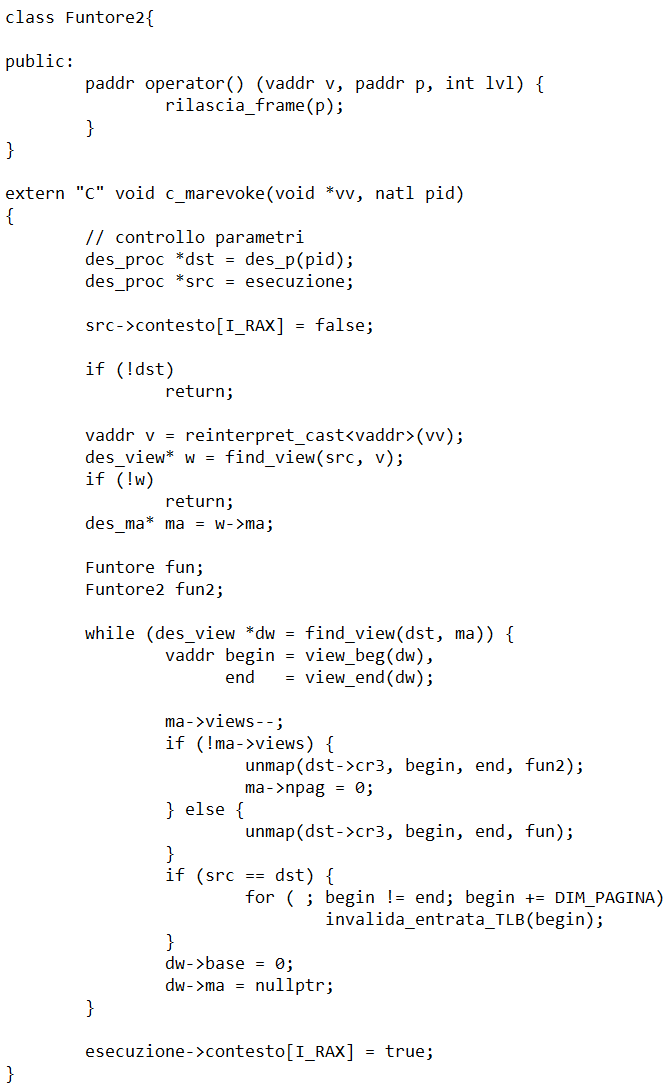
**Per trasformare un indirizzo virtuale in indirizzo fisico secondo la traduzione di un certo trie si usa la funzione trasforma.** In particolare, serve salvarci in un campo dati *proc* della classe il puntatore al des\_proc del processo sorgente, così da recuperare l’indirizzo fisico della tabella di livello 4 del trie del processo sorgente, e cioè il valore del campo cr3 (dato che src è una variabile locale, quindi non posso fare src->cr3 dentro la classe). A questo punto, facendo “paddr p = trasforma(proc->cr3, *base*)”, l’indirizzo virtuale *base* verrà trasformato in indirizzo fisico secondo la traduzione del trie del processo sorgente, e quindi si otterrà l’indirizzo fisico di un frame della msg\_area. Detto questo allora, definiamo il costruttore che inizializza i campi dati opportunamente (e lo facciamo con gli argomenti default, vedremo dopo perché).

La map restituisce l’ultimo indirizzo virtuale su cui non ha mappato nulla. Salviamoci questo valore in una variabile *va* e, se *va* != *end*, dunque la mappatura non è andata a buon fine, unmappiamo tutto ciò che è stato mappato, e quindi l’intervallo [begin, va), restituiamo 0 e torniamo al chiamante. Come parametro putpaddr() passiamo lo stesso oggetto classe *fun* del funtore, ridefinendo l’operatore parantesi a tre parametri. Sui frame fisici non dobbiamo fare nulla, perché dobbiamo solo togliere la mappatura dallo spazio di indirizzamento virtuale, così come non dobbiamo invalidare il TLB, perché non viene mai fatto l’accesso a queste pagine, quindi nel TLB non abbiamo le traduzioni di queste pagine. Allora il corpo dell’operator() a tre parametri lo facciamo vuoto, visto che non deve fare nulla. In questo modo, quando putpaddr(v, p, lvl) verrà chiamato pagina per pagina, non farà nulla.

Arrivati qui, significa che la mappatura è andata a buon fine. A questo punto dobbiamo aggiustare un po’ di cose nelle varie strutture dati:

* Restituiamo l’indirizzo virtuale della nuova view (*begin);*
* Incrementiamo il campo ma->views, visto che ora si ha una view in più per tale msg\_area;
* Dobbiamo aggiornare il des\_view corrispondente alla view appena creata. L’indice di questa des\_view è *j,* e l’array è contenuto nel des\_proc del processo *dst,* quindi basta fare   
  dst->view[j].base = begin e dst->view[j].ma = ma;
* Dobbiamo aggiornare il campo dst->next\_view, mettendo ora il valore *end,* visto che è questo ora il primo indirizzo in cui si può avere una view.

Facciamo ora la c\_marevoke:



E’ simile alla primitiva di prima, solo che ora invece di condividere la view, si eliminano dal processo di identificatore *pid* tutte le view corrispondenti alla msg-area visibile a partire dall’indirizzo vv nel processo corrente.

Come prima cosa, controlliamo l’input, e prima recuperiamo il des\_proc sia del processo che ha invocato la primitiva (visto che dobbiamo recuperare la sua view, e i des\_view si trovano nell’array del suo des\_proc), sia del processo di id == pid da cui dobbiamo togliere delle view, e il modo è lo stesso visto prima.

Anche in questo caso, mettiamo direttamente in esecuzione->contesto[I\_RAX] il valore false, così in caso di successo mettiamo true, altrimenti rimane false (anche qui, ricorda che si mette in esecuzione>contesto[I\_RAX], visto che la primitiva presenta salva/carica\_stato. Per lo stesso motivo, il tipo di ritorno della parte C++ è void, visto che non ritorna nulla facendo “return …”).

Se il processo non esiste (dst == nullptr), si restituisce false, quindi basta tornare al chiamante (false sta già in esecuzione->contesto[I\_RAX], e ricorda sempre di tornare al chiamante, cioè alla parte Assembly della primitiva).

Ora dobbiamo recuperare la view. Questa view è del processo sorgente, e per recuperarla usiamo la prima funzione *find\_view*, che restituisce il puntatore al descrittore della view. Allora convertiamo vv in un vaddr, assegnando il valore a v (sennò il compilatore si lamenta se proviamo a chiamare la funzione passando vv), e poi chiamiamo la funzione find\_view(src, v), assegnando il valore ad un puntatore a des\_view che chiamiamo w. Se w == nullptr, allora la view non esiste, si restituisce false e si ritorna al chiamante. Altrimenti, in w abbiamo il puntatore alla des\_view.

Superato questo controllo, la view esiste. Recuperiamo il puntatore al descrittore di msg\_area corrispondente. Ora dobbiamo togliere dal processo destinatario tutte le view corrispondenti a tale msg-area. *Tutte* le view, perché nel processo destinatario possiamo avere più view della stessa msg\_area. Bisogna allora scorrere tutto l’array view del des\_proc del processo destinatario e, per ogni volta che si trova una des\_view corrispondente a tale msg\_area, dobbiamo eliminare la view. Usiamo allora la seconda funzione *find\_view,* che restituisce il puntatore alla des\_view relativa alla prima view corrispondente alla msg\_area *ma* del processo *p*, o nullptr se tale view non esiste. Facciamo allora il seguente while:

while(des\_view \*dw = find\_view(dst, ma){

Ad ogni iterazione mi occuperò di eliminare la view, e così in dw ogni volta avrò il des\_view di una view diversa corrispondente alla msg\_area. Quando non ci saranno più view corrispondenti alla msg\_area, in dw finirà nullptr, e cioè 0. La condizione del while allora è false, e quindi uscirò dal while.

Ora dobbiamo unmappare la view. In dw abbiamo il puntatore des\_view della view da unmappare, e il valore dell’indirizzo l’indirizzo iniziale e finale (escluso) della view si ricavano rispettivamente con view\_beg(dw) e view\_end(dw). Assegniamo questi valori a due variabili vaddr *begin* ed *end.* Decrementiamo poi il campo   
ma->views, visto che adesso si avrà una view in meno di tale msg\_area. Adesso si distinguono due casi:

* Primo caso: ma->views == 0. **La traccia mi dice che una msg-area viene deallocata (i suoi frame rientrano nella lista dei frame liberi) quando non esistono più view corrispondenti. Allora, nella unmap, come parametro putpaddr va passato un oggetto classe di un funtore tale che, ogni volta che per ogni pagina viene fatto putpaddr(v, p, lvl), dealloca anche il corrispondente frame fisico, e cioè il frame fisico della msg\_area. Creiamo allora una seconda classe funtore Funtore2 (visto che per la prima abbiamo già ridefinito l’operatore () a tre parametri), ridefiniamo l’operatore () a tre parametri (vaddr v, paddr p e int lvl) che, quando chiamato, fa rilascia\_frame(p).** Creiamo allora un oggetto classe *fun2* di Funtore2 e lo passiamo alla unmap come parametro getpaddr.

Dobbiamo poi aggiustare il des\_ma, visto che ora la msg-area non esiste, mettendo ma->npag = 0.

* Secondo caso: ma->views != 0. In questo caso, basta unmappare senza fare nulla sui frame corrispondenti. Per semplicità, allora, passiamo passare come putpaddr un oggetto classe della classe Funtore *fun,* che ha l’operatore () a tre parametri ridefinito che non fa nulla (*e posso dichiararlo facendo “Funtore fun;” senza passare nulla, poiché il costruttore ha gli argomenti default*).

Sorge ora una domanda: dobbiamo fare qualcosa sul TLB? Si hanno due situazioni, che possono valere per entrambi i casi visti sopra:

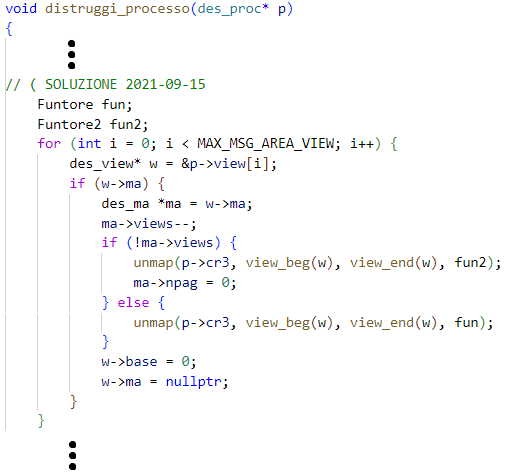
* src != dst: in questo caso, non si deve fare nulla sul TLB. Infatti, nel TLB abbiamo salvate le traduzioni delle pagine che riguardano il processo che precedentemente era in esecuzione, che è diverso dal processo a cui abbiamo tolto la mappatura a certe pagine. Dunque, quando lo stesso processo tornerà in esecuzione, il TLB non viene invalidato perché non si cambia processo, ma questo non è un problema, perché non abbiamo fatto modifiche al suo spazio di indirizzamento virtuale, e quindi non dobbiamo neanche fare modifiche al TLB.
* src == dst: in questo caso, dobbiamo invalidare le entrate del TLB che contenevano le traduzioni delle pagine su cui era mappata la msg\_area. Infatti, noi abbiamo tolto la mappatura da pagine dello spazio di indirizzamento virtuale del processo che prima era in esecuzione, dunque quando poi finita la primitiva tornerà in esecuzione lo stesso processo, il TLB non viene invalidato perché non si è cambiato processo, e questo è un problema, perché nel TLB continuano ad essere memorizzate le vecchie traduzioni delle pagine su cui ora invece non è più mappato nulla (ricorda che la unmap non fa nulla sul TLB). Allora, se il processo provasse ad accedere a queste pagine, anche se non c’è mappato nulla, poiché la vecchia traduzione di queste pagine è ancora salvata nel TLB, riuscirebbe comunque ad accedere ai precedenti frame corrispondenti alle pagine. Questo va impedito, e quindi vanno invalidate queste entrate del TLB.

