Sistemi Operativi Modulo I

Primo canale (A-L) e Teledidattica

A.A. 2021/2022

Corso di Laurea in Informatica

La Gestione della Concorrenza

Igor Melatti

Sapienza Università di Roma
Dipartimento di Informatica

Roadmap

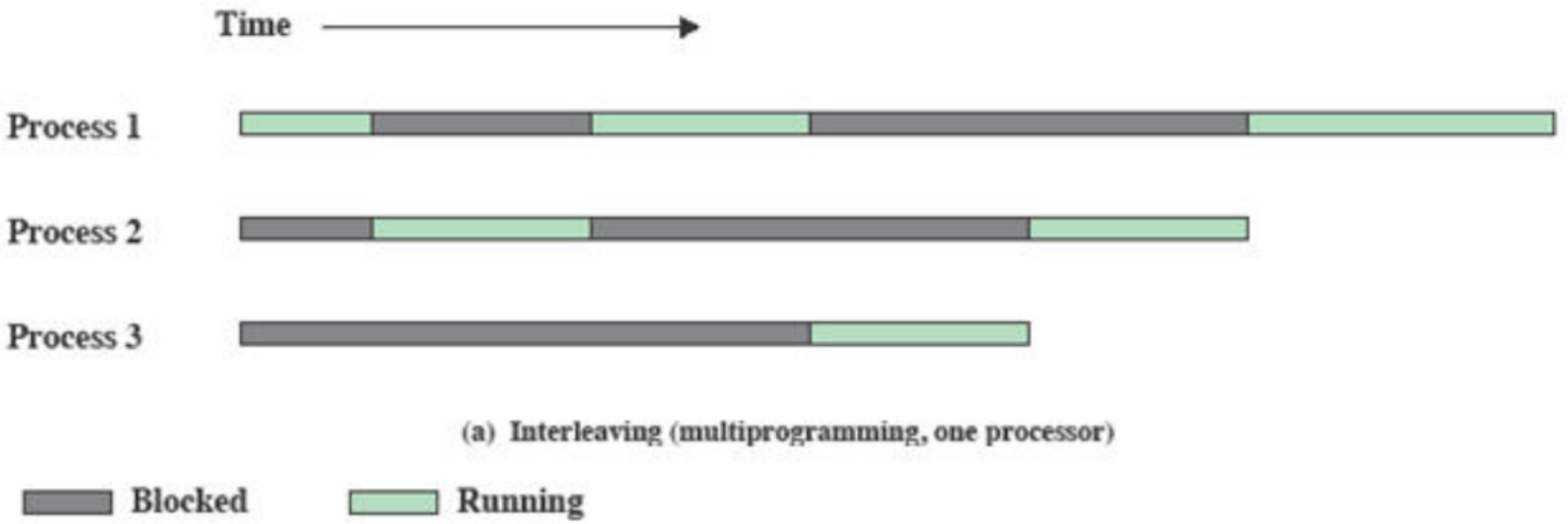
- Concetti basilari di concorrenza
- Mutua esclusione: supporto hardware
- Semafori
- Passaggio di messaggi
- Problema dei lettori/scrittori
- Equivalenze

Processi Multipli

- Per i SO moderni, è essenziale supportare più processi in esecuzione
  - multiprogrammazione
  - multiprocessamento (*multiprocessing*)
  - computazione distribuita (cluster)
- Grosso problema da affrontare: la concorrenza
  - gestire il modo con cui questi processi interagiscono

