Scuola universitaria professionale della Svizzera italiana
Dipartimento tecnologie innovative
Istituto sistemi informativi e networking

SUPSI

Strutture dati lock-free

Amos Brocco, Ricercatore, DTI / ISIN

14 maggio 2012

Aspetti negativi dei meccanismi di mutua esclusione / lock

- I thread possono finire in un **deadlock**
- Posso avere il problema di inversione della priorità
  - oppure, un thread che deve eseguire operazioni più dispendiose, in termini di tempo, all'interno di una sezione critica **può rallentare thread più veloci**
- Non è sicuro utilizzare lock all'interno di procedure asincrone (es. gestori di segnali)
- Scalabilità
  - se il lock è conteso da molti thread, posso rischiare di perdere troppo tempo e peggiorare le performances

Strutture dati lock-free

- Garantiscono la mutua esclusione senza l'utilizzo di primitive di sincronizzazione esplicite (mutex, semafori, ecc.)
  - **richiedono comunque il supporto dell'hardware attraverso istruzioni atomiche specifiche (es. CAS)** (possibile anche senza, ma complicato)

CAS (Compare-And-Swap)

- Istruzione hardware **atomica (CMPXCHG)**
  - confronta il contenuto di una cella di memoria con un valore dato, e se sono uguali, assegna un nuovo valore alla cella
- Pseudo codice (deve essere eseguito in maniera atomica!):

```
bool cas (ulong *cella, ulong vecchio, ulong nuovo)
{
        if (*cella == vecchio) {
                *cella = nuovo;
                return true;
        }
        return false;
}
```

Se il valore corrisponde effettua lo scambio

Esempio: uno stack lock-free

```
struct Elemento {
    Elemento*  succ;
    void* dati;
}

Elemento* tos; // Cima dello stack
```

Adattato da http://www.cs.cmu.edu/~410-s05/lectures/L31_LockFree.pdf, Geoff Langdale

Esempio: uno stack lock-free

```
void push(Elemento * e) {
  do {
    Elemento* succ = tos;
    e->succ = succ;
  } while (!cas(&tos, succ, e));
}
```


Ad ogni iterazione guarda se il tos corrisponde ancora a succ, e se sì lo fa puntare al nuovo elemento e

Adattato da http://www.cs.cmu.edu/~410-s05/lectures/L31_LockFree.pdf, Geoff Langdale

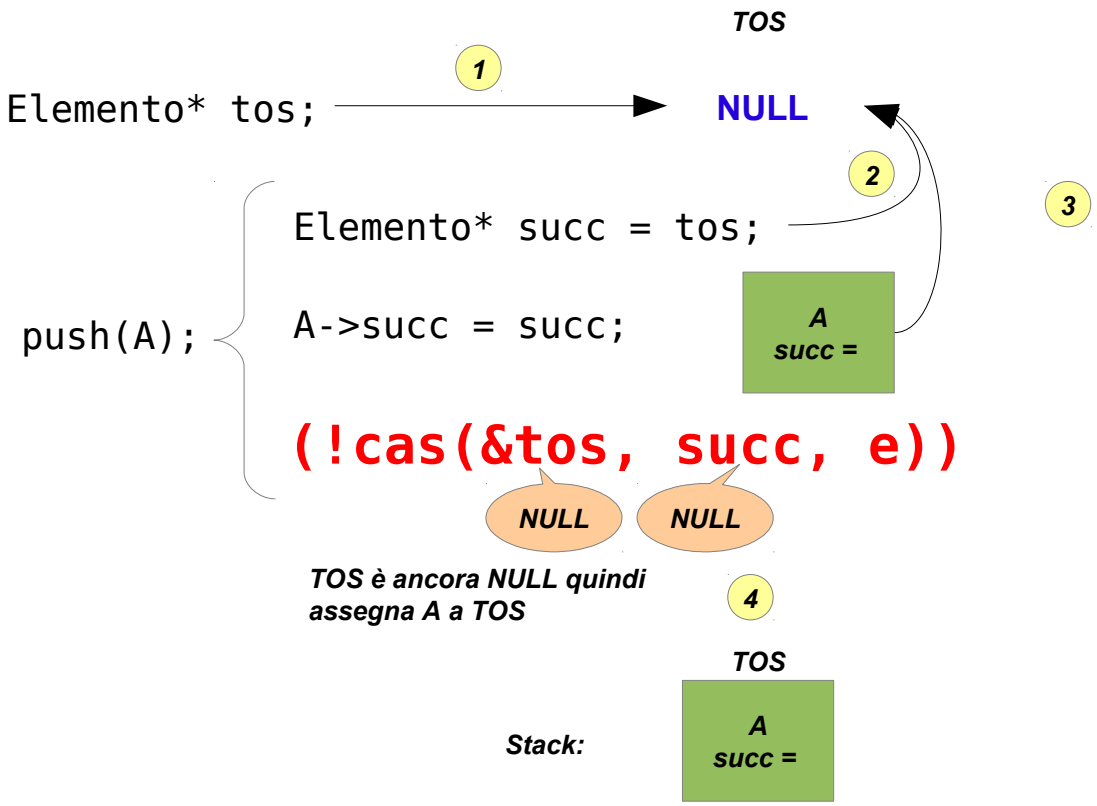
Esempio: uno stack lock-free

```
// Toglie il TOS e lo ritorna
Elemento* pop() {
  for(;;) {
    Elemento* tolto = tos;
    if (!tolto) return NULL;
    Elemento* succ = tolto->succ;
    if (cas(&tos, tolto, succ)) return tolto;
  }
}
```