Per fare questo allora, facciamo quel *for* partendo da begin e finché begin != end, si fa invalida\_entrata\_TLB(begin) e poi incrementiamo *begin* di DIM\_PAGINA, così da invalidare tutte le entrate del TLB che contengono le vecchie traduzioni di tutte le pagine su cui si è fatto unmap.

Sempre dentro il while poi, per ogni view che eliminiamo, dobbiamo aggiustare il corrispondente des\_view all’interno dell’array view. In particolare, visto che la view non esiste più, mettiamo   
dw->ma = nullptr (così da rendere il des->view libero) e dw->base = 0 (non so se quest’ultima cosa sia necessaria).

Infine, poiché si è fatto tutto con successo (anche nel caso particolare in cui *pid* non aveva alcuna view della msg-area), mettiamo in esecuzione->contesto[I\_RAX] il valore true.

Modifichiamo ora la distruggi\_processo, aggiungendo questo:



Dobbiamo eliminare tutte le view del processo da distruggere. Noi abbiamo il puntatore al des\_proc *p* del processo da distruggere, e questo ha un array p->view[] che contiene tutti i possibili des\_view relativi a tutte le view che il processo può avere. Quello che facciamo allora è scorrere questo array, assegnando di volta il volta il des\_view a cui si arriva ad un puntatore a des\_view *w.* Ogni volta che si trova un des\_view con   
w->ma != nullptr, significa che è un des\_view valido. Assegniamo allora il campo w->ma ad un puntatore a des\_ma *ma*, decrementiamo ma->views e, se questo è 0, dobbiamo anche deallocare la msg\_area, in maniera simile a quanto visto prima. L’unica differenza è che non vediamo se si deve invalidare il TLB, perché tanto il processo lo stiamo distruggendo, quindi non esisterà più. Ad ogni descrittore di view mettiamo poi   
w->base = 0 e w->ma = nullptr, visto che ora non c’è più la view.

**NB:** per ogni parte C++ creiamo un nuovo oggetto classe funtore, perché li creiamo come variabili locali, e quindi non sono visibili tra una parte C++ e l’altra. Magari poi creali con gli argomenti default, così nel caso in cui non ti servono i suoi membri dato, lo dichiari semplicemente senza passare argomenti.

**ESERCITAZIONI I/O**

**PROVA D’ESAME 15/06/16**

****

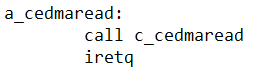
**SOLUZIONE**

**Partiamo dal modificare *io.s*.** Innanzitutto, vedendo i tag “soluzione”, i primi compaiono dove bisogna caricare il gate relativo al tipo di interruzione associato alla routine *a\_nomeprimitiva*. Questo viene fatto usando la macro **fill\_io\_gate,** che si usa allo stesso modo di come si usava carica\_gate:

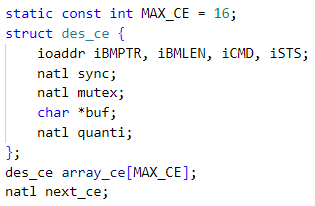


Anche in questo caso, IO\_TIPO\_CEREAD si ricava dal modulo utente.s

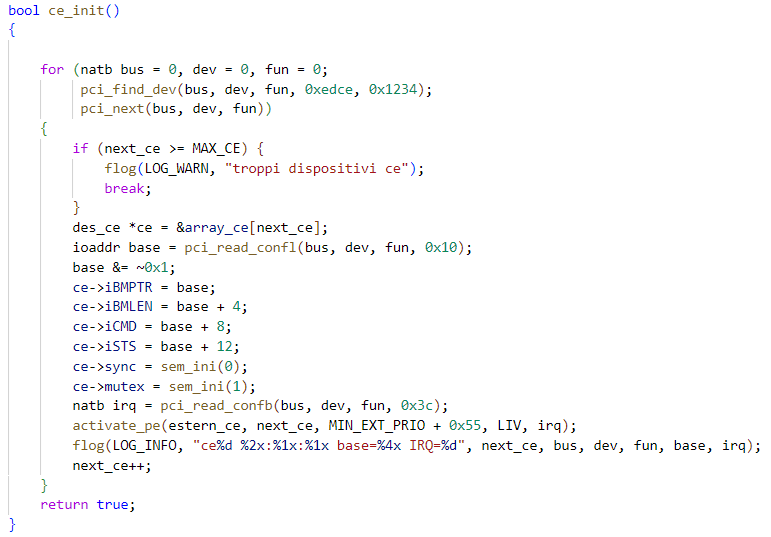
I prossimi tag “soluzione” si trovano dove bisogna scrivere la parte Assembly della primitiva**.** Senza sapere né leggere né scrivere, anche qui basta copiare la struttura delle altre routine pari pari mettendo al posto di c\_*primitiva* il nome della parte C++ della primitiva che dobbiamo implementare, ricordando che non va messo carica/salva\_stato per le primitive d’I/O:



**Modifichiamo ora *io.cpp*.** Andiamo a vedere quali aggiunte sono state fatte:

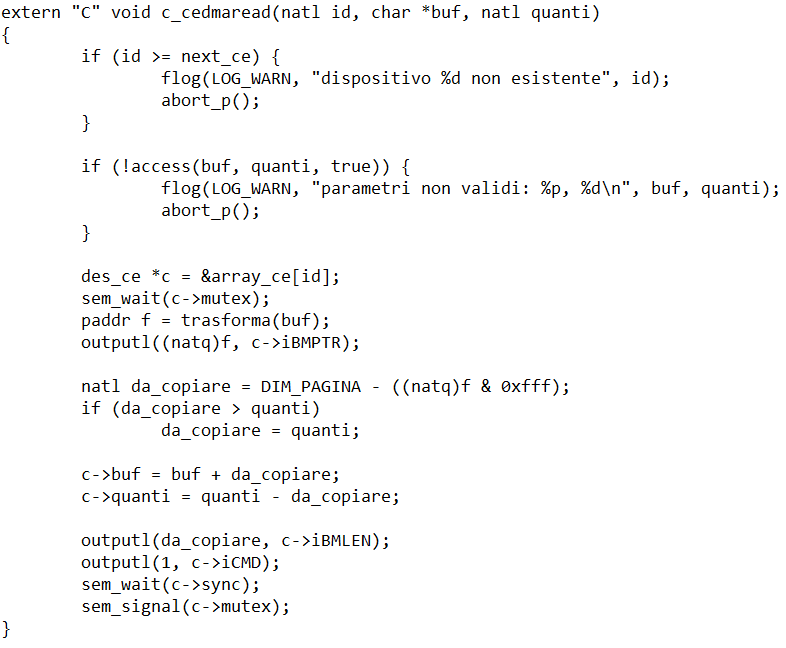


La struttura des\_ce descrive una periferica di tipo *ce* e contiene al suo interno gli indirizzi dei registri *BMPTR*, *BMLEN*, *CMD* e *STS*, l’indice di un semaforo inizializzato a zero (*sync*), l’indice di un semaforo inizializzato a 1 (*mutex*), il numero di byte che restano da trasferire (*quanti*) e l’indirizzo virtuale a cui vanno trasferiti (*buf*). Questi des\_ce sono contenuti in un *array\_ce* e la variabile *next\_ce* contiene qual è il prossimo indice dell’array da riservare ad un des\_ce, e contengono effettivamente des\_ce tutti gli elementi dell’array di indice che va da 0 a *next\_ce* escluso. Ogni periferica è identificata dall’indice del suo descrittore.



C’è poi la funzione **ce\_init** che è sempre già definita nei testi d’esame, e si occupa di inizializzare le periferiche ce. In particolare, ad ognuna assegna un id, riempie i campi “fissi” del des\_ce relativo (es. si occupa di inizializzare i campi del des\_ce relativi agli indirizzi cui sono montati i registri, visto che è ce\_init che si occupa assegnare ad ogni registro della funzione un indirizzo, inizializza i semafori e assegna ai campi relativi l’id del semaforo creato) e, soprattutto, **crea il** **processo esterno, che andrà in esecuzione ogni volta che il dispositivo manderà una richiesta di interruzione** (activate\_pe si occupa di caricare l’handler che poi andrà in esecuzione a seguito della richiesta di interruzione e manderà in esecuzione proprio il processo esterno, fa tutto activate\_pe)**.** Il nome della funzione eseguita dal processo esterno è *estern\_ce,* dunque dovremo creare poi una funzione *estern\_ce* all’interno di io.cpp.

Facciamo ora la c\_cedmaread:



Innanzitutto, si fa il controllo sull’input passato dall’utente. Come prima cosa, si vede se l’id passato dall’utente corrisponde effettivamente ad una periferica. Se no, **si fa solo abort\_p(),** e non c\_abort\_p() e return (visto che c\_abort\_p è del modulo sistema, non I/O. Non serve return, perché tanto poi con la abort\_p dopo carica\_stato e IRETQ automaticamente si passerà ad un altro processo.

Ora si fa un controllo sul buffer che occupa gli intervalli virtuali [buf, buf + quanti). In particolare, bisogna assicurarsi che:

* Sia un intervallo fatto da pagine che hanno una traduzione e che l’utente possa accederci (problema del Cavallo di Troia);
* Sia un intervallo che sia scrivibile (la periferica infatti dovrà scriverci, e non dobbiamo farle scrivere in una parte non scrivibile, perché potrebbe essere ad esempio la sezione text, e così si avrebbero inconsistenze.)
* Al momento non sappiamo se deve far parte della zona condivisa, dipende da come scriviamo il processo esterno (infatti è il dispositivo stesso ad occuparsi di fare l’operazione di I/O, lavorando con indirizzi fisici. Se poi il processo esterno dovrà maneggiare con gli indirizzi virtuali del buffer, allora dovrà per forza essere un buffer contenuto nella zona condivisa, altrimenti in caso contrario il processo esterno andrebbe ad accedere ad indirizzi virtuali della propria sezione utente/privata.

**Per fare questi controlli usiamo la primitiva access.** Se non sono rispettati, abort\_p().

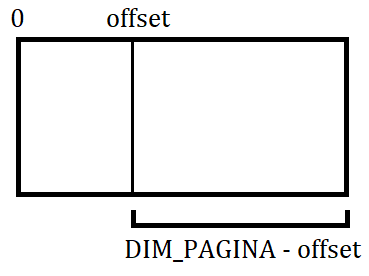
**Notare una cosa**: ora è in esecuzione il modulo I/O, ma il trie di traduzione non è cambiato, quindi correttamente stiamo facendo controlli sull’intervallo di indirizzi virtuali dello spazio di indirizzamento virtuale del processo che ha invocato la primitiva, e quindi che ha richiesto l’operazione di I/O.