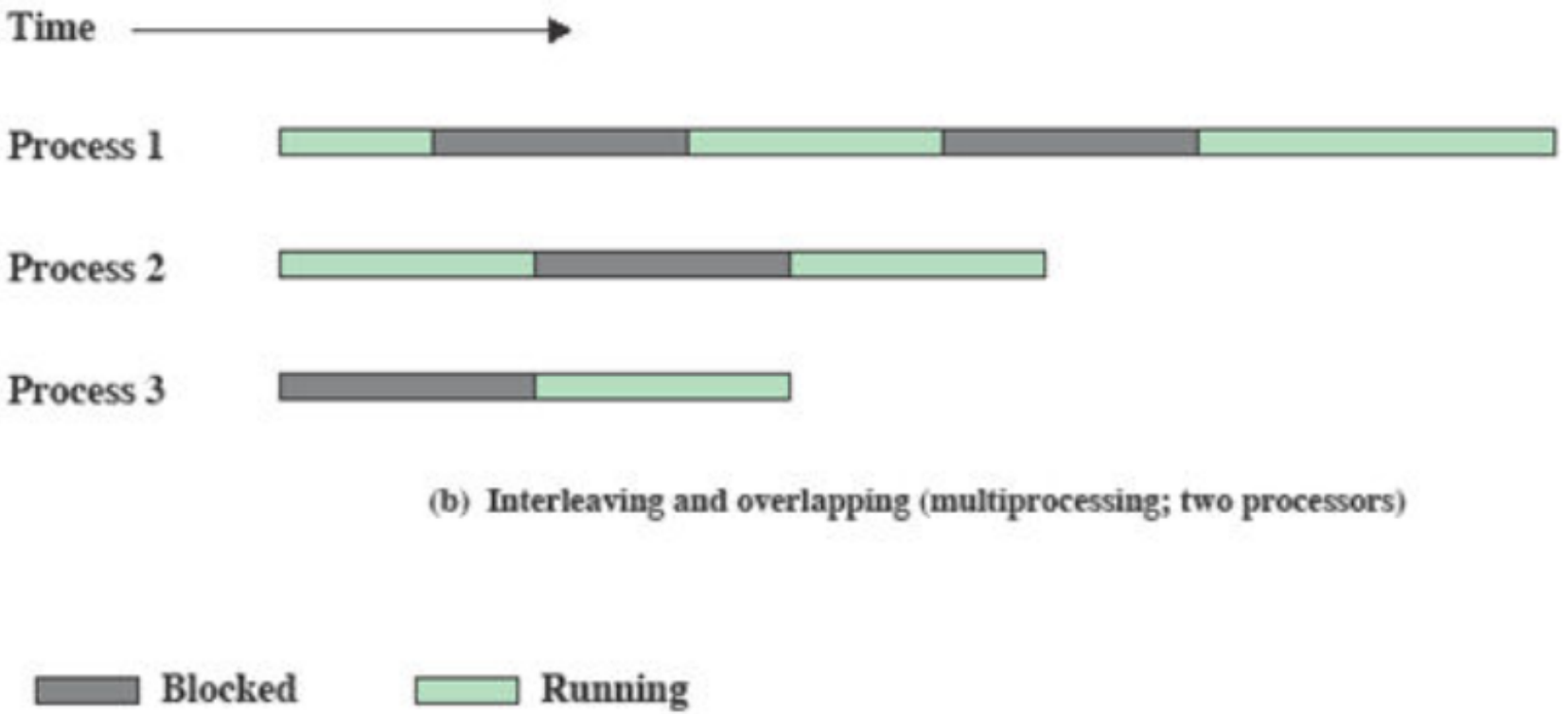
Multiprogramming

Se c'è un solo processore, i processi si alternano nel suo uso (*interleaving*)



Multiprocessing

Se c'è più di un processore, i processi si alternano (*interleaving*) nell'uso di un processore, e possono sovrapporsi nell'uso dei vari processori (*overlapping*)



Concorrenza

Si manifesta nelle seguenti occasioni:

- Applicazioni multiple
  - condivisione di tempo di calcolo
- Applicazioni strutturate per essere parallele
  - estensione della progettazione modulare
- Struttura del sistema operativo
  - gli stessi SO operativi sono costituiti da svariati processi o thread in esecuzione parallela

Terminologia

Operazione atomica una sequenza indivisibile di comandi

- nessun altro processo può vedere uno stato intermedio della sequenza o interrompere la sequenza

Sezione critica parte del codice di un processo in cui c'è un accesso esclusivo ad una risorsa condivisa

- nessun altro processo che voglia accedere in modo esclusivo alla stessa risorsa può farlo

Mutua esclusione requisito che impone che un solo processo sia in una data sezione critica (per risorsa condivisa)

Corsa critica (*race condition*) violazione della mutua esclusione

- il risultato dipende da come i processi si alternano

Terminologia

Stallo (*deadlock*) situazione nella quale due o più processi non possono procedere con la prossima istruzione, perché ciascuno attende l'altro

Stallo attivo (*livelock*) situazione nella quale due o più processi cambiano continuamente il proprio stato, l'uno in risposta all'altro, senza fare alcunché di "utile"

Morte per fame (*starvation*) un processo, pur essendo ready, non viene mai scelto dallo scheduler