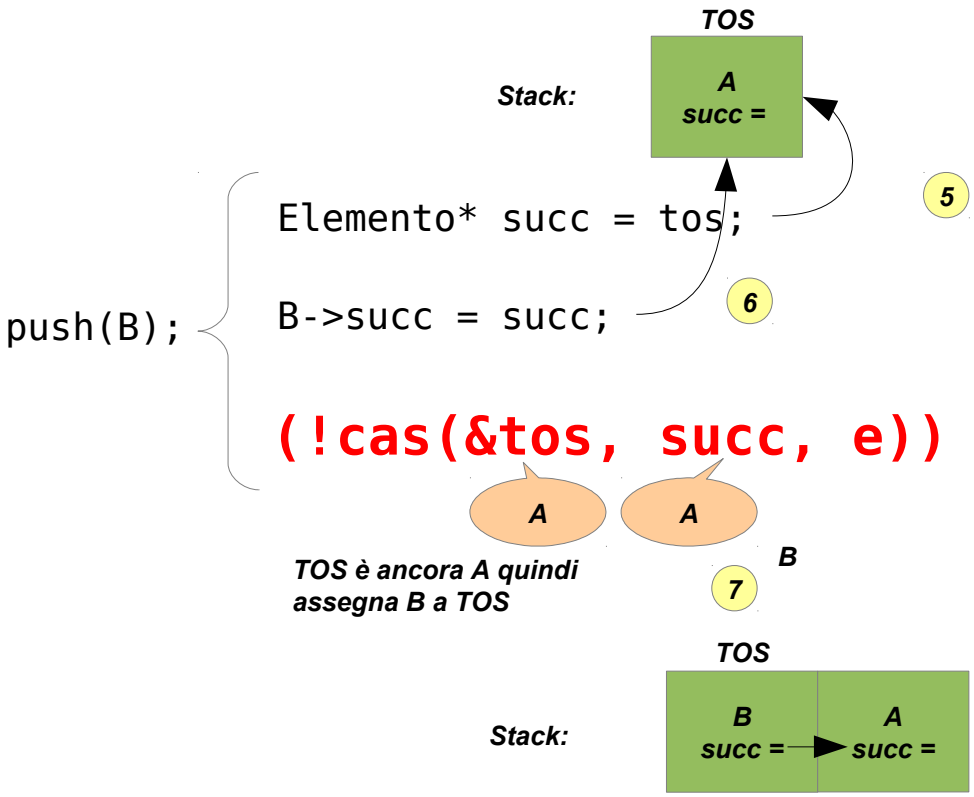
Ad ogni iterazione guarda se il tos corrisponde ancora a tolto, e se sì lo fa puntare al nuovo elemento succ

Adattato da http://www.cs.cmu.edu/~410-s05/lectures/L31_LockFree.pdf, Geoff Langdale