**Fatto questo, si segue lo schema tipico di una primitiva di I/O:**

1. recuperiamo il descrittore di operazioni di I/O, assegnando l’indirizzo ad un puntatore a des\_ce *c* (così per modificarlo si può fare c->… = …);
2. Adesso, poiché la primitiva deve usare risorse condivise (deve modificare il des\_ce, che è di tutti i processi, e lavorare con il dispositivo), prendiamo la mutua esclusione.
3. Ora si avvia l’operazione di I/O. Dobbiamo scrivere sia nel des\_ce, sia nei registri della periferica. Partendo da BMPTR, qui va messo il puntatore al buffer di destinazione. La periferica lavora con gli indirizzi fisici (sta dopo la MMU), dunque non va passato buf, ma l’indirizzo fisico corrispondente a buf, secondo il trie del processo che ha invocato la primitiva (che è quindi il trie attualmente attivo). Si può usare allora la **primitiva *trasforma****,* che traduce un indirizzo virtuale in indirizzo fisico secondo il trie attualmente attivo. Il valore lo assegniamo ad una variabile *f*, e poi scriviamo questo valore dentro il registro BMPTR (usando outputl, “output” visto che si sta scrivendo ad un indirizzo dello spazio di I/O e “l” perché il registro ha dimensione 4 byte). **Ora verrebbe da fare “outputl(quanti, c->iBMLEN)”, ma dobbiamo stare attenti a non sforare la pagina. Infatti, in questo modo la periferica scriverebbe *quanti* byte consecutivi partendo dall’indirizzo fisico *f* ma non è detto che a pagine contigue dell’intervallo di indirizzi virtuali [buf, buf + quanti) corrispondano anche frame contigui dell’intervallo di indirizzi fisici [f, f + quanti). Si rischierebbe allora di scrivere in frame innocenti e non si scriverebbe invece nei frame corretti.**

**Allora *quanti*** **deve essere un numero tale fino ad arrivare alla fine della pagina (o alla fine del frame, tanto l’offset all’interno di una pagina == offset all’interno di un frame).**

**Per calcolare quanti byte ci sono partendo da *buf* fino alla fine della pagina, basta fare   
DIM\_PAGINA – offset di buf all’interno della pagina**, come si vede in questo disegno:



DIM\_PAGINA – offset di buf all’interno della pagina mi dice esattamente quanti byte ci sono da buf incluso fino alla fine della pagina. Vediamolo con degli esempi:

* se *offset* = 4095, DIM\_PAGINA – offset = 1, e infatti c’è solo un byte;
* se *offset* = 4094, DIM\_PAGINA – offset = 2, e infatti ci sono 2 byte.

e quindi:

Gli ultimi 12 bit dell’indirizzo fisico mi dà offset all’interno della pagina (in questo caso, all’interno del frame, ma i 2 offset sono uguali)

da\_copiare = DIM\_PAGINA – (f & 0xFFF)



Ora dobbiamo vedere se questo valore è >quanti, perché altrimenti la periferica sforerebbe il buffer e scriverebbe oltre. Allora, in questo caso, da\_copiare = quanti.

Nel caso in cui invece quanti > da\_copiare, e quindi il buffer è su più pagine, faremo in modo che il processo esterno si occupi di fare nuovi trasferimenti finché non verranno fatti tutti.

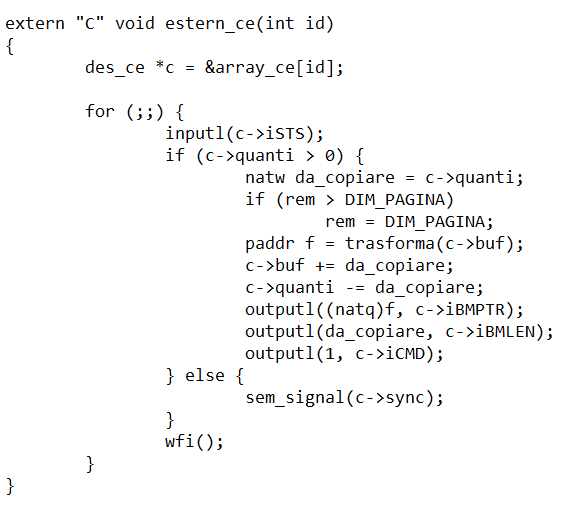
Riempiamo poi i campi del des\_ce. Per comodità per il processo esterno, mettiamo in c->buf il valore   
buf + da\_copiare, così che ora abbia il valore dell’indirizzo iniziale dell’eventuale pagina successiva in cui c’è il buffer, e c->quanti = quanti - da\_copiare, così il processo esterno sa quanti sono i byte da copiare rimasti.

Infine, mettiamo in BMLEN il valore “da\_copiare” e avviamo l’operazione di I/O, mettendo 1 in CMD.

1. A questo punto, aspettiamo che l’operazione di I/O si concluda. A questo punto, la periferica farà il trasferimento da sé, prendendo il controllo del bus.
2. Una volta che l’operazione di I/O si è conclusa, rilasciamo la mutua esclusione.

**Notare una cosa:** il modulo I/O ha le interruzioni abilitate, quindi si può interrompere la sua esecuzione e va in esecuzione altro. Quando tornerà in esecuzione il modulo I/O, verrà prima messo in *esecuzione* il des\_proc che ha invocato la primitiva, e poi con carica\_stato e IRETQ tornerà in esecuzione il modulo I/O e sempre con il trie del processo che ha invocato la primitiva. Dunque non si avrebbero possibili inconsistenze dovute al fatto che il trie di traduzione cambia mentre si esegue lo stesso codice (es. la usando la trasforma). Ecco perché è conveniente pensare alle primitive di I/O come se ancora fosse in esecuzione lo stesso processo che le ha invocate.

Facciamo ora il processo esterno:



Non si fanno controlli, in quanto il processo esterno viene messo in esecuzione da un handler, e cioè da qualcosa del modulo sistema, e ci fidiamo. Quell’id come parametro che viene passato al processo esterno è specificato nell’activate\_pe:

Immagine che contiene testo

Descrizione generata automaticamente

next\_ce era il primo indice libero all’epoca per un descrittore di CE, e quindi è l’identificatore della periferica. Quindi “id” è l’identificatore della periferica.

La struttura del processo esterno è sempre la stessa:

* Recuperiamo il descrittore di CE relativo al dispositivo;
* Si fa un for infinito dove si fanno varie elaborazioni e alla fine wfi().

Vediamo cosa fare dentro il for. Innanzitutto, possiamo leggere dal registro STS, così da rispondere alla richiesta di interruzione del dispositivo (e questo non crea problemi, tanto il dispositivo manderà nuove richieste di interruzioni solo dopo che noi gli abbiamo dato il comando di fare un trasferimento e lui ha finito).

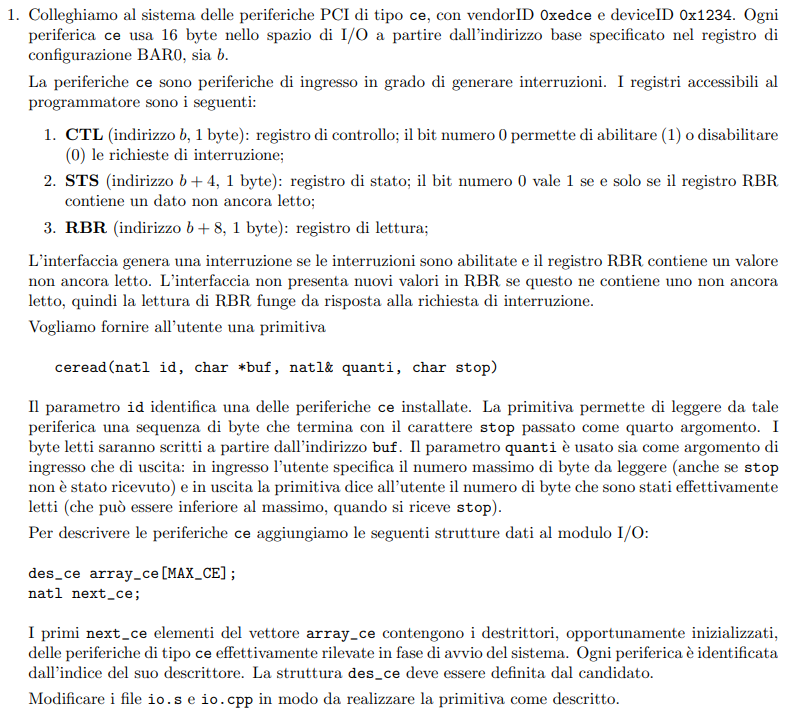
A questo punto, se c->quanti > 0, dunque bisogno fare ulteriori trasferimenti, mettiamo in una variabile da\_copiare il valore c->quanti. Se da\_copiare > DIM\_PAGINA, visto che si può copiare al massimo fino alla fine della pagina, e ora c->buf punta all’inizio di una pagina, si possono copiare al massimo 4096 byte, e quindi mettiamo in da\_copiare il valore DIM\_PAGINA. **Ora attenzione**: in BMPTR dobbiamo mettere l’indirizzo fisico del buffer. Noi siamo in un processo esterno, che ha un proprio trie, e se facciamo *f* = trasforma(buf), in *f* finirà la traduzione di *buf* in indirizzo fisico secondo il trie del processo esterno. Dunque, tornando all’access di prima, dobbiamo assicurarci che il buffer faccia parte della sezione utente/condivisa, visto che per ogni processo le pagine di tale sezione si traducono negli stessi frame, e quindi scrivendo in quei frame si scrive nella sezione utente/condivisa di ogni processo, compreso quello che ha invocato la primitiva. Altrimenti, se fosse stata utente/privata, si sarebbe scritto nella parte privata del processo esterno, e non in quella del processo che ha invocato la primitiva. Allora, se mettiamo *f* in BMPTR, metteremo proprio l’indirizzo del frame in cui si trova la parte del buffer interessata nel trasferimento.

Incrementiamo poi c->buf di *da\_copiare*, così che eventualmente punti alla pagina successiva in cui si trova il buffer; decrementiamo poi c->quanti di *da\_copiare*, così abbiamo il numero di byte che rimangono da copiare. A questo punto, mettiamo in BMPTR il valore di *f;* in BMLEN il valore di *da\_copiare* e in CMD il valore 1, così da avviare l’operazione.

Se invece c->quanti = 0, basta segnalare che è finita l’operazione di I/O.

Infine, in ogni caso, mettiamo wfi().