Concorrenza: Difficoltà

- LA difficoltà: non si può fare nessuna assunzione sul comportamento dei processi
  - e neanche su come funzionerà lo scheduler
- Condivisione di risorse
  - es.: una stampante
  - ma anche più semplice: 2 thread che accedono alla stessa variabile globale
- Gestione dell'allocazione delle risorse condivise
  - diventa impossibile la gestione ottima
  - ad es., un processo potrebbe richiedere un I/O e poi essere rimesso in ready prima di usarlo: quell'I/O va considerato locked oppure no?
- Difficile tracciare gli errori di programmazione
  - spesso il manifestarsi di un errore dipende dallo scheduler e dagli altri processi presenti
  - rilanciare l'ultimo processo spesso non riproduce lo stesso errore

Esempio Facile

```
/* chin e chout sono globali */
void echo()
{
     chin = getchar();
     chout = chin;
     putchar(chout);
}
```

Esempio su Un Processore

```
Process P1
          .
chin = getchar();
          .
chout = chin;

putchar(chout);
          .
          .
```

```
Process P2
          .
          .
chin = getchar();
          .
chout = chin;
          .
putchar(chout);
          .
```

Esempio su Più Processori

```
Process P1
          .
chin = getchar();
          .
chout = chin;
putchar(chout);
          .
          .
```

```
Process P2
     .
      .
chin = getchar();
chout = chin;
       .
putchar(chout);
     .
```

Restrizione all'Accesso Singolo

- Si supponga di far sì che dentro la funzione `echo` ci possa entrare solo un processo alla volta
  - l'intera funzione `echo` viene vista come una risorsa condivisa
- P1 ci entra per primo
- P2 ci prova ma viene bloccato finché P1 non finisce
- A quel punto, P2 viene riesumato e può essere completato
- Niente più comportamenti indesiderati
  - $\rightarrow$ occorrono meccanismi di protezione

Race Condition

- Si ha una *corsa critica* quando:
  - più processi o thread leggono e scrivono su una stessa risorsa condivisa
  - lo fanno in modo tale che lo stato finale della risorsa dipende dall'ordine di esecuzione dei detti processi e thread
  - come prima: il contenuto di `chin`, e quindi di cosa viene poi scritto su monitor, dipende dal processo che esegue l'assegnamento per ultimo
- In particolare, il risultato può dipendere dal processo o thread che finisce per ultimo
- *Sezione critica* è la parte di codice di un processo che può portare ad una corsa critica

Per Ciò che Riguarda il SO

- Quali problemi di progetto e gestione sorgono dalla presenza di concorrenza?
- Il SO deve
  - tener traccia di vari processi
  - allocare e deallocare risorse (processore, memoria, file, dispositivi di I/O)
  - proteggere dati e risorse dall'interferenza (non autorizzata) di altri processi
  - assicurare che processi ed output siano indipendenti dalla velocità di computazione (ovvero dallo scheduling)
    - "indipendenti" da intendere cum grano salis
    - rispetto alle specifiche di ciascun processo

Interazione tra Processi

Comunicazione	**Relazione**	**Influenza**	**Problemi di Controllo**
Nessuna (ogni processo pensa di essere solo)	Competizione	Risultato di un processo indipendente dagli altri; Tempo di esecuzione di un processo dipendente dagli altri	Mutua esclusione; deadlock; starvation
Memoria condivisa (i processi sanno che c'è qualche altro processo)	Cooperazione	Risultato di un processo dipendente dall'informazione data da altri; Tempo di esecuzione di un processo dipendente dagli altri	Mutua esclusione; deadlock; starvation; coerenza dei dati
Primitive di comunicazione (i processi sanno anche i PID di alcuni altri processi)	Cooperazione	Risultato di un processo dipendente dall'informazione data da altri; Tempo di esecuzione di un processo dipendente dagli altri	Deadlock; starvation

I Processi e la Competizione per le Risorse

- Problema principale per i processi del primo tipo
  - ovvero, ogni processo pensa solo a sé
  - ma per l'accesso alle risorse deve fare una richiesta al sistema operativo (tramite syscall)
- Tre problemi di controllo principali:
  - Necessità di mutua esclusione
    - sezioni critiche
  - Deadlock
  - Starvation

