Coding part

Locked List.c

Abbiamo due tipi di utilizzo: lock: `gcc try.c locked-list.c -DLOCK -lpthread -o test`

rcu: `gcc try.c rcu-list.c -lpthread -o test`

La versione "lock" lavora con i lock, la versione "rcu" senza lock. Si lavora a livello user, quindi un thread può cedere la cpu (è sleepable). La RCU-list con sleepable thread lavora con metadati. Ricordiamo che l'aggiunta di elementi alla lista non crea problemi a differenza di una rimozione, in quanto la rimozione comporta per thread che ci operano sopra la possibilità di "staccarsi" dalla lista, e perderne il puntamento.
La soluzione è quella già vista: quando si rimuove un elemento, inizia una nuova epoca, e quindi un nuovo contatore. Non posso usarne solo 1 di contatore, altrimenti potrebbero arrivare altri lettori, e non si svuoterebbe mai. L'idea è che ci sia una fase in cui aggiorno il contatore per i futuri lettori, mentre "aspetto" i vecchi lettori sul vecchio contatore. Ciò dipende dalla concorrenza con gli scrittori, cioè quando cambio contatatore, a che punto sono i lettori precedenti, se devo aspettarli o se hanno già visto l'update.

Uso dei counter

Il counter è aumentato in maniera atomica, mediante `__sync_fetch_and_add` : scrivo su un counter " c " che io lettore sto leggendo. scrivo su un counter " c' " che io lettore ho finito di leggere. Non potevo operare direttamente su " c ", e decrementarlo? No, perchè identificare il pointer e incrementare il contatore non è una operazione atomica. Con swap atomica posso porre c = 0 (azzero), ma chi arriva dove scriverà? Atomicamente associo " c " " per evitare confusione. Ho sempre counter per lettori concorrenti e lettori futuri! Alterno " c' " e " c" ", ma uso sempre " c " per l'accesso.

Graficamente

----------|-------+---------|---------->
past (vecchi let.) | rmv elements | future (no prob.)

Quando faccio la remove (all'istante "+", sgancio l'elemento, sto eseguendo una swap, ovvero resetto il counter e so quanti sono entrati prima, alternando il bit più significativo (0 o 1).

Thread house-keeper

Esegue una write_lock (esclude gli scrittori), aggiorna l'epoca, la incrementa, poi aspetta il 'grace period'. Evita l'overflow dell'epoca.

list.h

Implementa le logiche:

- rculist (spinlock solo writer)

- lock-list (spinlock writer/reader)

alcuni metadati sono: pointer al primo elemento (head), spinlock per lavorare, due contatori.

```
typedef struct _rcu_list{
 unsigned long standing[EPOCHS]; // lettori entranti
 unsigned long epoch; // indice epoca
 int next_epoch_index; // indice epoca successiva
 pthread_spinlock_t write_lock; // spinlock per scrivere
element * head; // pointer alla testa
} __attribute__((packed)) rcu_list; // packed: non ottimizzare
```

successivamente eseguo "aligned" perchè devo fare op. atomiche e mi allineo con linee cache.

Operazioni

- INIT: setta epoca = 0, ed epoca futura = 1, setta tutti i lettori 'standing' a 0.
- INSERT: alloca lo spazio per un nuovo elemento, setta valore e ptr nullo al successore. Abbiamo operazioni "mfence" per salvare in memoria. Lo store buffer è FIFO, se così non fosse avrei un ordine inverso nelle operazioni di store.
- SEARCH: ho ptr alla testa e chiave da cercare come parametri. Scorro la lista, prima faccio `my_epoch = __sync_fetch_and_add(epoch,1);` per dire che sono "reader nell'epoca corrente" L'epoca può essere solo "0" o "1", mi basta per sapere dove rilasciare info che sansisce la mia terminazione in lettura.
- REMOVE: Quando concludo, sono al tempo 't', quindi i nuovi 'reader' non possono agganciarsi al vecchio elemento, tuttavia quelli vecchi potrebbero ancora essere collegati!

Lezione 26/10/23

Un modulo di linux è un kernel object, "ko", è un oggetto, non è un eseguibile, è reso eseguibile dal kernel (in particolare thread lato kernel). Noi analizziamo `usctm.c` e `the_usctm.ko` (usato per montare)

inserimento modulo

`sudo insmod the_usctm.ko` C'è funzione di startup che produce messaggi, posso vederli con `dmesg`, tutti i msg scritti in un array circolare. (come un log)

Mi stampa dove è sita la *syscall table*, *sys_ni_syscall* (indirizzi logici). Mi vengono dette anche le entry di *sys_ni_syscall*, anche dove è stata installata una syscall a due parametri (in sys table, posizione con entry nil).
Nb: syscall table compatibile con uno standard, quindi l'entry di spiazzamento 134 è perchè dallo standard risulta che 134 è ni_sys_call. Quindi già so quali aspettarmi (134,174,182,...)

Per ogni modulo che installo c'è anche la parte user, che è di test sostanzialmente, ovvero chiama il servizio che stiamo aggiungendo.

Usando anche qui `dmesg`, vediamo il thread che l'ha richiesta. Vediamo `usctm.c`, di cui ancora non possiamo capire tutto. vediamo init module, chiama `syscall_table_finder();`

Se le trova le scrive in variabile globale, se fallisce vuol dire che non l'ha trovata. Ho quindi le info, scorro le entry per vedere se contengono le entry della `ni_sys_call`. Tutte queste entry buone per scriverci sopra le metto in un altro vettore.