**PROVA D’ESAME 06/07/16**



**SOLUZIONE**

**Partiamo dal modificare *io.s*.** Innanzitutto, vedendo i tag “soluzione”, i primi compaiono dove bisogna caricare il gate relativo al tipo di interruzione associato alla routine *a\_nomeprimitiva*. Questo viene fatto usando la macro fill\_io\_gate**,** che si usa allo stesso modo di come si usava carica\_gate:



Anche in questo caso, IO\_TIPO\_CEREAD si ricava dal modulo utente.s

I prossimi tag “soluzione” si trovano dove bisogna scrivere la parte Assembly della primitiva**.** Senza sapere né leggere né scrivere, anche qui basta copiare la struttura delle altre routine pari pari mettendo al posto di c\_*primitiva* il nome della parte C++ della primitiva che dobbiamo implementare, ricordando che non va messo carica/salva\_stato per le primitive d’I/O:

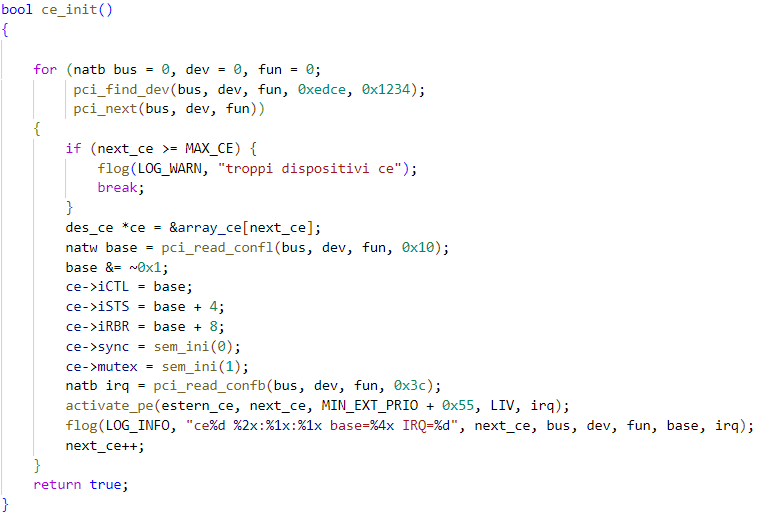
Immagine che contiene testo

Descrizione generata automaticamente

**Modifichiamo ora *io.cpp*.** Andiamo a vedere quali aggiunte sono state fatte:

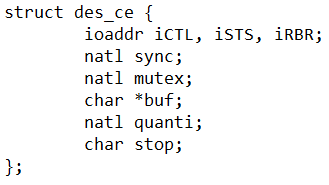


I primi next\_ce elementi del vettore array\_ce contengono i descrittori, opportunamente inizializzati, delle periferiche di tipo ce effettivamente rilevate in fase di avvio del sistema. Ogni periferica è identificata dall’indice del suo descrittore. La struttura des\_ce deve essere definita dal candidato. Come definirla? Andiamo a vedere come è fatta la ce\_init, che è stata aggiunta:



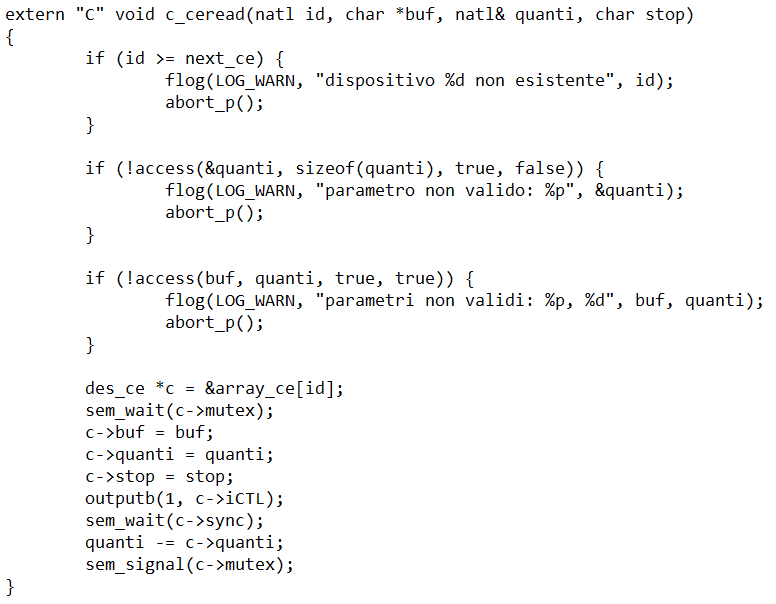
Vediamo che viene dichiarato un puntatore a des\_ce, e vengono inizializzati dei campi iCTL, iRBR, iSTS, iRBR, sync e mutex. Dunque sicuramente il des\_ce deve contenere questi campi. Altri campi che possiamo aggiungere noi sono i tipici “char\* buf”, “natl quanti” e salviamoci anche il carattere di stop “char stop”.

Allora, in base a quanto detto, la struttura del des\_ce la definiamo così:



Sempre vedendo la ce\_init, vediamo che c’è un processo esterno che deve avere come nome “estern\_ce” e che come parametro ha l’identificatore del dispositivo ce. Ne terremo conto per dopo.

Facciamo ora la c\_cedmaread:



Innanzitutto, controllo sull’input.

Prima controlliamo che l’id passato sia valido. Se no, abort\_p().

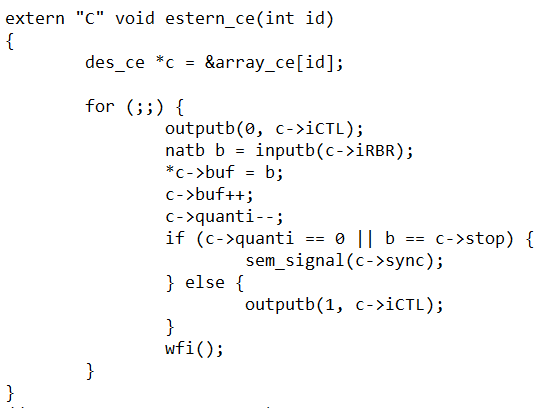
Dovremo fare un controllo sull’intervallo [buf, buf + quanti) (ricordiamo che è corretto farlo, poiché è ancora attivo il trie del processo che ha invocato la primitiva, quindi effettivamente stiamo facendo un controllo sul buffer). Tuttavia, prima di fare questo, ricordiamo che quanti è passato per riferimento, e cioè viene passato l’indirizzo, e quindi bisogna fare un controllo sull’intervallo di indirizzi virtuali [quanti, quanti + sizeof(quanti)]. Questo intervallo, oltre a dover essere fatto da pagine che hanno una traduzione ed accessibile a livello utente, deve essere scrivibile (si deve infatti scrivere a tale indirizzo). Non sappiamo se deve anche far parte della sezione utente/condivisa, dipende da come faremo il processo esterno. Facciamo allora questo controllo con access e, se non viene superato, abort\_p().

A questo punto, facciamo effettivamente il controllo su [buf, buf + quanti). Questo intervallo, oltre a dover essere fatto da pagine che hanno una traduzione ed accessibile a livello utente, deve essere scrivibile (si deve infatti scrivere sul buffer) e deve anche far parte della zona utente/condivisa. Questo possiamo già dirlo senza aspettare a fare il processo esterno: vedendo la traccia, il dispositivo non lavora in DMA, ma sarà il processo esterno ad occuparsi di fare il trasferimento. Essendo un processo, ha un proprio trie di traduzione e lavorerà sul buffer usando indirizzi virtuali, che sono gli indirizzi virtuali [buf, buf + quanti) e quindi per forza il buffer deve trovarsi nella zona utente/condivisa, in modo che usando tali indirizzi virtuali modifichi effettivamente il buffer, e non la sezione privata del proprio spazio di indirizzamento virtuale. Facciamo allora questo controllo con access e, se non viene superato, abort\_p(). Se il controllo non è superato, abort\_p().

Finiti i controlli, si segue lo schema di una primitiva che fa I/O:

1. Recuperiamo il des\_ce;
2. Prendiamo la mutua esclusione, visto che dobbiamo lavorare con risorse condivise a tutti i processi (il des\_ce e la periferica);
3. Ora si avvia l’operazione di I/O. Inizializziamo i campi del des\_ce *buf*, *quanti* e *stop* del des\_ce. In *buf* come sempre va messo l’indirizzo virtuale del buffer, non fisico. Aggiustato tutto, si abilitano le interruzioni mettendo 1 in CTL;
4. A questo punto, aspettiamo che si concluda l’operazione di I/O. In particolare, ogni volta che il dispositivo ha un dato pronto, manderà una richiesta di interruzione, e andrà in esecuzione l’handler, che metterà in esecuzione il processo esterno.
5. Finita l’operazione di I/O, rilasciamo la mutua esclusione. Prima, però, possiamo prevedere (in realtà l’idea viene mentre si fa il processo esterno) che il processo esterno, ogni volta che fa un trasferimento, decrementa c->quanti. In questo modo, facendo quanti -= c->quanti, si mette in quanti correttamente il valore di quanti byte sono stati trasferiti (perché se c->quanti = 0, quindi sono stati trasferiti tutti i byte, allora in *quanti* si lascia il valore originale; se c->quanti != 0, quindi non sono stati trasferiti tutti i byte perché si è incontrato prima un carattere di stop, facendo   
   “byte totali – byte non trasferiti” si trovano i byte trasferiti, che mettiamo in *quanti*).

Facciamo ora il processo esterno:



Anche qui, la struttura è la stessa:

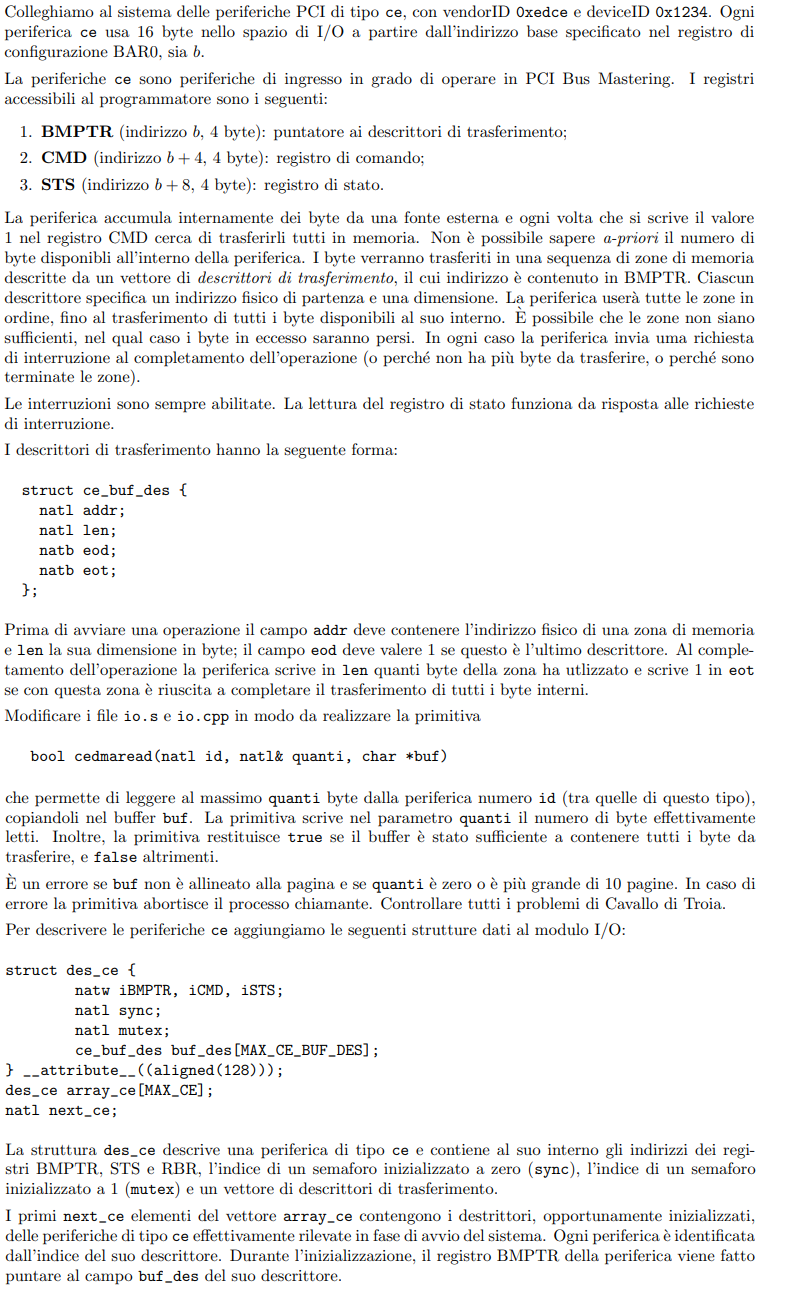
* Recuperiamo il des\_ce;
* Si fa un for infinito seguito dalla wfi().