Mutua Esclusione per Processi in Competizione

- Basta che chiamino una syscall che fa tutto lei: entra nella sezione critica, fa l'operazione, esce
- Tuttavia, non è sempre possibile: se occorre accedere ad una risorsa che potrebbe essere condivisa con altri processi, occorre fare una richiesta esplicita di "bloccaggio" della risorsa
- E si ricade nel caso dei processi cooperanti

```
/* PROCESS 1 */

void P1
{
  while (true) {
    /* preceding code /;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

```
/* PROCESS 2 */

void P2
{
  while (true) {
    /* preceding code */;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

. . .

```
/* PROCESS n */

void Pn
{
  while (true) {
    /* preceding code */;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

Mutua Esclusione per Processi Cooperanti

- Gli stessi processi devono essere scritti pensando già alla cooperazione
  - usando opportune syscall (ad es.: quelle sui semafori)
  - devono preoccuparsi di scrivere `entercritical` ed `exitcritical`
  - le specifiche dei processi potrebbero richiedere comportamenti particolari

```
/* PROCESS 1 */

void P1
{
  while (true) {
    /* preceding code /;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

```
/* PROCESS 2 */

void P2
{
  while (true) {
    /* preceding code */;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

. . .

```
/* PROCESS n */

void Pn
{
  while (true) {
    /* preceding code */;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

Deadlock

- Nasce dalla gestione della mutua esclusione
- Esempio di deadlock
  - A richiede accesso prima alla stampante e poi al monitor
  - B il contrario
  - capita che lo scheduler faccia andare B in mezzo alle 2 richieste di A
  - quel tanto che basta per fargli richiedere il monitor
  - le successive richieste (del monitor per A e della stampante per B) non possono essere soddisfatte
  - A e B restano bloccati per sempre
  - eppure, è tutto avvenuto legalmente: la mutua esclusione non è violata

Starvation

- Come il deadlock, è un problema che nasce dalla gestione della mutua esclusione
- Esempio di starvation
  - A richiede accesso prima alla stampante
  - B pure
  - il SO dà la stampante ad A
  - A rilascia la stampante e lo scheduler gli permette di richiederla nuovamente
  - il SO dà nuovamente la stampante ad A
  - e così via per sempre
  - B non "mangia" mai, quindi muore di fame (*starvation*)
  - con 2 soli processi è assai improbabile, ma con tanti processi diventa probabile che uno soffra di starvation

Requisiti per la Mutua Esclusione

Qualsiasi meccanismo si usi per offrire la mutua esclusione, deve soddisfare i seguenti requisiti:

- Solo un processo alla volta può essere nella sezione critica per una risorsa
- Niente deadlock né starvation
- Nessuna assunzione su scheduling dei processi, né sul numero dei processi
- Un processo deve entrare subito nella sezione critica, se nessun altro processo usa la risorsa
- Un processo che si trova nella sua sezione non-critica non deve subire interferenze da altri processi
  - in particolare non può essere bloccato
- Un processo che si trova nella sezione critica ne deve prima o poi uscire
  - più in generale, ci vuole cooperazione
  - ad es., non bisogna scrivere un processo che entra nella sua sezione critica senza chiamare enter_critical

Mutua Esclusione for Dummies

```
int bolt = 0;
void P(int i)
{
  while (true) {
    bolt = 1;
    while (bolt == 1) /* do nothing */;
    /* critical section */;
    bolt = 0;
    /* remainder */;
  }
}

parbegin(P(0), P(1), ..., P(n))
```

Basta che lo scheduler faccia eseguire i 2 processi in interleaving perfetto, ed è deadlock

Oppure basta anche che ci sia un processo solo

Mutua Esclusione for Dummies

```
int bolt = 0;
void P(int i)
{
  while (true) {
    while (bolt == 1) /* do nothing */;
    bolt = 1;
    /* critical section */;
    bolt = 0;
    /* remainder */;
  }
}
```

Basta che lo scheduler faccia eseguire i 2 processi in interleaving perfetto, e si viola la mutua esclusione

Scheduler e Livello Macchina

- Altra corsa critica meno evidente: lo scheduler interrompe *a livello di istruzione macchina*
  - diciamo assembler per visualizzarlo meglio
- Supponiamo che `P(0)` venga eseguito fino a `bolt = 1` compreso
- Si potrebbe pensare che almeno così la mutua esclusione sia sia sicuramente ok
- Invece no! Infatti, `P(1)` potrebbe essere arrivato in precedenza fino a "metà" del `while (bolt == 1);`
  - ovvero, fino a caricare il valore della variabile `bolt` (che, a quel punto, era ancora 0) dentro un registro
- Quando il controllo ritorna a `P(1)`, per lui `bolt` vale 0...

