




# **Introduzione ai concetti di safety**



Nelle pagine man delle funzioni di libreria è indicato l'attributo *Thread Safety* che può essere MT-Safe (Multi Thread Safe), oppure MT-Unsafe, con indicata una ragione.

#### ATTRIBUTES

For an explanation of the terms used in this section, see `attributes(7)`.

Interface	Attribute	Value
<code>strtok()</code>	Thread safety	MT-Unsafe race: strtok
<code>strtok_r()</code>	Thread safety	MT-Safe



Una definizione di **Multi Thread Safe** è

*“a function whose side effects, when called by two or more threads, is guaranteed to be as if the threads each executed the function one after another in an undefined order, even if the actual execution is interleaved” (ISO/IEC 9945:1-1996, §2.2.2).*


Quindi si fa riferimento non ad un intero programma ma ad una singola funzione invocata da due o più thread diversi “contemporaneamente”



## Esempio funzione non MT-safe

```
int somma(int x)
{
    static c=0;
    c += x;
    return c;
}
```


La variabile statica **c** è condivisa tra tutte le invocazioni (a differenza di **x**) sappiamo già che l'incremento non è atomico quindi due chiamate “sovrapposte” rischiamo di incrementare **c** una volta sola.



L'uso di variabili statiche o globali rendono spesso la funzione MT-unsafe ma non è l'unico caso:

```
int incrementa(int *y, int x)
{
    *y += x;
}
```

Due chiamate simultanee con lo stesso puntatore modificano **\*y** in maniera non thread safe



A volte possiamo rendere una funzione MT-safe con gli strumenti che abbiamo visto nel corso

```
struct safeint {  
    int c;  
    pthread_mutex_t *m;}  
  
void somma(struct safeint *y, int x)  
{  
    pthread_mutex_lock(y->m);  
    y->c += x;  
    pthread_mutex_unlock(y->m);  
}
```


Ora due thread *diversi* non causano problemi



Esiste il concetto di [Asynchronous safe function](#):  
(vedere anche **man 7 signal-safety**)

*A function is asynchronous-safe, or asynchronous-signal safe, if it can be called safely and without side effects , without interfering other operations, from within a signal handler context. That is, it must be able to be interrupted at any point to run linearly out of sequence without causing an inconsistent state. It must also function properly when global data might itself be in an inconsistent state.*

Questo tipo di funzioni sono quelle che posso essere utilizzate come signal handler.



I signal handler sono delicati perché un segnale interrompe l'esecuzione e lo stesso thread si mette ad eseguire dell'altro codice, che potrebbe coincidere con quello della funzione interrotta.

Anche se il signal handler non chiama nessuna delle altre funzioni usate dal programma, se arriva un secondo segnale può essere lui che interrompe se stesso.






Questa funzione non è async-signal-safe

```
int incrementa(int *y, int x)
{
    *y += x;
}
```

se l'esecuzione viene interrotta dopo che ho letto \*y ma prima di riscriverlo e viene eseguita una seconda chiamata a **incrementa** il valore finale in \*y non sarà corretto.



```
struct safeint {  
    int c;  
    pthread_mutex_t *m;}
```

```
void somma(struct safeint *y, int x)  
{  
    pthread_mutex_lock(y->m);  
    y->c += x;  
    pthread_mutex_unlock(y->m);  
}
```

se uso un mutex rischio il deadlock perché la seconda invocazione rimane in attesa che la prima sblocchi il mutex, ma la prima attende la terminazione della seconda....



```
/* NO thread-safe, NO async signal-safe */
```

```
void swap(int *x, int *y)
{
    static int t=0; // condivisa dai thread
    t = *x;
    *x = *y;
    *y = t;
}
```



```
/* thread-safe, NO async signal-safe */
```

```
void swap(int *x, int *y)
{
    // le variabili thread local
    // hanno copie distinte per ogni thread
    static thread_local int t=0;
    t = *x;
    *x = *y;
    *y = t;
}
```



La pagina **man 7 signal-safety** fornisce questa semplice ricetta per essere async-signal-safe:


*Ensure that (a) the signal handler calls only async-signal-safe functions, and (b) the signal handler itself is **reentrant** with respect to global variables in the main program.*

E segue elenco di funzioni di libreria async-signal-safe



## Consigli pratici:

- Nei programmi multi-thread evitate variabili globali e statiche, ed eventualmente proteggetele con mutex
- Non usate signal handler ma un thread dedicato che gestisce i segnali con `sigwait`.
- Scrivete signal handler brevi e semplici che chiamano solo funzioni di libreria `async-signal-safe`; se possibile cercate di evitare che lo stesso handler sia interrotto (bloccando i segnali con `pthread_sigmask` o `sigaction`)



Nella documentazione viene citato il concetto di *re-entrancy* che però si può riferire ad entrambi i concetti di safety:

*If a piece of code can be safely reentered again inside a signal handler, it is **reentrant with respect to signal**.*

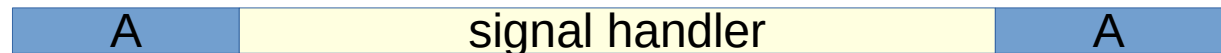
*If a piece of code can be safely reentered again by a different thread, it is **reentrant with respect to thread**.*


Una differenza fra thread e segnali è l'interleaving temporale delle operazioni:

Thread A e B



Thread A interrotto  
dal signal handler





La libreria del C contiene diverse funzioni con il sufisso `_r` che in qualche modo sono legate alla reentrancy, ma non sempre, o non in modo ovvio: [leggete con attenzione le pagine man](#)

Esempi: **`strtok`**, **`strtok_r`**, **`qsort`**, **`qsort_r`**