POLITECNICO DI TORINO

Programmazione di sistema

Esame 08 Luglio 2022 - Programming

	GIORGIO DANIELE LUPPINA 295445
Iniziato	venerdì, 8 luglio 2022, 13:32
Terminato	venerdì, 8 luglio 2022, 14:57
Tempo impiegato	1 ora 25 min.
Valutazione	**8,80** su un massimo di 15,00 (**59**%)

Domanda 1

Completo

Punteggio ottenuto 1,50 su 3,00

Si definiscano i concetti di Dangling Pointer, Memory Leakage e Wild Pointer, facendo esempi concreti, usando dello pseudocodice, che possono generare questi fenomeni.

La condizione di Memory Leakage si verifica quando un'area / un blocco di memoria diventa priva di puntatori per il suo accesso in lettura e/o scrittura: una funzione invocata che alloca dinamicamente memoria e non rilascia esplicitamente il blocco di memoria acquisito, ad esempio.

Un Wild Pointer è un puntatore che manca di inizializzazione, che non è associato ad alcuna area di memoria contenuta nello spazio di indirizzamento del processo. L'utilizzo esplicito di quel puntatore può portare ad accessi illegali e non validi ad aree nelle quali non si hanno i permessi di accesso (arrestati in hardware dalla MMU) oppure ad accessi illegali e non validi potenzialmente disastrosi poichè aree USER ma di altri spazi di indirizzamenti, di altri processi (arrestati in hardware dalla MMU).

Un Danglin Pointer è tale se è assume il valore di un indirizzo RAM valido ma non più attivo.

Memory Leakage, in C

```
##############
main()  {
  func() //Memory Leakage
  func() // Memory Leakage
 /* Altro */

}
```

```
void func (void) {
  int * array = malloc(10 * sizeof (int));
  //foo
  return;  //Avviene la contrazione dello stack ma l'area di memoria allocata dinamicamente
attraverso la funzione malloc perde il suo
            //unico riferimento (int * array): il blocco di memoria non è più accessibile
}
############
```

Dangling Pointer, in C

```
void func(void) {
 int * array = malloc(10 * sizeof(int));
free(array); //Funzione di libreria per la deallocazione esplicita di memoria che sta sullo heap
//Qui, int * array è un puntatore non più attivo ma valido
for(int i=0; i<100; i++) [
  array[i] = 0; //Accessi ad aree di memoria non più attive
}
```

Wild Pointer, in C

```
int main() {
  int * ptr = NULL; //Il puntatore non punta ad un indirizzo effettivo, potenzialmente qualsiasi valore
può essere contenuto in ptr
  for(int i=0; i<100; i++) {
   ptr[i] = 0; //Accesso a indirizzi di memoria potenzialmente disastrosi
 }
```

Commento: memory leakage = memoria non rilasciata. non ha niente a che vedere con essere privi di puntatore. Il wild è tale se lo uso... il dangling è un indirzzo già liberato non "non più attivo" (la RAM non viene disintegrata)

Domanda 2

Completo

Punteggio ottenuto 1,00 su 3,00

In relazione al concetto di Atomic, si definisca cosa esso mira a garantire, come tale garanzia possa essere fornite a livello architetturale, e quali siano i suoi limiti.

Atomic è una struttura dati offerta dalla libreria Rust che offre meccanismi di protezione per strutture dati in ambito concorrente (accessibili contemporaneamente da più flussi di esecuzione all'interno di un comune spazio di indirizzamento). Garantisce che a basso livello si utilizzino istruzioni atomiche basate ad esempio su test-and-set oppure compare-and-swap. I limiti sono imposti proprio dal contatto diretto con dettagli di così basso livello, riducendo potenzialmente la portabilità del codice.

Commento:
detto che Atomic è un concetto generale e detto che la portabilità del codice non è un problema.. i limiti sono altri e si basano sulle garanzie architetturali che qui sono palesemente imparate a memoria e non comprese...