Roadmap

- Concetti basilari di concorrenza
- Mutua esclusione: supporto hardware
- Semafori
- Passaggio di messaggi
- Problema dei lettori/scrittori
- Equivalenze

Disabilitazione delle Interruzioni

```
while (true) {
    /* prima della sezione critica */;
    disabilita_interrupt();
    /* sezione critica */;
    riabilita_interrupt();
    /* rimanente */;
}
```

Disabilitazione delle Interruzioni

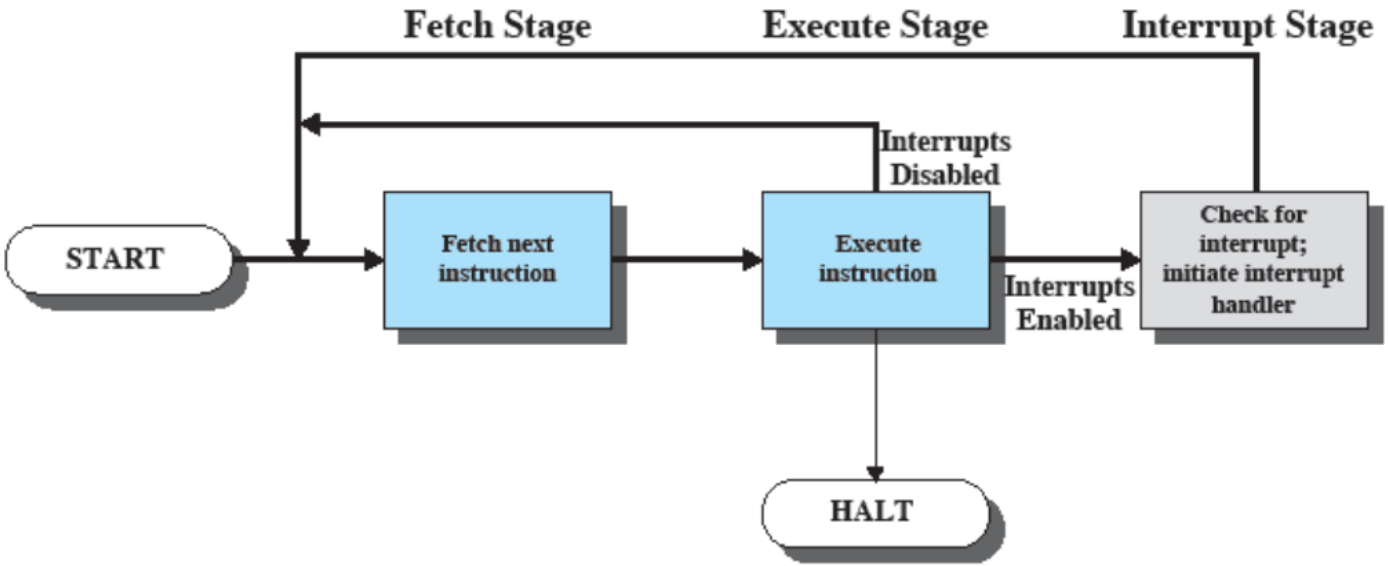


Figure 1.7 Instruction Cycle with Interrupts

Disabilitazione delle Interruzioni

- I sistemi con un solo processore permettono solo l'interleaving
- Un processo viene eseguito finché non invoca il sistema operativo o viene interrotto
- Disabilitare le interruzioni garantisce la mutua esclusione
  - ovvero, se il processo può decidere di non essere interrotto, allora nessun altro lo può interrompere mentre si trova nella sezione critica
- Non funziona su sistemi con più processori
- Se i processi abusano della disabilitazione, peggiorano le prestazioni del SO
  - cala la multiprogrammazione e quindi l'uso del processore

Istruzioni Macchina Speciali

- Istruzione `compare_and_swap` (confronta e scambia)
- Istruzione `exchange`
- Entrambe *atomiche*
  - l'hardware garantisce che un solo processo per volta possa eseguire una chiamata a tali istruzioni/funzioni
  - anche se ci sono più processori

compare_and_swap: Pseudocodice

```
int compare_and_swap (int word, int testval,
                      int newval)
{
    int oldval;
    oldval = word;
    if (word == testval) word = newval;
    return oldval;
}
```

Linguaggio C "finto": l'istruzione `word = newval;` non funzionerebbe in questo modo (passaggio per *valore*)