E' presente `ifdef Sys_call_Install`. Tuttavia abbiamo entry che sono read only. In CR0, il 16esimo bit ci dice se la protezione di page table, tlb è attiva o meno. Questo è aggiornabile, anche se a ring0, perchè quando inizializzo sono a ring0. Possiamo fare `read_cr0()`, non `write_cr0()`, dobbiamo implementarla noi! (avremmo problemi con la maschera di bit). Facciamo `unprotect_memory()` (cambio il bit). Nel vettore di appoggio per le ni_sys_call, ci metto `sys_trial`. Poi con `protect_memory()` rimetto lo stato protetto di cr0.

Come è fatta unprotect_memory?

Fa `write_cr0_forced` (passo cr0 e maschera bit per resettare il bit che mi serve). Essa come è fatta? prende unsigned long val, mette in un registro e fa mov su cr0. Fare una write su cr0 non vuol dire che leggo subito, perchè scrivo in un'altra zona della memoria. Devo quindi serializzarla. Lo faccio con " `__force_order__` ". Se non lo facessi, andrei a scrivere sulla page table qualcosa che non è stato ancora scritto.

sys_call_install

Stiamo andando a puntare allo specifico oggetto syscall. Usiamo facility per vedere versione kernel, è > 4.17.0? E' da qui che esistono i wrapper, prima no. Se falso, opero con `asmlinkage`, dispatchato dal dispatcher, perchè deve sapere che deve prendere parametri dalla stack area. Altrimenti, ho syscall a due parametri con due parametri, A e B. sys_trail non esiste in versione >4.17.0, lo creo io in questi casi.

NB: Nel pc del prof non randomizzazione, quindi ciò che trovo non cambia di posizione. `grep sys_call_table /boot/System.map[versione]` mi dice syscall table a 32 e 64 bit. La "principale" che cerchiamo è 64 bit.

Su kernel >5 ho randomizzazione, l'indirizzo non corrisponde a ciò che mi viene fornito a compile time dalla system map.

`cat /sys/module/` ogni cartella è un modulo montato, troviamo anche il modulo appena montato (the_usctm), e varie sottocartelle. Troviamo `sys_call_table_address` con sudo, ci viene dato un indirizzo della syscall table in decimale (prima era esadecimale). Qui trovo l'indirizzo della tabella usabile per fare installazioni.

`sudo cat /sys/module/the_usctm/parameters/free_entries` mi da' entry libere (134,174,177,180,...)

Queste operazioni avvengono anche quando eseguo update kernel a livello di release.

Se smonto syscall rimane qualcosa?

Nel modulo installato c'è una funzione per cleanup, in cui metto il valore che c'era prima. Senza non lo smonto. Ma basta cosi? Se smonto modulo ed elimino syscall, lo faccio in safe solo se tale syscall non è usata da thread. Se smonto, dove tornerebbe il thread?

TLS - soluzione

main crea thread con `pthread`, attività su memoria globale, locale, memoria tls (classica) e un tls alternativo.

- thread deve eseguire funzione target (`the work`).
  Per tale thread esiste già tls (by libreria). Come fa a farlo la libreria?
  Lungo il thread abbiamo starters , funzione e completamento. Quindi devo fare gestione tls negli "starters", potremmo far partire il "nostro starter" al posto di quello default.

`gcc main.c ./lib/tls.c -Xlinker --wrap=pthread_create -lpthread -DTLS -o test -1./include` con quel wrap uso wrapping, (dico di far partire un'altra cosa invece di quella di default) cioè due varianti di pthread_create. Il wrapper di `pthread_create` chiamerà prima o poi la libreria originale, ma diciamo che non deve partire (`*start_routine`), bensì dobbiamo far girare un'altra cosa. Chiamiamo la pthread vera ("real").

Adesso manca implementazione TLS. Come accedo? come metto variabili? Facciamo lavorare qualcuno a low-level. Per mettere oggetti nell'area, ognuno per thread, questi saranno accessibili tramite offset, posso farlo calcolare con "->". Esempio: per una struct uso operatore freccia per offsettarmi sulle componenti.

Con *architecture process control* mi faccio dare la base, applico freccia e mi faccio dare valore che mi interessa.
Ma questo vuol dire usare molte syscall. Alternativa:

Faccio accesso di offset rispetto GS, con offset = 0, abbiamo la base di GS.
Per l'operatore "->" mi serve indirizzo, quindi all'inizio di ogni area TLS scrivo indirizzo area TLS, cosi le altre entry sono solo offset rispetto ad indirizzo che ho salvato. Identifichiamo quindi `PER_THREAD_MEMORY_START` , `PER_THREAD_MEMORY_END` , e `TLS_SIZE` .
Se non la chiudessi, avrei errore dopo l'apertura. TLS position: fa mov di 0 in rax, (spiazzamento 0 a gs), poi faccio mov per ritornare ad indirizzo di questa area.

Per leggere, mi piazzo allo start, applico offset, e leggo con ->

TLS startup

`mmap` usa `TLS_SIZE` (dobbiamo usare il meno possibile le syscall).
Abbiamo ptr a tabella, registriamo due variabili e richiamo la funzione (stiamo tra starters e completamento). Il *main thread* è già tirato su con la libreria, quindi questi discorsi non valgono. Qui lancio thread, lancio wrapper.

- posso wrappare il main, quando compilo e gli giro il mio di main.
- la funzione wrapper viene indicata, nel make si ha `wrap=pthread_create` cioè se trovo questa funzione, passo il rifermento ad una funzione del tipo `wrap_pthread_create` o simile. Poi se la richiamo con `real`, invece chiamo la versione originale.