Domanda 3

Completo

Punteggio ottenuto 2,00 su 3,00

All'interno di un programma è definita la seguente struttura dati

```
struct Bucket {
    data: Vec<i32>,
    threshold: Option<i32>
}
```

Usando il debugger si è determinato che, per una istanza di Bucket, essa è memorizzata all'indirizzo 0x00006000014ed2c0.
Osservando la memoria presente a tale indirizzo, viene mostrato il seguente contenuto (per blocchi di 32bit):

```
308a6e01 00600000 03000000 00000000 03000000 00000000 01000000 0a000000
```

Cosa è possibile dedurre relativamente ai valori contenuti dei vari campi della singola istanza?

Fat Pointer di data
- 0x0000600 01e6a803 (8 Byte) è l'indirizzo al primo i32 contenuto all'interno di data
- 0x0000000 00000030 (8 Byte) è il numero di elementi presenti in data
- 0x0000000 00000030 (8 Byte) è la capacità di data

Enum di threshold
- 0x0000000 (4 Byte) è il Tag Some
- 0x00000a0 00000010 (8 Byte) è il valore dentro Option

Commento:
ha fatto spuntare 32bit di più...

Domanda 4

Completo

Punteggio ottenuto 4,30 su 6,00

All'interno di un programma è necessario garantire che non vengano eseguite **CONTEMPORANEAMENTE** più di N invocazioni di operazioni potenzialmente lente.

A questo scopo, è stata definita la struttura dati **ExecutionLimiter** che viene inizializzata con il valore N del limite. Tale struttura è thread-safe e offre solo il metodo pubblico generico **execute(f)**, che accetta come unico parametro una funzione f, priva di parametri che ritorna il tipo generico R. Il metodo execute(...) ha, come tipo di ritorno, lo stesso tipo R restituito da f ed ha il compito di mantere il conteggio di quante invocazioni sono in corso. Se tale numero è già pari al valore N definito all'atto della costruzione della struttura dati, attende, senza provocare consumo di CPU, che scenda sotto soglia, dopodiché invoca la funzione f ricevuta come parametro e ne restituisce il valore. Poiché l'esecuzione della funzione f potrebbe fallire, in tale caso, si preveda di decrementare il conteggio correttamente.

Si implementi, usando i linguaggi Rust o C++, tale struttura dati, garantendo tutte le funzionalità richieste.

```
pub mod implementazione {


  pub struct ExecutionLimiter <R> where R: Send + 'static {
    count: Mutex<usize>,
    cv:  Condvar,
    N: usize,
  }

   impl Send for ExecutionLimiter {}

  //Sintassi per un puntatore a funzione
  //let funzione = (n1: i32, n2: i32) -> i32;

  impl <R> ExecutionLimiter <R> where R: Send + 'static {

    pub fn new(limit: usize) -> Self {
      let execution_limiter = ExecutorLimiter {N: limit, count: Mutex::new(0), cv: Condvar::new()};
      execution_limiter
    }

    pub fn execute (self: & Self, f: ()->R) -> R {

        let count = self.count.lock().unwrap();
        let current_count = count;

        /* Attendi fin quando il current_count (count) è almeno sotto
           un'unità rispetto al limite: "se  tale valore
           è gia pari (o maggiore) al valore N definito all'atto
           della costruzione ..."
        */

        while current_count >= self.N {
          /* Attesa senza consumo di quantum della CPU */
          count = self.cv.wait(count).unwrap();
```

```
        }

        /* Aggiornamento del conteggio */
        current_count = current_count + 1;
        count = current_count;

        /* Esecuzione / Invocazione della funzione */
        let res = f();

        /* Sia che va male, sia che va bene, si aggiusta il conteggio */

        current_count = current_count - 1;
        count = current_count;
        self.cv.wake_all(count);

        res

    }
  }

}
```

Commento:
non mi è chiaro perché si perde tempo con count.... serve una notify_one altrimenti si sprecano cicli macchina.

Inoltre, sarebbe necessario evitare che la new ritorni un EL che non è copiabile...