Mutua Esclusione con compare_and_swap

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true) {
     while (compare_and_swap(bolt, 0, 1) == 1)
          /* do nothing */;
      /* critical section */;
      bolt = 0;
      /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), ... ,P(n));

}
```

exchange: Pseudocodice

```
void exchange (int register, int memory)
{
    int temp;
    temp = memory;
    memory = register;
    register = temp;
}
```

Mutua Esclusione con exchange

```
   /* program mutualexclusion */
int const n = /* number of processes**/;
int bolt;
void P(int i)
{
   int keyi = 1;
   while (true) {
      do exchange (keyi, bolt)
      while (keyi != 0);
      /* critical section */;
      bolt = 0;
      /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), ..., P(n));
}
```

Mutua Esclusione con exchange

```
   /* program mutualexclusion */
int const n = /* number of processes**/;
int bolt;
void P(int i)
{
   int keyi = 1;
   while (true) {
      do exchange (keyi, bolt)
      while (keyi != 0);
      /* critical section */;
      bolt = 0;
      /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), ..., P(n));
}
```

È SBAGLIATO!!!

Mutua Esclusione con Istruzioni Speciali

- In entrambi i casi precedenti l'idea è che il processo che trova `bolt` a 0 può entrare
- E per impedire ad altri di entrare mette `bolt` a 1
  - se un processo si mette in mezzo nel frattempo, la mutua esclusione può essere violata
  - per questo sono istruzioni atomiche
- Una volta finito, rimette `bolt` a 0, così gli altri processi possono entrare
  - o anche lui stesso, se è solo oppure se lo scheduler lo favorisce
- Nel caso della `exchange`, l'errore sta nel fatto che, se un processo fa 2 iterazioni consecutive e rientra nella sezione critica, `bolt` non viene messo a 1
  - quindi altri processi possono entrare, dando luogo ad una race condition

Istruzioni Macchina Speciali: Vantaggi

- Applicabili a qualsiasi numero di processi, sia su un sistema ad un solo processore che ad un sistema a più processori con memoria condivisa
- Semplici e quindi facili da verificare
- Possono essere usate per gestire sezioni critiche multiple

Istruzioni Macchina Speciali: Svantaggi

- Basate sul *busy-waiting*: spreco di tempo di computazione
  - un ciclo di busy wait non è distinguibile da codice "normale"
  - quindi la CPU lo deve eseguire fino al timeout (preemption obbligatoria!)
  - oppure, ci deve essere più di una CPU
- Possibile la starvation (come mostrato sopra)
- Possibile anche il deadlock, se a questi meccanismi viene abbinata la priorità (fissa)
  - se un processo A a bassa priorità viene interrotto mentre è già nella sezione critica...
  - ... e un processo B a priorità alta entra nel busy waiting...
  - B non può essere interrotto per eseguire A a causa della priorità, e A non può andare avanti perché solo B, finendo la sua sezione critica, lo può far uscire dal busy-waiting

Roadmap

- Concetti basilari di concorrenza
- Mutua esclusione: supporto hardware
- Semafori
- Passaggio di messaggi
- Problema dei lettori/scrittori
- Equivalenze

Semafori

- Un valore intero usato dai processi per scambiarsi segnali
- Solo tre operazioni definite, tutte e 3 *atomiche*:
  - `initialize`
  - `decrement` o `semWait`
    - può mettere il processo in blocked: niente CPU sprecata come con il busy waiting
  - `increment` o `semSignal`
    - può mettere un processo blocked in ready
- Si tratta di syscall, quindi sono eseguite in kernel mode e possono agire direttamente sui processi

Semafori: Pseudocodice

```
struct semaphore {
      int count;
      queueType queue;
};
void semWait(semaphore s)
{
      s.count--;
      if (s.count < 0) {
            /* place this process in s.queue */;
            /* block this process */;
      }
}
void semSignal(semaphore s)
{
      s.count++;
      if (s.count <= 0) {
            /* remove a process P from s.queue */;
            /* place process P on ready list */;
      }
}
```

Semafori Binari: Pseudocodice

```
struct binary_semaphore {
      enum {zero, one} value;
      queueType queue;
};
void semWaitB(binary_semaphore s)
{
      if (s.value == one)
            s.value = zero;
      else {
               /* place this process in s.queue */;
               /* block this process */;
      }
}
void semSignalB(semaphore s)
{
      if (s.queue is empty())
            s.value = one;
      else {
               /* remove a process P from s.queue */;
               /* place process P on ready list */;
      }
}
```

Semafori: Pseudocodice "Vero"