Vediamo il for. Dentro questo for, il processo esterno deve fare l’operazione di I/O. In particolare, questo processo va in esecuzione quando nel registro RBR è pronto un byte, dunque il processo esterno si deve occupare di trasferire un byte. Come prima cosa, prima di prelevare il byte, disabilitiamo preventivamente le interruzioni, perché se dovesse essere l’ultimo byte da trasferire, se prima leggessimo, la periferica potrebbe preparare un nuovo byte e mandare un’ulteriore richiesta di interruzione. Questa non viene accettata mentre è in esecuzione il processo esterno, ma viene accettata dopo la wfi(), poiché viene mandato l’EOI all’APIC. Si ha quindi che riandrebbe in esecuzione il processo esterno, che farebbe quindi un trasferimento in più rispetto a quelli voluti. Fatto questo, leggiamo RBR, salvando in b ciò che si legge. Questo valore lo mettiamo in \*c->buf, e cioè nell’elemento cui si è arrivati dell’array, incrementiamo c->buf, così che ora punti al byte successivo dell’array e decrementiamo c->quanti per indicare che abbiamo trasferito un byte. A questo punto, si fa un controllo: se c->quanti == 0 (sono stati trasferiti tutti i byte) o b == c->stop (è stato trasferito un carattere di stop), segnaliamo che l’operazione di I/O è finita. Altrimenti, riabilitiamo le interruzioni, così da poter fare ulteriori trasferimenti.

In ogni caso, alla fine si fa wfi().

Tornando all’access fatta per *quanti*, possiamo mettere shared = false, perché il processo esterno non ci accede, ma solo la primitiva, e lo fa mentre è attivo il trie del processo che ha invocato la primitiva, e quindi può benissimo trovarsi anche nella sezione utente/privata.

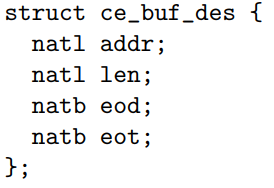
**PROVA D’ESAME 27/07/16**

****

**SOLUZIONE**

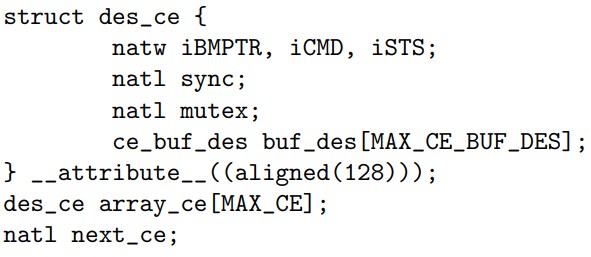
***io.s* lo modifichiamo come al solito.**

**Modifichiamo ora *io.cpp*.** Andiamo a vedere quali aggiunte sono state fatte. Innanzitutto, i byte vengono trasferiti in una sequenza di zone di memoria descritte da un vettore di descrittori di trasferimento, il cui indirizzo è contenuto in BMPTR. Un descrittore di trasferimento ha questa struttura:



Prima di avviare una operazione il campo *addr* deve contenere l’indirizzo fisico di una zona di memoria e *len* la sua dimensione in byte; il campo *eod* deve valere 1 se questo è l’ultimo descrittore. Al completamento dell’operazione la periferica scrive in *len* quanti byte della zona ha utilizzato e scrive 1 in *eot* se con questa zona è riuscita a completare il trasferimento di tutti i byte interni.

Per descrivere le periferiche ce aggiungiamo le seguenti strutture dati al modulo I/O:

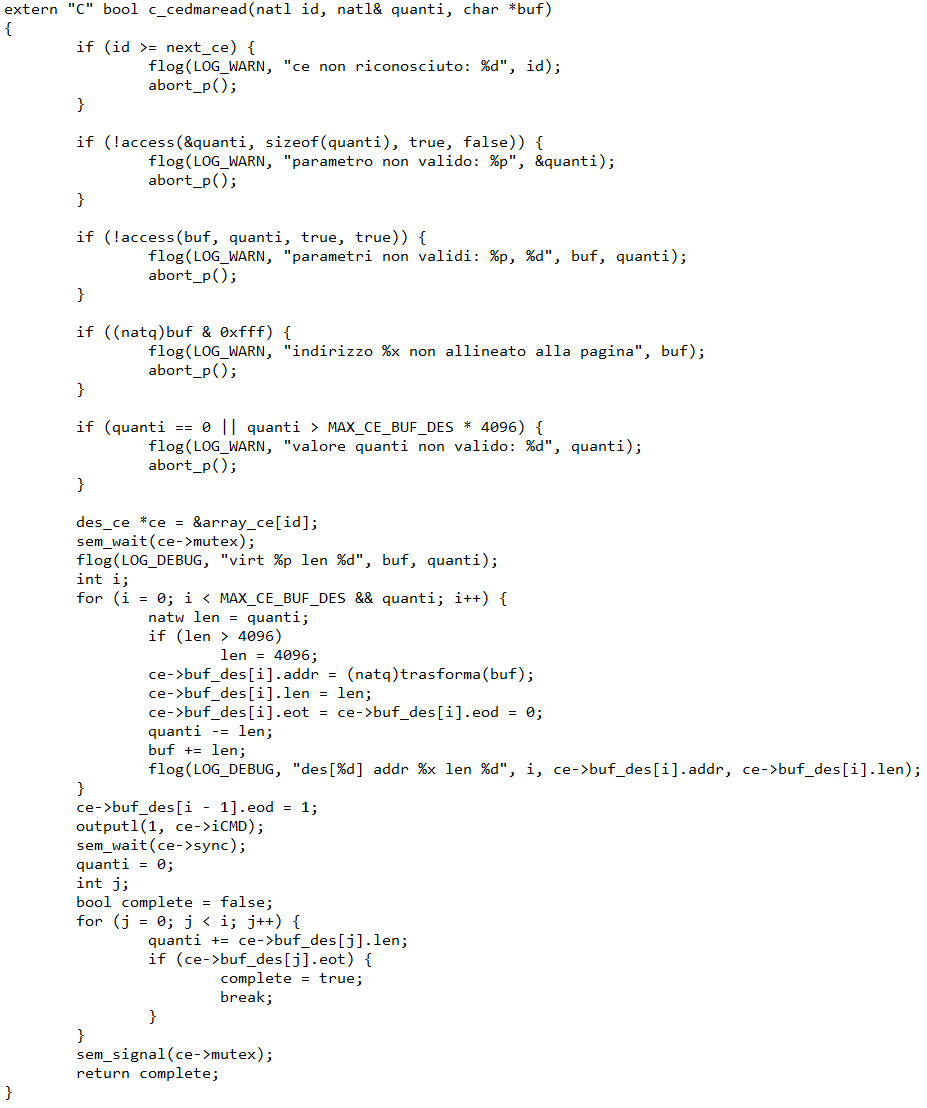


La struttura *des\_ce* descrive una periferica di tipo ce e contiene al suo interno gli indirizzi dei registri BMPTR, STS e RBR, l’indice di un semaforo inizializzato a zero (sync), l’indice di un semaforo inizializzato a 1 (mutex) e un vettore di descrittori di trasferimento.

I primi next\_ce elementi del vettore array\_ce contengono i descrittori, opportunamente inizializzati, delle periferiche di tipo ce effettivamente rilevate in fase di avvio del sistema. Ogni periferica è identificata dall’indice del suo descrittore. Durante l’inizializzazione, il registro BMPTR della periferica viene fatto puntare al campo *buf\_des* del suo descrittore.

Sempre vedendo la ce\_init, vediamo che viene creato un processo esterno che deve avere come nome “estern\_ce” e che come parametro ha l’identificatore del dispositivo ce. Ne terremo conto per dopo.

Facciamo ora la c\_cedmaread:



**Notare innanzitutto una cosa: nel caso di primitive del modulo I/O, si possono restituire valori facendo “return …”, poiché non c’è la carica\_stato dopo che sovrascriverebbe RAX. Tuttavia, facendo questo, va messo come tipo di ritorno il tipo di ciò che va effettivamente restituito facendo “return …”.** Nel nostro caso, la primitiva deve restituire un booleano, e quindi si mette *bool*.

Detto questo, si parte dal controllo sull’input:

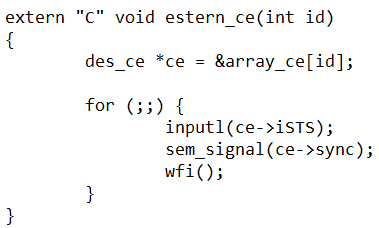
* L’id non deve essere >= next\_ce;
* Controllo sull’intervallo di indirizzi virtuali [quanti, quanti + sizeof(quanti)). shared per ora lo lasciamo in sospeso;
* Controllo sull’intervallo di indirizzi virtuali [buf, buf + quanti). shared per ora lo lasciamo in sospeso;
* buf deve essere allineato alla pagina, e dunque avere i 12 bit meno significativi uguali a 0. Questo si ha se l’and tra *buf* (convertito in un natq, sennò compilatore dà problemi) e 0xFFF mi deve dare 0.
* Quanti deve essere != 0 e non deve essere più grande di 10 pagine. Questo “10 pagine” è espresso dalla costante MAX\_CE\_BUF\_DES, dunque quanti non deve essere > MAX\_CE\_BUF\_DES \* DIM\_PAGINA.

In caso di errore, abort\_p().