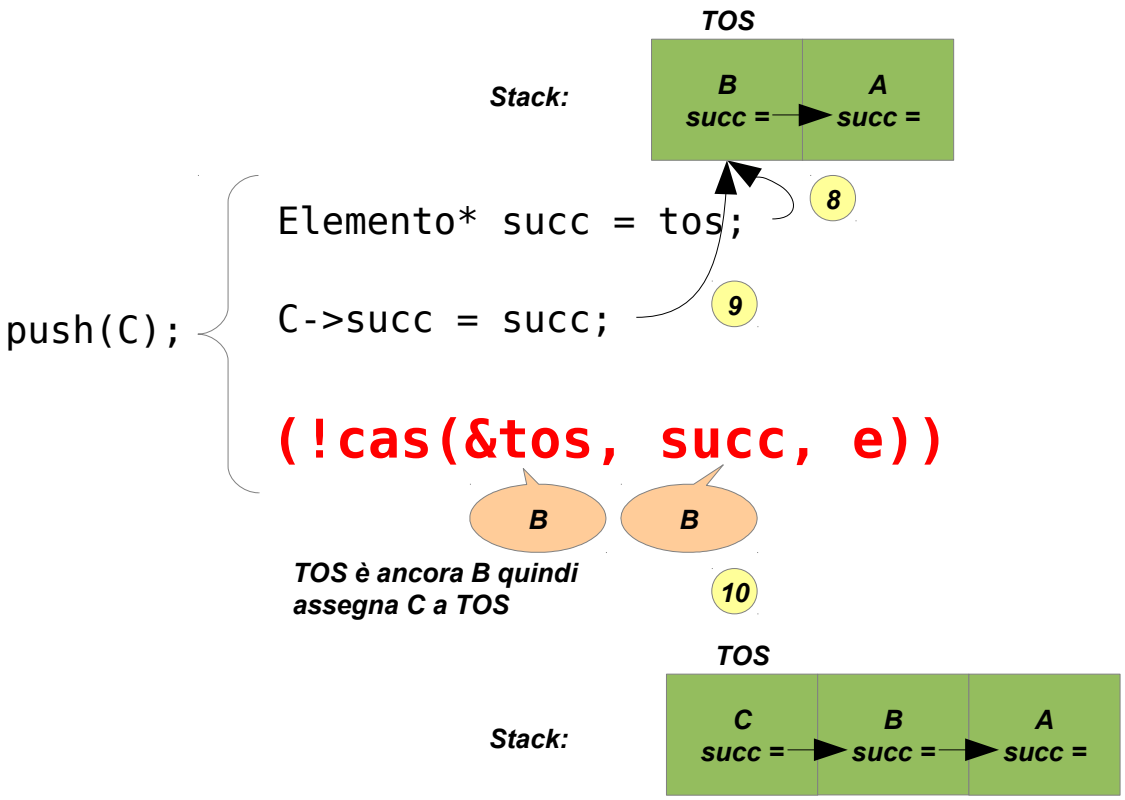
Esempio: uno stack lock-free



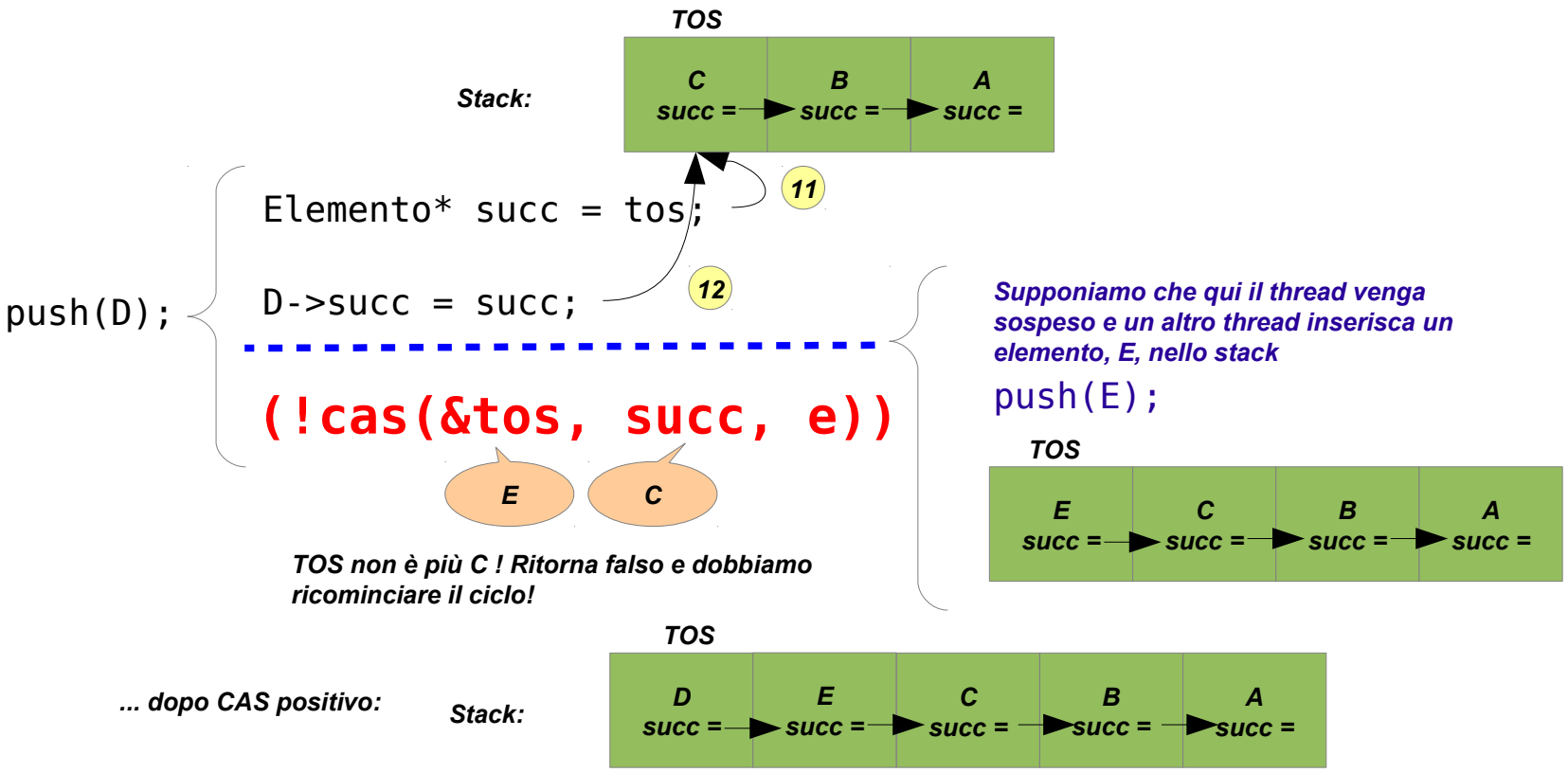
Esempio: uno stack lock-free



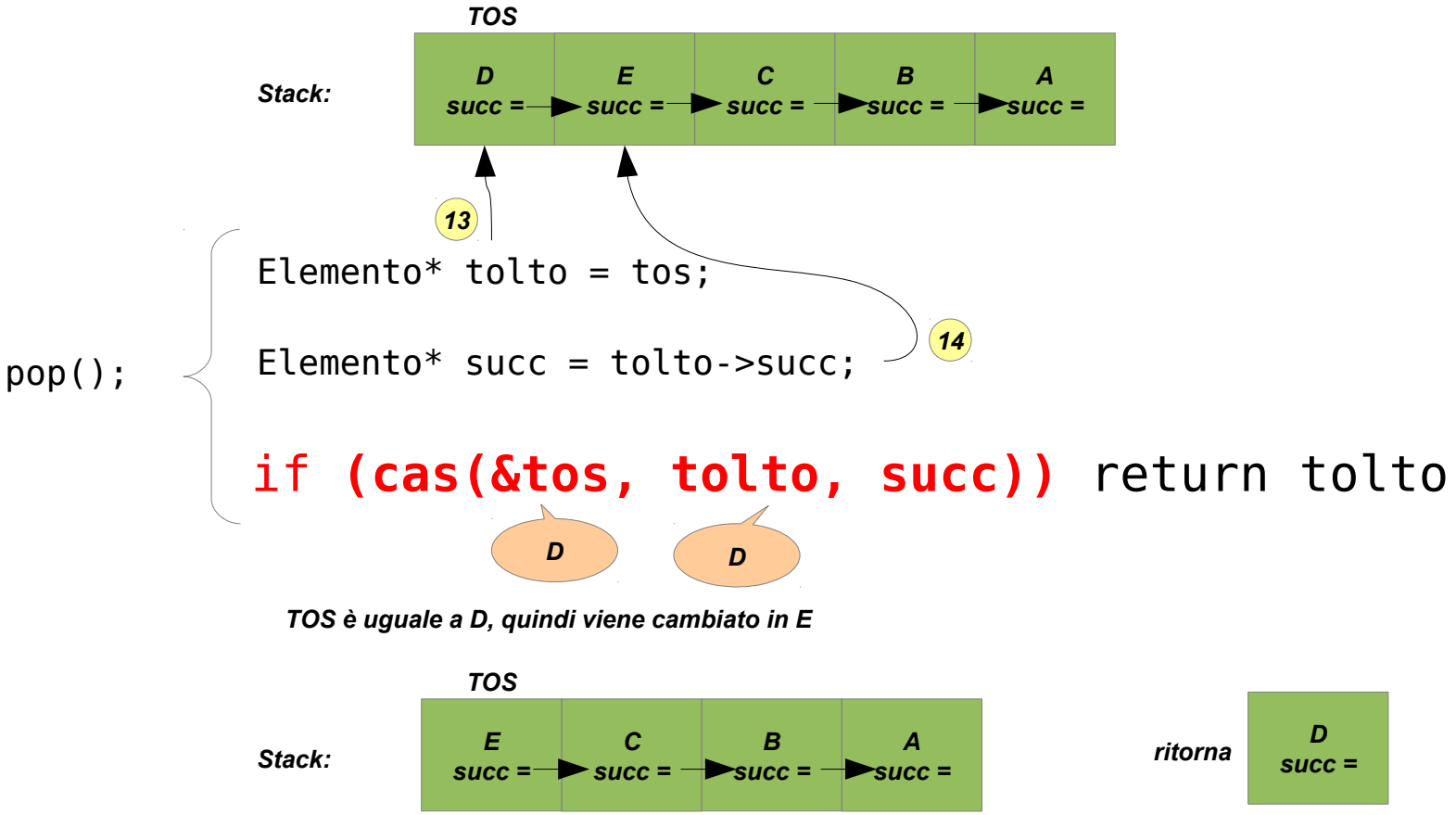
Esempio: uno stack lock-free