```
semWait(s)
{
   while (compare_and_swap(s.flag, 0 , 1) == 1)
      /* do nothing */;
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue*/;
      /* block this process (must also set
s.flag to 0) */;
   }
   s.flag = 0;
}

semSignal(s)
{
   while (compare_and_swap(s.flag, 0 , 1) == 1)
      /* do nothing */;
   s.count++;
   if (s.count <= 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
   }
   s.flag = 0;
}
```

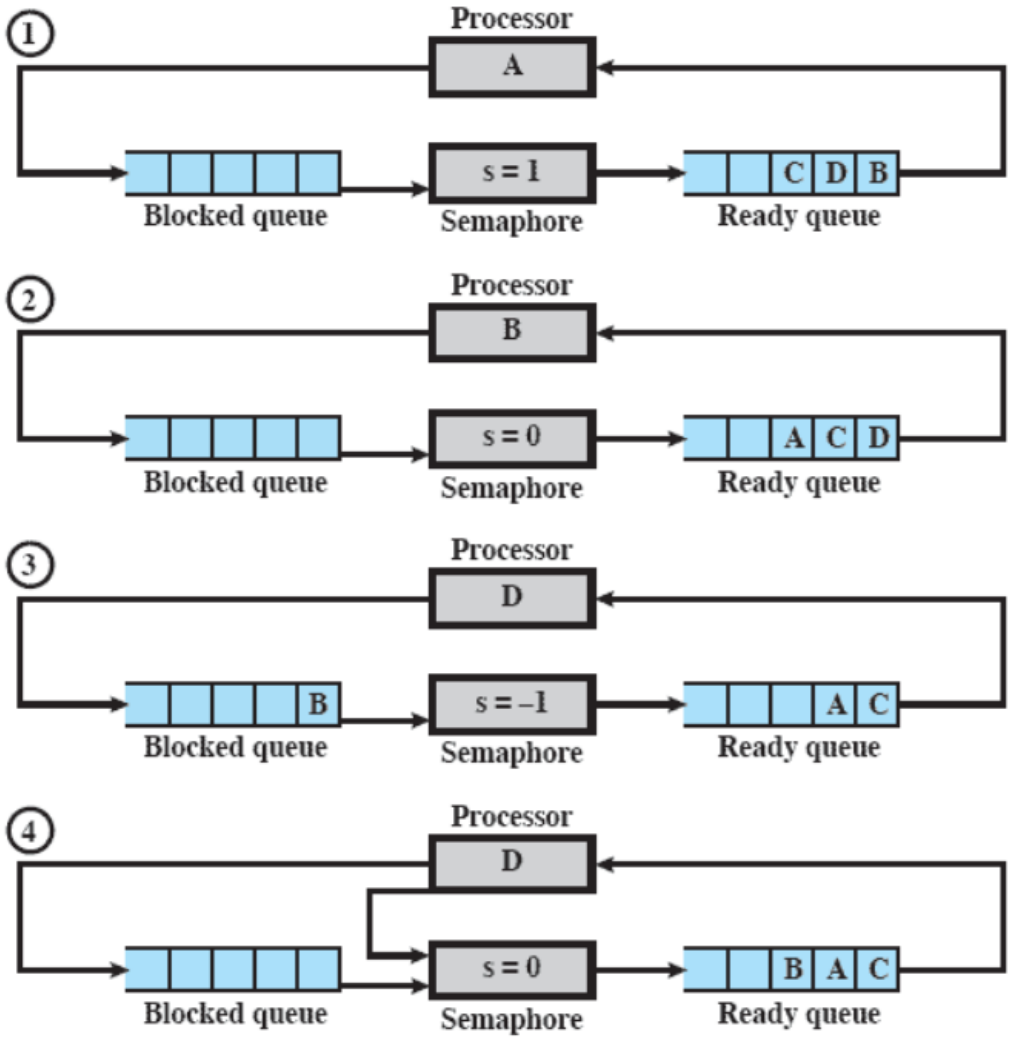
```
semWait(s)
{
   inhibit interrupts;
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue */;
      /* block this process and allow inter-
rupts */;
   }
   else
      allow interrupts;
}