La struttura della primitiva che fa I/O è poi la stessa:

1. Si recupera il des\_ce;
2. Prendiamo la mutua esclusione, visto che dobbiamo lavorare con risorse condivise a tutti i processi (il des\_ce e la periferica);
3. Ora si avvia l’operazione di I/O. Dobbiamo scrivere sia nel des\_ce, che nei registri della periferica. In particolare, nel des\_ce dobbiamo scrivere nell’array ce->buf\_des[]. Salviamoci allora un puntatore al primo elemento dell’array. Questo array dobbiamo inizializzarlo fino all’elemento che serve. Scorriamo allora l’array finché quanti > 0 (che decrementeremo noi all’interno del for) e al massimo fino alla fine dell’array (i < MAX\_CE\_BUF\_DES). Dichiariamo una variabile *len*, che conterrà il valore dei byte da trasferire. All’inizio, la inizializziamo a *quanti*, ma al massimo si può fare un trasferimento fino alla fine della pagina, perché il dispositivo opera in Bus Mastering e trasferirà byte consecutivi partendo da un certo indirizzo fisico, e se pagine sono contigue non è detto che i frame siano anch’essi contigui. Il buffer è allineato alla pagina, quindi possiamo trasferire al massimo pagina per pagina, quindi se len > DIM\_PAGINA, mettiamo in len il valore DIM\_PAGINA. Nel campo addr mettiamo trasforma(buf); nel campo len mettiamo len, e poi negli altri due campi mettiamo inizialmente 0. Incrementiamo poi buf di len, così che punti alla pagina successiva (così la trasforma è consistente), e decrementiamo quanti di len, visto che questo descrittore è relativo al trasferimento di len byte, e quindi ci saranno len byte in meno da trasferire (questo serve per la consistenza della condizione “quanti > 0”). Usciti dal for, dentro la variabile *i* (che abbiamo dichiarato fuori dal for) ci sarà il primo indice dell’array su cui non si è scritto nulla. L’ultimo descrittore allora sarà quello di indice *i – 1*, e a quel descrittore mettiamo eod = 1, essendo l’ultimo. Aggiustato tutto, si abilitano le interruzioni mettendo 1 in CMD.
4. A questo punto, si aspetta che l’operazione di I/O si concluda.
5. Una volta conclusa, rilasciamo la mutua esclusione, ma prima di farlo dobbiamo ancora lavorare su risorse condivise, quindi aspettiamo. In particolare, dobbiamo ritornare un booleano che mi dica se il buffer è stato sufficiente a contenere tutti i byte da trasferire, e in più dobbiamo scrivere in *quanti* il numero di byte effettivamente trasferiti. Innanzitutto allora dichiariamo un booleano *complete*, inizialmente a false, e mettiamo *quanti* a 0. Sappiamo che il dispositivo ha scritto sui descrittori di processo, scrivendo in *len* quanti byte della zona ha utilizzato e 1 in eot se con questa zona è riuscita a completare il trasferimento di tutti i byte interni. Allora scorriamo l’array al massimo fino all’ultimo descrittore inizializzato all’inizio (j < i) e ad ogni iterazione incrementiamo *quanti* di   
   ce->buf\_des[j].len. Se poi si vede che ce->bus\_des[j].eot = 1, dunque se con questa zona si è riusciti a completare il trasferimento di tutti i byte interni, impostiamo a *true* il valore di *complete*, e usciamo dal for. Usciti dal for, ora rilasciamo la mutua esclusione (abbiamo finito di lavorare su risorse condivise), e restituiamo *complete,* che varrà true o false, a seconda se il buffer è bastato o meno.

Vediamo ora il processo esterno:



La struttura è la stessa:

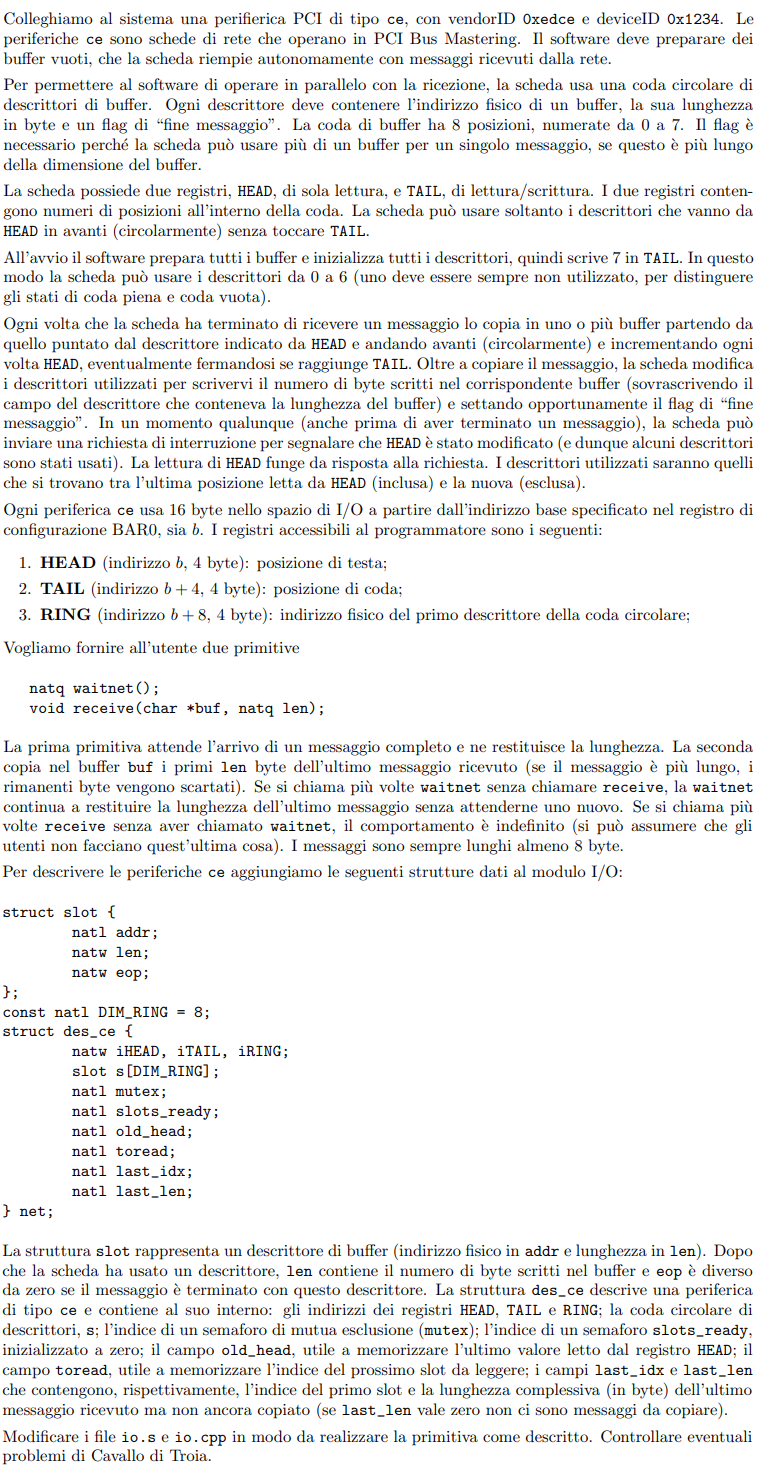
* Recuperiamo il des\_ce;
* Facciamo un for infinito seguito da wfi().

Vediamo il for. Il processo esterno non deve occuparsi di fare ulteriori trasferimenti, quindi basta leggere in STS come risposta alla richiesta di interruzione (così che effettivamente il dispositivo ne possa mandare altre, quando verrà incaricato di fare un altro trasferimento e avrà finito), segnaliamo che l’operazione di I/O è finita e mettiamo wfi().

A questo punto, possiamo tornare alle due access:

* Quella per *quanti* deve avere shared = false, visto che il processo esterno non ci scrive;
* Per quanto riguarda quella per il buffer, nelle soluzioni sta shared = true, ma il processo esterno non ci scrive, quindi secondo me può essere false. Provando a compilare ed eseguire con shared = false, sembra che vada tutto bene.

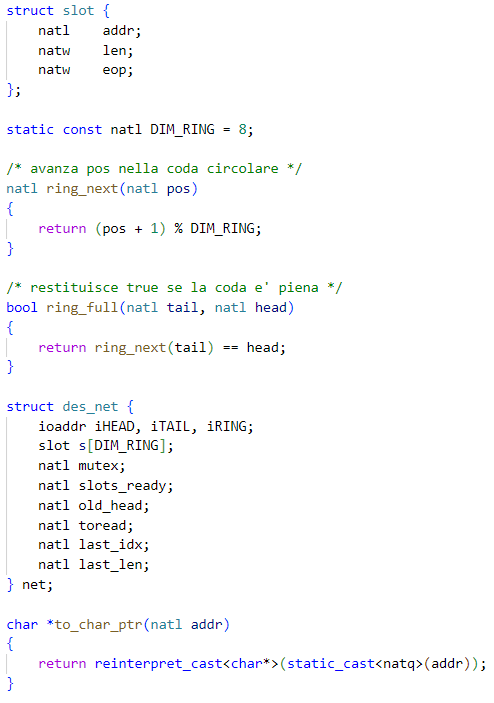
**PROVA D’ESAME 06/02/19**

****

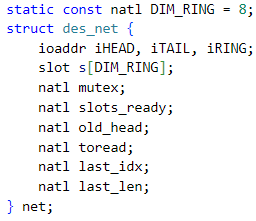
**SOLUZIONE**

***io.s* lo modifichiamo come al solito.**

**Modifichiamo ora *io.cpp*.** Vediamo prima le aggiunte:



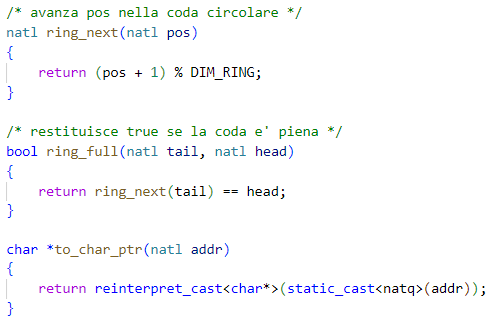
La struttura *slot* rappresenta un descrittore di buffer (indirizzo fisico in *addr* e lunghezza in *len*) Questi buffer vengono creati in fase di inizializzazione, e la periferica ci scriverà autonomamente. Dopo che la scheda ha usato un descrittore (e cioè ha scritto sul buffer corrispondente a tale descrittore), *len* contiene il numero di byte scritti nel buffer e *eop* è diverso da zero se il messaggio è terminato con questo descrittore.



La struttura *des\_net* descrive una periferica di tipo ce e contiene al suo interno:

* gli indirizzi dei registri HEAD, TAIL e RING;
* la coda circolare di descrittori *s*;
* l’indice di un semaforo di mutua esclusione (*mutex*);
* l’indice di un semaforo *slots\_ready*, inizializzato a zero;
* il campo *old\_head*, utile a memorizzare l’ultimo valore letto dal registro HEAD;
* il campo *toread*, utile a memorizzare l’indice del prossimo slot da leggere;
* i campi *last\_idx* e *last\_len* che contengono, rispettivamente, l’indice del primo slot e la lunghezza complessiva (in byte) dell’ultimo messaggio ricevuto ma non ancora copiato (se *last\_len* vale zero non ci sono messaggi da copiare).

Abbiamo poi delle funzioni di utilità**:**

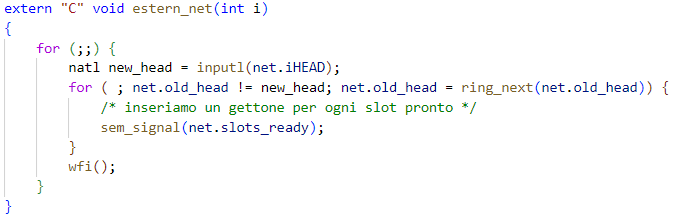
****

Abbiamo poi la net\_init:

Immagine che contiene testo

Descrizione generata automaticamente

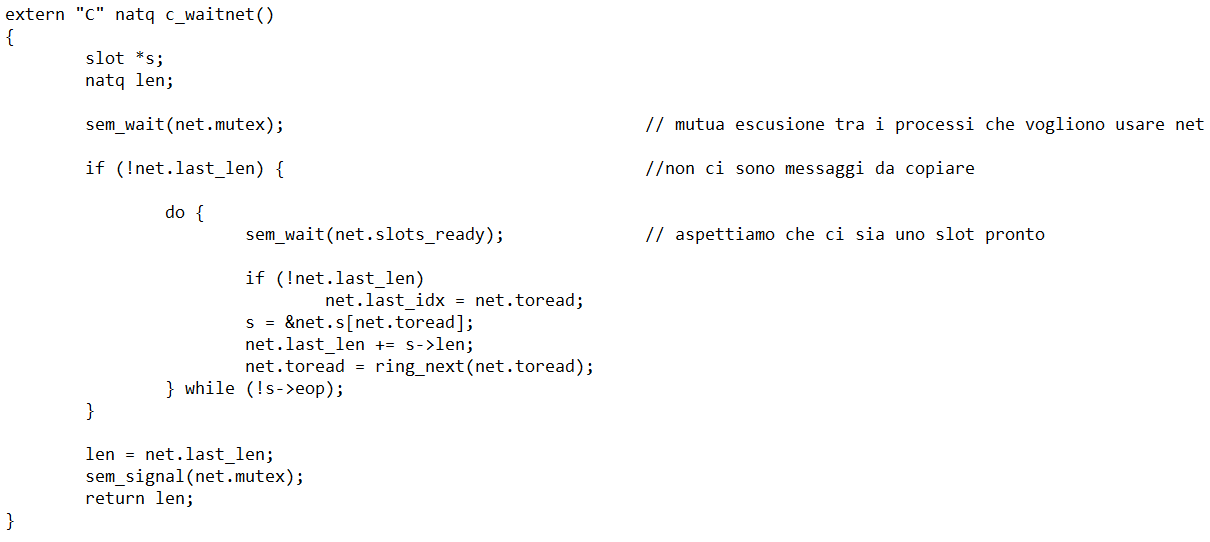
Oltre alla ce\_init, abbiamo già definito un processo esterno:



Questo processo esterno va in esecuzione ogni volta che HEAD viene modificato (e cioè autonomamente la scheda ha scritto su dei buffer, e quindi ha usato alcuni descrittori). Quello che fa questo processo esterno è allora leggere il nuovo valore di HEAD, salvandoselo nella variabile new\_head. A questo punto, dovremmo assegnare a net.old\_head il valore di new\_head (old\_head infatti memorizza l’ultimo valore di HEAD letto). Tuttavia, si fa così: new\_head è il valore attuale di HEAD, e punta ancora non utilizzato. Questo significa allora che ora sono pronti gli slot che vanno da old\_head incluso fino a new\_head escluso. Allora facciamo avanzare circolarmente old\_head (usando la ring\_next) finché non diventa uguale a new\_head, e ad ogni iterazione, per segnalare che c’è un nuovo slot pronto, inseriamo un gettone nel semaforo net.slots\_ready (e quindi ora si capisce l’utilità di questo semaforo. Questo sarà utile per dopo.

**NB:** per messaggio ricevuto si intende usando la waitnet. La periferica scrive quando vuole nei buffer descritti dagli slot, ma per messaggio ricevuto si intende comunque usando la waitnet.

Facciamo ora la c\_waitnet:

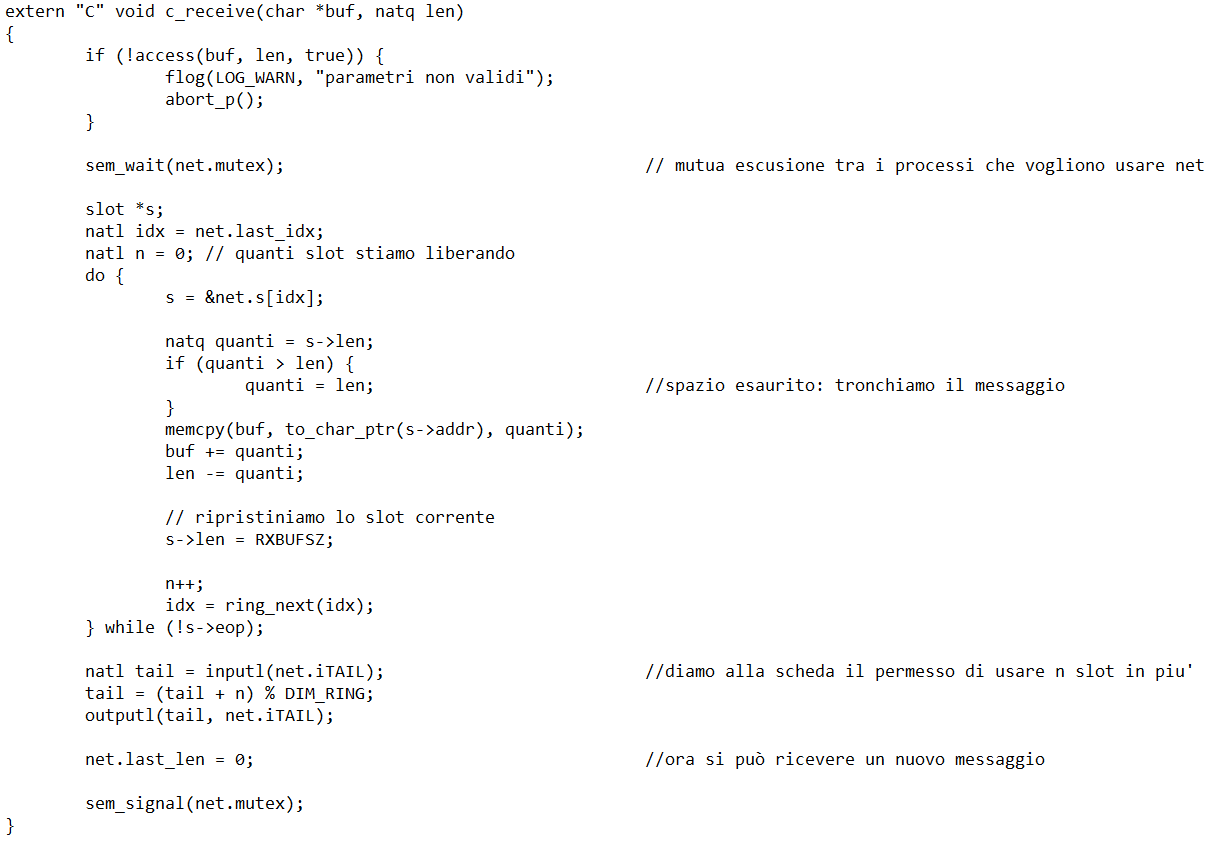
****

Dichiarate quelle due variabili di utilità, prendiamo la mutua esclusione, poiché useremo *net*, che è una risorsa condivisa. Ora bisogna distinguere due casi:

* Se *net.last\_len != 0,* dunque c’è già un messaggio da copiare (cioè si è chiamata più volte la waitnet senza chiamare receive), si restituisce la lunghezza dell’ultimo messaggio senza attenderne uno nuovo, e quindi si lascia la mutua esclusione e si restituisce len = net.last\_len.
* Se *net.last\_len == 0*, dunque non c’è un nuovo messaggio da copiare (non si è chiamata più volte la waitnet senza la receive), allora si deve attendere l’arrivo di un messaggio completo e restituirne la lunghezza. Si fa allora questo: prima avevamo visto nel processo esterno che, quando è pronto un nuovo slot, viene messo un gettone dentro il semaforo net.slots\_ready. Allora, finché non si incontra uno slot con s->eop == 1 che sancisce la fine del messaggio, si prende un gettone dal semaforo, e quindi si aspetta che ci sia uno slot pronto. Quando ne è pronto uno, se è il primo slot che viene letto, assegniamo l’indice di questo slot al campo last\_idx, visto che questo è l’indice del primo slot di quest’ultimo messaggio ricevuto non ancora copiato (e l’indice di questo slot è contenuto in net.toread, che contiene l’indice del prossimo slot da leggere, e cioè l’indice del primo slot del messaggio che si vuole ricevere. Questo indice è quello dello slot attuale, visto che non abbiamo ancora aggiornato *toread*). A questo punto, per ottenere la lunghezza del messaggio ricevuto, visto che dobbiamo anche aggiornare il valore di *net.last\_len*, scorriamo gli slot finché non si verifica la condizione detta prima e ogni volta facciamo net.last\_len += s->len (la periferica infatti avrà modificato il campo s->len, scrivendo il numero di byte scritti nel buffer). Facciamo poi avanzare circolarmente net.toread, visto che ora si è letto lo slot attuale e bisognerà leggere il successivo.

Usciti dal do-while, in net.last\_len avremo la lunghezza di questo messaggio ricevuto. Lo assegniamo a len e restituiamo len.

Facciamo ora la c\_receive:



Come prima cosa, controllo sull’input. Dobbiamo controllare che l’intervallo di indirizzi virtuali [buf, buf + len) sia fatto da pagine su cui c’è mappato qualcosa, siano accessibili a livello utente, siano scrivibili e nella soluzione mette anche shared a true, ma è solo la primitiva scrivere su questo buffer, quindi boh. Se non è rispettato il controllo, abort\_p().

A questo punto, dobbiamo usare il *net,* e poiché è una risorsa condivisa, prima chiediamo la mutua esclusione. Ora dobbiamo scorrere gli slot partendo da quello che ha indice idx = net.last\_idx fino allo slot in cui si ha s->eop == 1 incluso, e dovremmo ogni volta copiare *quanti* = s->len byte dal buffer specificato dallo slot al buffer passato dall’utente. Tuttavia, il buffer passato dall’utente potrebbe non bastare, e quindi per vedere i byte da copiare si vede se quanti > len: se sì, si tronca il messaggio a *len*. Per copiare, usiamo la memcpy. Una volta copiato, incrementiamo buf di quanti, così che ora punti a partire dall’elemento nell’array in cui vanno messa la parte successiva del messaggio, e decrementiamo *len* di quanti, visto che ora abbiamo *quanti* byte in meno da copiare. Si ripristina poi lo slot corrente, mettendo s->len = RXBUFSZ (la periferica infatti, prima di usare gli slot, deve sapere quanto sono grandi, e questo va scritto dentro len. Questo veniva fatto in fase di inizializzazione, ma poi la periferica ci scrive sopra quando scrive dentro lo slot, e quindi bisogna poi ripristinare il campo con il valore pari alla dimensione del buffer, che è RXBUFSZ). Incrementiamo poi n, per indicare che ormai uno slot non serve più, e avanziamo idx al prossimo slot.

Usciti dal do-while, in n abbiamo la dimensione del messaggio che ormai si è letto, e quindi il numero di slot che si possono liberare. Per liberarli, non possiamo modificare HEAD, ma possiamo modificare TAIL, dunque prima prendiamo il suo valore, lo facciamo facciamo avanzare circolarmente TAIL di n, facendo   
tail = (tail + n) % DIM\_RING e poi questo valore lo mettiamo dentro il registro TAIL facendo   
outputl(tail, net.iTAIL). Mettiamo poi a 0 net.last\_len, ad indicare che ora non c’è un messaggio da copiare, e rilasciamo la mutua esclusione, avendo finito di lavorare con net.

In realtà, aggiornare il TAIL alla fine non so se funzioni sempre, perché in caso fossero finiti gli slot la periferica non può mandare altri messaggi. Questo è il caso se un messaggio è più lungo di dim\_array \* 8, e quindi c’è bisogno di fare il giro degli slot più e più volte.

**PROVA D’ESAME 12/06/19**

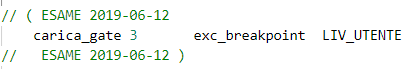
**Immagine che contiene testo

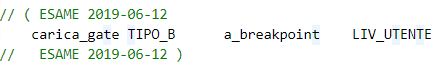
Descrizione generata automaticamente**

**SOLUZIONE**

**Partiamo da sistema.s.** Vediamo le aggiunte:

E’ stato caricato un gate per l’eccezione di tipo 3 (eccezione di breakpoint).





E’ stato già caricato il gate per la primitiva.

Immagine che contiene testo

Descrizione generata automaticamente

E’ stata già definita la parte Assembly della primitiva.