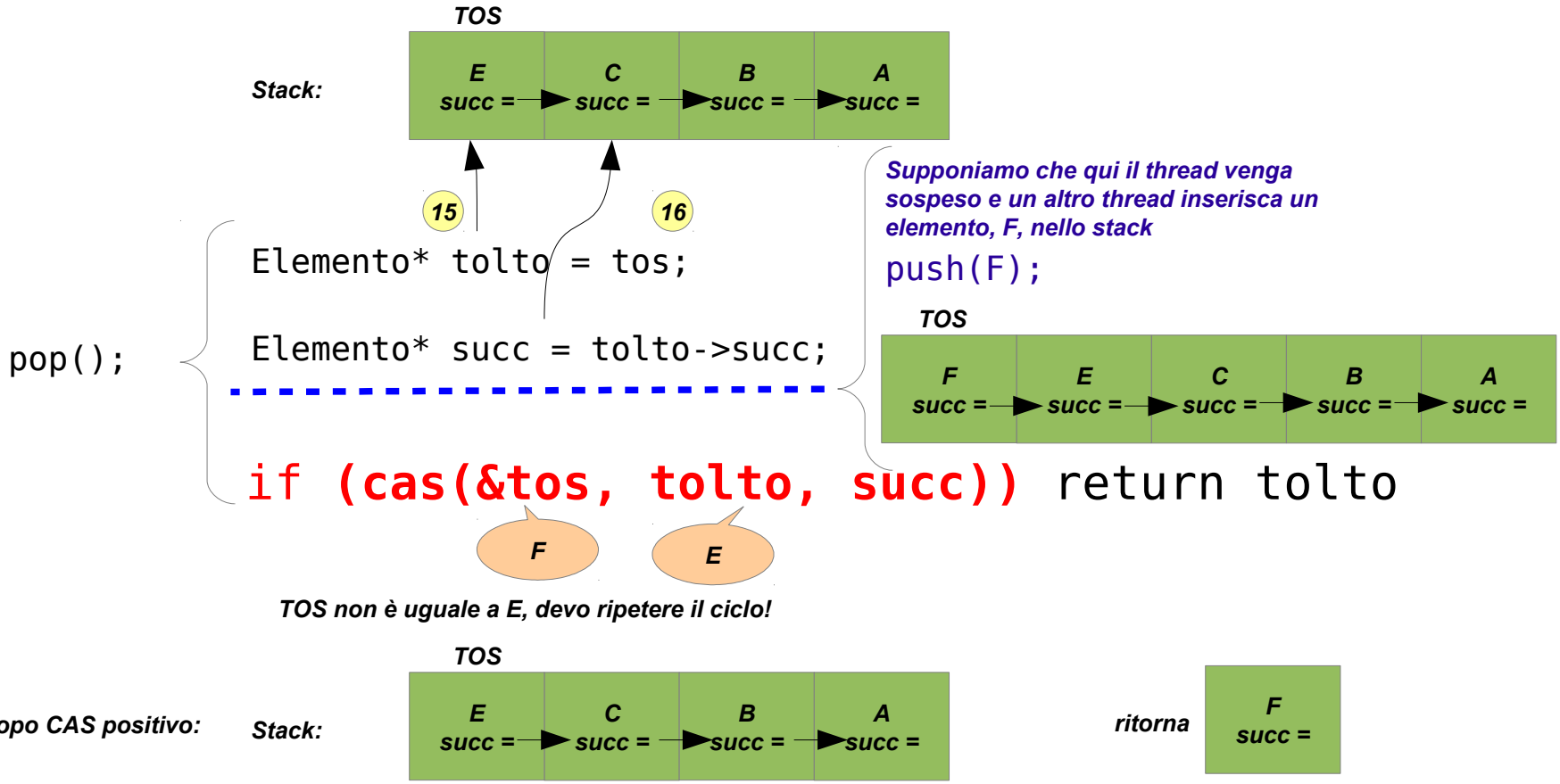
Esempio: uno stack lock-free



Esempio: uno stack lock-free



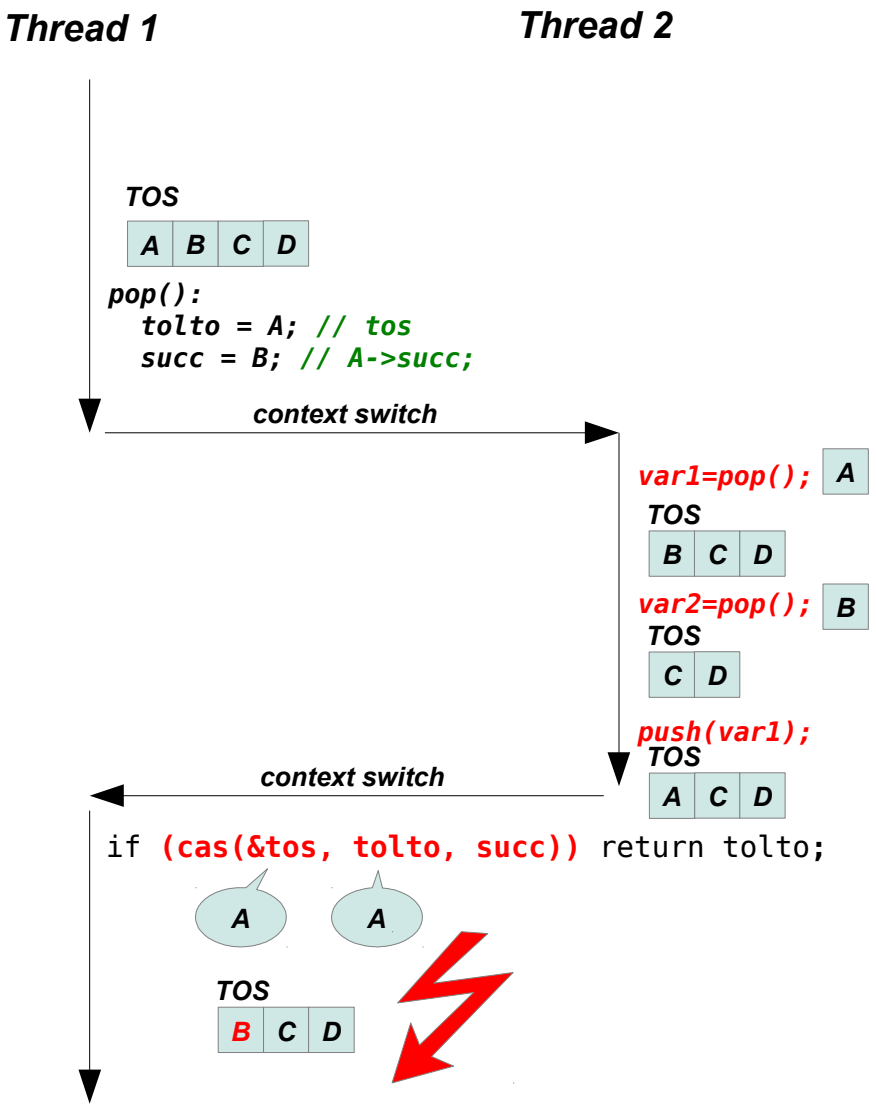
Esempio: uno stack lock-free



Problema ABA

- Il thread 1 legge il valore di una variabile
  - Sulla base di questo valore, il thread esegue delle operazioni
- Il thread 1 viene sospeso, e il controllo passa al thread 2
- Il thread 2 cambia il valore della variabile
  - Se il thread 1 si svegliasse e facesse un CAS si accorgerebbe del cambiamento

  Il thread 2 ripristina il valore della variabile di nuovo al valore osservato dal thread 1
- Il thread 1 si sveglia e non si accorge di nulla...