Vedendo invece i tag soluzione, compaiono qui:

Immagine che contiene testo

Descrizione generata automaticamente

Compare nella routine di gestione dell’eccezione “exc\_breakpoint”. Vedendo come è fatta quella sotto, sarà simile, con la differenza che non chiameremo gestore\_eccezioni, visto che l’eccezione di tipo 3 non va gestita come la gestirebbe normalmente il gestore\_eccezioni, ma va gestita in maniera particolare proprio per l’esame. Vedremo dopo.

**Vediamo ora sistema.cpp.** Non è stato modificato, e abbiamo un unico tag soluzione.

Iniziamo allora a scrivere la c\_breakpoint:

Immagine che contiene testo

Descrizione generata automaticamente

La funzione deve ritornare qualcosa, ma essendo una primitiva del modulo sistema, che ha quindi salva/carica\_stato, non si deve fare tramite “return …”, ma mettendo in esecuzione->contesto[I\_RAX]. Visto che allora non usiamo “return …”, mettiamo come tipo di ritorno void.

Innanzitutto, controllo sull’input. Dobbiamo controllare che *rip* faccia parte dell’intervallo   
[ini utn c, fin utn c). In caso contrario, la traccia dice di abortire, quindi c\_abort\_p() e return.

Ci dice poi che questa primitiva deve restituire 0xFFFFFFFF se un altro processo sta già aspettando un breakpoint. Come facciamo a sapere questo? L’idea allora sarebbe implementare una struct, chiamata des\_b, e come campo mettiamo un puntatore al des\_proc *waiting,* che avrà come valore l’indirizzo del processo che sta aspettando (ci salviamo proprio l’indirizzo del des\_proc, anche perché il processo che aspetta un breakpoint si sospende, dunque va messo in una lista di attesa, e questa lista di attesa di un solo elemento la implementiamo con il puntatore *waiting*). Può esserci utile anche salvarci l’indirizzo *rip* in cui c’è il breakpoint, quindi aggiungiamo anche un campo vaddr *rip*, che conterrà tale indirizzo. Non sappiamo ora se servono altre aggiunte, vedremo dopo. In ogni caso, creiamo una variabile des\_b, che chiamiamo *deb\_b*. Questa variabile sarà quella che la primitiva modificherà. Se deb\_b.waiting != nullptr, dunque c’è un altro processo che sta aspettando un breakpoint, restituiamo 0xFFFFFFFF, e dunque lo mettiamo in   
esecuzione->contesto[I\_RAX] e torniamo al chiamante.

Adesso sappiamo che l’indirizzo è valido e nessun’altro sta attendendo un breakpoint, dunque dobbiamo aggiungerlo noi. Per farlo, dobbiamo mettere al posto di del byte puntato da RIP il valore 0xCC, che è il codice operativo dell’istruzione int3. Si potrebbe pensare allora di fare così:

Cioè assegno *rip* ad un puntatore a natb, convertendo prima tale valore in un natb\*, e poi scrivo a tale indirizzo

natb\* puntatore = reinterpret\_cast<natb\*>(rip);

\*puntatore = 0xCC;

Ma siamo sicuri che funziona? In RIP ci può essere l’indirizzo di una pagina che non è scrivibile, e non è scrivibile neanche da parte del modulo sistema, dunque se facessimo una cosa del genere si genererebbe un’eccezione di page fault. Come già detto, però, siamo nel modulo sistema, e possiamo accedere a tutta la memoria fisica attraverso la finestra, e la traduzione è 1:1. Allora facciamo questo: prima trasformiamo rip in un indirizzo fisico (e possiamo usare direttamente la parte C++ della primitiva trasforma, e cioè c\_trasforma, che chiama la funzione trasforma passando come primo parametro l’indirizzo fisico della tabella di livello 4 del trie attualmente attivo). Questo indirizzo fisico corrisponde anche all’indirizzo virtuale della finestra di memoria in cui c’è il byte puntato da RIP, e lo assegniamo al puntatore. **Le pagine della finestra di memoria hanno R/W = 1,** dunque facendo ora \*puntatore = 0xCC, si va a scrivere proprio al byte puntato da RIP attraverso la finestra di memoria. Prima, però, salviamoci il byte che andiamo a sovrascrivere da qualche parte, e dunque aggiungiamo un campo *original* all’interno del des\_b, che conterrà proprio tale byte.

A questo punto, inseriamo il des\_proc del processo che ha invocato la primitiva dentro la lista di attesa e chiamiamo schedulatore(), così da schedulare un nuovo processo. La traccia dice che la primitiva dovrebbe restituire l’id del processo intercettato, ma questo non può saperlo la primitiva, ma si saprà quando effettivamente un processo passerà per tale breakpoint, e quindi quando ormai la primitiva è terminata.

Andiamo ora alla exc\_breakpoint.

Immagine che contiene testo

Descrizione generata automaticamente

Andando a vedere come sono fatte le altre routine simili, si vede che si passano tre parametri e poi si chiama la gestore\_eccezioni. Come già detto, noi non dobbiamo chiamare la gestore\_eccezioni, visto che dobbiamo gestire l’eccezione di tipo 3 in maniera particolare, ma un’altra routine in C++ scritta da noi, che chiamiamo ad esempio c\_breakpoint\_exception. Come parametri passiamo gli stessi che si passerebbero se poi si dovesse chiamare la gestore\_eccezioni, e andando a vedere la gestore\_eccezioni in sistema.cpp, questi sono sono un int che contiene il tipo dell’eccezione (primo argomento), un natq che è particolare per certe eccezioni e che noi possiamo passare semplicemente 0 (secondo argomento) e un vaddr che è il valore del *rip* che si trova in cima alla pila (terzo argomento). Il motivo per cui si passano questi parametri è il seguente: leggendo la traccia, sta scritto che se un processo esegue int3 senza che ciò sia richiesto da una primitiva breakpoint() attiva, il processo deve essere abortito. Andando in sistema.s, sta scritto che “*gestiamo tutte le eccezioni (tranne nmi) nello stesso modo: inviamo un messaggio al log e terminiamo forzatamente il processo che ha causato l'eccezione. Per questo motivo, ogni gestore chiama la funzione gestore\_eccezioni()*”. Quindi nel caso in cui si incontra un breakpoint non è richiesto da una primitiva, si chiama la gestore\_eccezioni, e quindi vanno passati gli stessi tre parametri. Basta allora copiare pari pari il corpo di exc\_breakpoint da sotto, mettendo $3 al posto di $4. NB: abbiamo fatto mov %rsp, %rdx, e cioè abbiamo messo proprio il valore di rsp in rdx, e non invece il valore puntato da rsp, e cioè il valore di rip in cima alla pila. Il motivo lo vedremo dopo. Dunque come terzo parametro non abbiamo il valore di rip nella pila, ma un puntatore al rip nella pila.

Scriviamo ora la c\_breakpoint\_exception:

Immagine che contiene testo

Descrizione generata automaticamente

Questa parte C++ andrà in esecuzione ogni volta che viene incontrata l’istruzione int3. La traccia dice che se un processo esegue int3 senza che ciò sia richiesto da una primitiva breakpoint() attiva, il processo deve essere abortito. Allora, se deb\_b.waiting == nullptr (dunque non c’è nessuna primitiva breakpoint() attiva) o \*rip != deb\_b.rip + 1 (e cioè c’è una primitiva breakpoint() attiva, ma non fa riferimento all’istruzione int3 che si è appena incontrata), si chiama il gestore\_eccezioni (che poi abortirà il processo) e si ritorna al chiamante. Notare quel “deb\_b.rip + 1”: l’eccezione di tipo 3 è un’eccezione trap, dunque il valore di RIP salvato in pila (e puntato da *rip*) è quello dell’istruzione successiva, e quindi del byte successivo a quello dove c’è 0xCC. Per questo motivo bisogna fare il confronto tra \*rip e deb\_b.rip + 1.

Superato questo controllo, significa che si è finiti nel breakpoint richiesto da una primitiva breakpoint() attiva. Allora ripristiniamo il byte al cui posto avevamo messo 0xCC (anche qui poi usiamo la trasforma, per lo stesso motivo visto prima nella c\_breakpoint) e decrementiamo “\*(rip)”, e cioè andiamo a modificare il valore di RIP salvato in pila, così che il processo che è finito nel breakpoint poi torni ad eseguire l’istruzione corretta (quella che non si è eseguita prima, poiché si è eseguita 0xCC).

A questo punto, possiamo effettivamente restituire l’id del processo intercettato: mettiamo dentro deb\_b.waiting->contesto[I\_RAX] il valore di esecuzione->id, dunque il valore dell’id che era in esecuzione prima che si fosse generata l’eccezione.

A questo punto, risvegliamo il processo. Mettiamo prima il processo in *esecuzione* in testa alla lista pronti per gestire un’eventuale preemption, inseriamo il processo puntato da deb\_b.waiting nella lista pronti, azzeriamo il puntatore deb\_b.waiting visto che ora il processo non sta più aspettando e chiamiamo schedulatore() così da schedulare un nuovo processo.

**NOTE:**

* Nel modulo sistema abbiamo:
  + La funzione *trasforma*;
  + La primitiva *trasforma*, che il modulo sistema fornisce al modulo I/O, e la sua parte C++ chiama la funzione trasforma, passando come primo parametro l’indirizzo fisico della tabella di livello 4 del trie attualmente attivo.
* L’eccezione 14 (page fault) è, come dice il nome, fault, e cioè in pila si salva l’indirizzo dell’istruzione che si voleva eseguire e che ha causato il page fault.
* CR2 contiene l’indirizzo che si è provato a tradurre e che ha causato page fault (e **NB**: mica è detto che sia l’indirizzo dell’istruzione che si voleva eseguire! Magari si stava eseguendo una certa istruzione, questa ha richiesto di accedere ad un certo operando in memoria, ma l’indirizzo cui si trova ha causato page fault. Sarà allora questo indirizzo a finire in CR2, e non l’indirizzo dell’istruzione che si è eseguita!)
* In caso di errore, controlla spesso la backtrace, può essere utile. Inoltre, controlla spesso le condizioni, e ragiona sui vari if. Gli errori che fai non sono gravissimi, ma devi stare attento.
* I processi esterni e le primitive di I/O falle sempre con la stessa struttura che si è vista. In particolare:

**primitiva:**

extern “C” tipo\_di\_ritorno *c\_primitiva*(…){

//controllo dell’input

//recupero descrittore di operazioni di I/O

sem\_wait(mutex);

//avvio operazione di I/O

sem\_wait(sync);

sem\_signal(mutex);

//eventuali altre cose

}

**Processo esterno:**

extern “C” void estern\_proc(int i){

//recupero il descrittore di operazioni di I/O

for(;;){

//elaborazioni

wfi();

}

}

* Usa des\_p per ottenere des\_proc.
* distruggi\_processo non fa processi--. Questo è il motivo perché se ad esempio la chiami e non fai processi-- Qemu non si chiude: Qemu si chiude solo quando processi == 0.
* Quando si blocca tutto e non va avanti (no panic), nell’IO può essere magari qualche sem\_wait non ricambiata da sem\_signal, oppure qualche access con controlli troppo stringenti (hai messo true quando non serviva), oppure ancora se è con HEAD e TAIL ti sei dimenticato di fare avanzare uno dei due (in base a quale dei due il software può fare avanzare). Ù
* La access prende come parametro un char\* che è l’indirizzo virtuale del buffer (a differenza della c\_access, che prende invece un vaddr che ha come valore l’indirizzo virtuale del buffer).