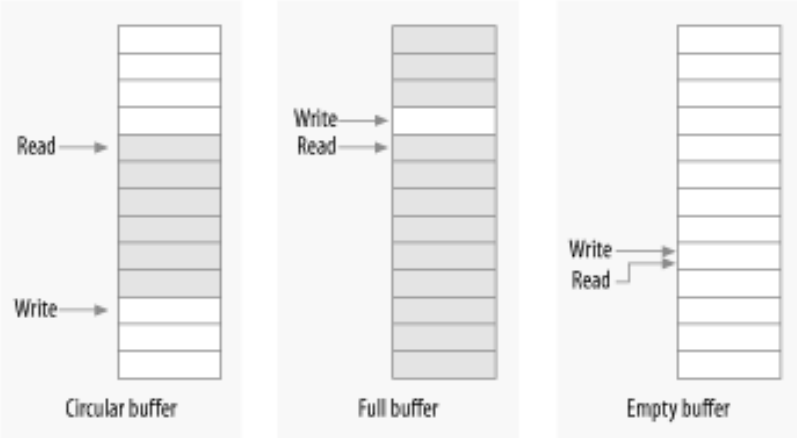
Soluzioni al problema ABA

- Mantenere un contatore che conta ogni modifica fatta alla struttura dati
  - posso verificare se è successo qualcosa

- Non eliminare gli elementi immediatamente (nell'esempio precedente, non liberare la memoria occupata da B, così il puntatore ritornato al thread 1 rimane valido)

Esempio: buffer circolare FIFO senza lock (e senza CAS!)

- Limitazioni: **è possibile gestire al massimo un lettore e uno scrittore** (se non vogliamo usare lock)
- Implementato nel kernel Linux come KFIFO (kfifo.h, kfifo.c)
  - http://www.cs.fsu.edu/~baker/devices/lxr/http/source/linux/kernel/kfifo.c
  - http://www.cs.fsu.edu/~baker/devices/lxr/http/source/linux/kernel/kfifo.h



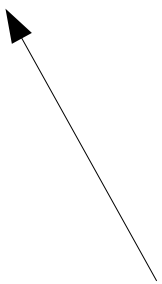
da Linux Device Drivers 3rd Edition, di Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. Creative Commons Attribution-ShareAlike 2.0 license. http://lwn.net/Kernel/LDD3/

KFIFO

```
struct kfifo {
  unsigned char *buffer;  /* the buffer holding the
                             data */
  unsigned int size;      /* the size of the allocated
                             buffer */
  unsigned int in;        /* data is added at offset
                             (in % size) */
  unsigned int out;       /* data is extracted from
                             off. (out % size) */
  spinlock_t *lock;       /* protects concurrent
                             modifications */
};
```

Scrittore inizia a scrivere da qui

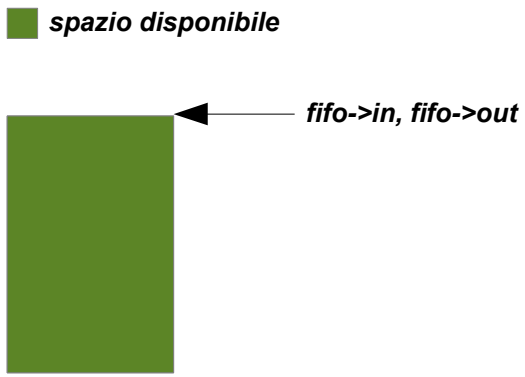
Lettore legge da qui



Non necessario con solo 1 lettore e 1 scrittore

KFIFO

```
39 struct kfifo *kfifo_init(unsigned char *buffer, unsigned int size,
40                          gfp_t gfp_mask, spinlock_t *lock)
41 {
42         struct kfifo *fifo;
43
44         /* size must be a power of 2 */
45         BUG_ON(!is_power_of_2(size));
46
47         fifo = kmalloc(sizeof(struct kfifo), gfp_mask);
48         if (!fifo)
49                 return ERR_PTR(-ENOMEM);
50
51         fifo->buffer = buffer;
52         fifo->size = size;
53         fifo->in = fifo->out = 0;
54         fifo->lock = lock;
55
56         return fifo;
57 }
```



KFIFO

```
 81 static inline unsigned int kfifo_put(struct kfifo *fifo,
 82                                        unsigned char *buffer, unsigned int len)
 83 {
 84         unsigned long flags;
 85         unsigned int ret;
(...)
 89         ret = __kfifo_put(fifo, buffer, len);
(...)
 93         return ret;
 94 }
```

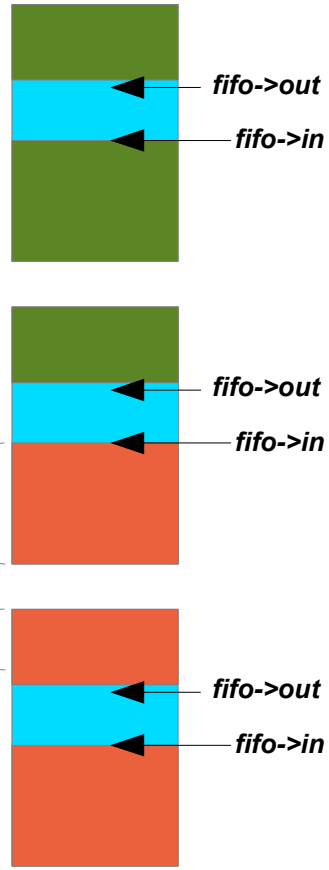
KFIFO

```
unsigned int __kfifo_put(struct kfifo *fifo,
120                          unsigned char *buffer, unsigned int len)
121 {
122         unsigned int l;
123
124         len = min(len, fifo->size - fifo->in + fifo->out);
125
126         /*
127          * Ensure that we sample the fifo->out index -before- we
128          * start putting bytes into the kfifo.
129          */
(...)
133         /* first put the data starting from fifo->in to buffer end */
134         l = min(len, fifo->size - (fifo->in & (fifo->size - 1)));
135         memcpy(fifo->buffer + (fifo->in & (fifo->size - 1)), buffer, l);
136
137         /* then put the rest (if any) at the beginning of the buffer */
138         memcpy(fifo->buffer, buffer + l, len - l);
139
140         /*
141          * Ensure that we add the bytes to the kfifo -before-
142          * we update the fifo->in index.
143          */
(...)
147         fifo->in += len;
148
149         return len;
150 }
```

spazio disponibile

spazio disponibile

dati nel buffer



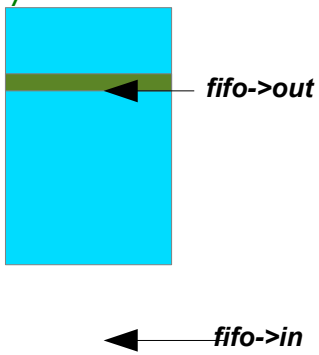
Visto che fifo->size è una potenza di 2 posso fare l'operazione modulo con un AND binario

KFIFO

```
static inline unsigned int kfifo_get(struct kfifo *fifo,
106                                      unsigned char *buffer, unsigned int len)
107 {
108         unsigned long flags;
109         unsigned int ret;
(...)
113         ret = __kfifo_get(fifo, buffer, len);
114
115         /*
116          * optimization: if the FIFO is empty, set the indices to 0
117          * so we don't wrap the next time
118          */
119         if (fifo->in == fifo->out)
120                 fifo->in = fifo->out = 0;
(...)
124         return ret;
125 }
```

KFIFO

```
unsigned int __kfifo_get(struct kfifo *fifo,
166                          unsigned char *buffer, unsigned int len)
167 {
168         unsigned int l;
169
170         len = min(len, fifo->in - fifo->out);
171
172         /*
173          * Ensure that we sample the fifo->in index -before- we
174          * start removing bytes from the kfifo.
175          */
(...)
179         /* first get the data from fifo->out until the end of the buffer */
180         l = min(len, fifo->size - (fifo->out & (fifo->size - 1)));
181         memcpy(buffer, fifo->buffer + (fifo->out & (fifo->size - 1)), l);
182
183         /* then get the rest (if any) from the beginning of the buffer */
184         memcpy(buffer + l, fifo->buffer, len - l);
185
186         /*
187          * Ensure that we remove the bytes from the kfifo -before-
188          * we update the fifo->out index.
189          */
(...)
193         fifo->out += len;
194
195         return len;
196 }
```



Algoritmi senza attesa (wait-free)

- **!= Lock-free**
- **Senza attesa**: ogni operazione termina in un numero finito di passi
  - implica lock-free (non deve essere possibile attendere indefinitamente)
    - ma lock-free non implica wait-free (es. nello stack abbiamo un loop che può ripetersi indefinitamente)
- Più difficile da implementare rispetto a lock-free
  - in alcuni casi impossibile, o possibile solo per un numero prefissato di thread
- Veramente utile solo se abbiamo bisogno di garanzie circa i tempi di esecuzione (es. sistemi hard real-time)

Risorse, librerie lock-free

- **http://www.rossbencina.com/code/lockfree**