semSignal(s)
{
   inhibit interrupts;
   s.count++;
   if (s.count <= 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
   }
   allow interrupts;
}
```

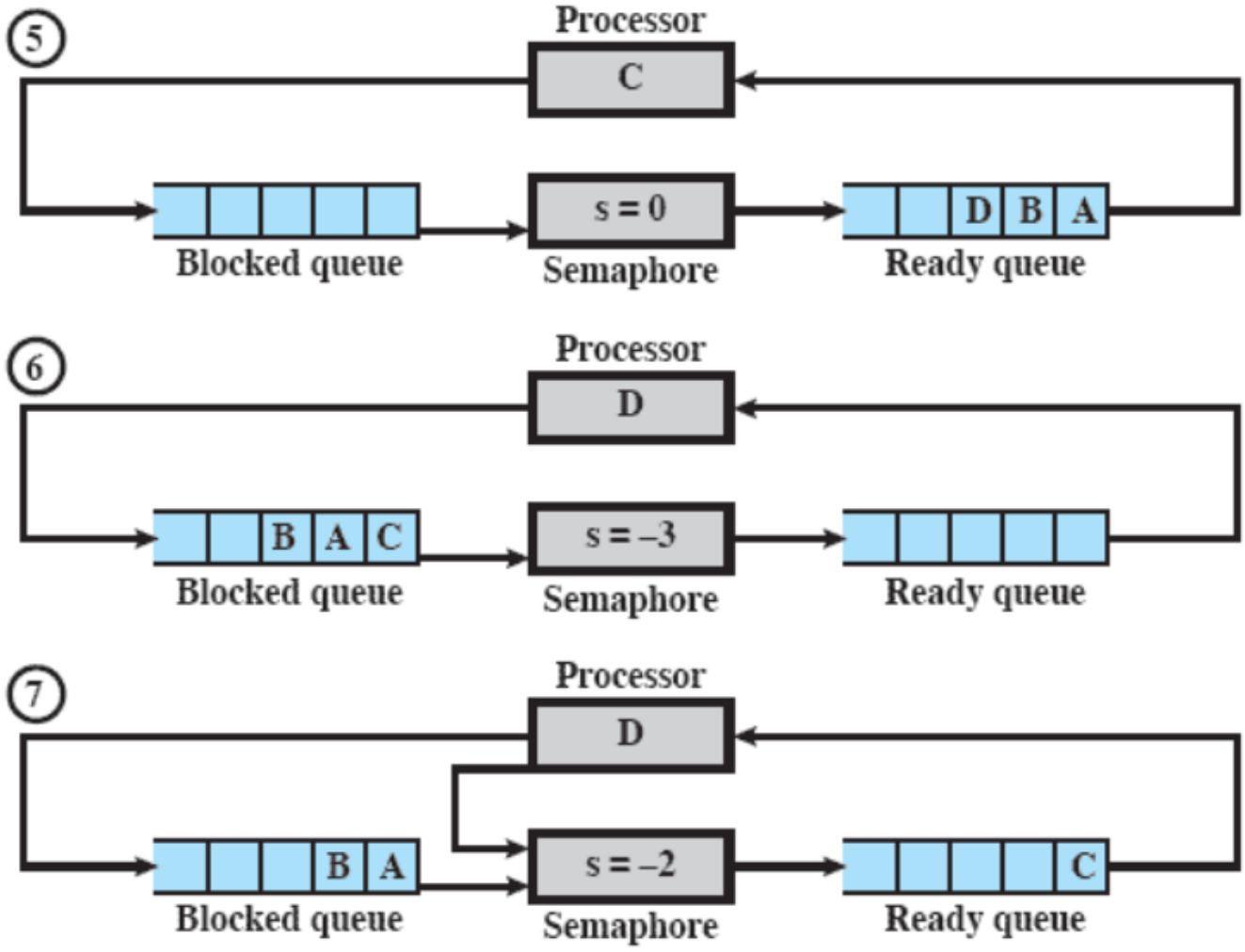
Semafori Deboli e Forti

- Il SO usa una coda per memorizzare i processi in attesa su un semaforo
- Quale politica viene usata per prendere il prossimo processo in attesa nella coda?
- Se è l'"ovvia" FIFO, allora si parla di *strong semaphore* (semaforo forte)
- Se la politica non viene specificata, allora si parla di *weak semaphore* (semaforo debole)
- Con i semafori forti, ovviamente usati bene, si può evitare la starvation; con i deboli no

Semafori Forti: Esempio



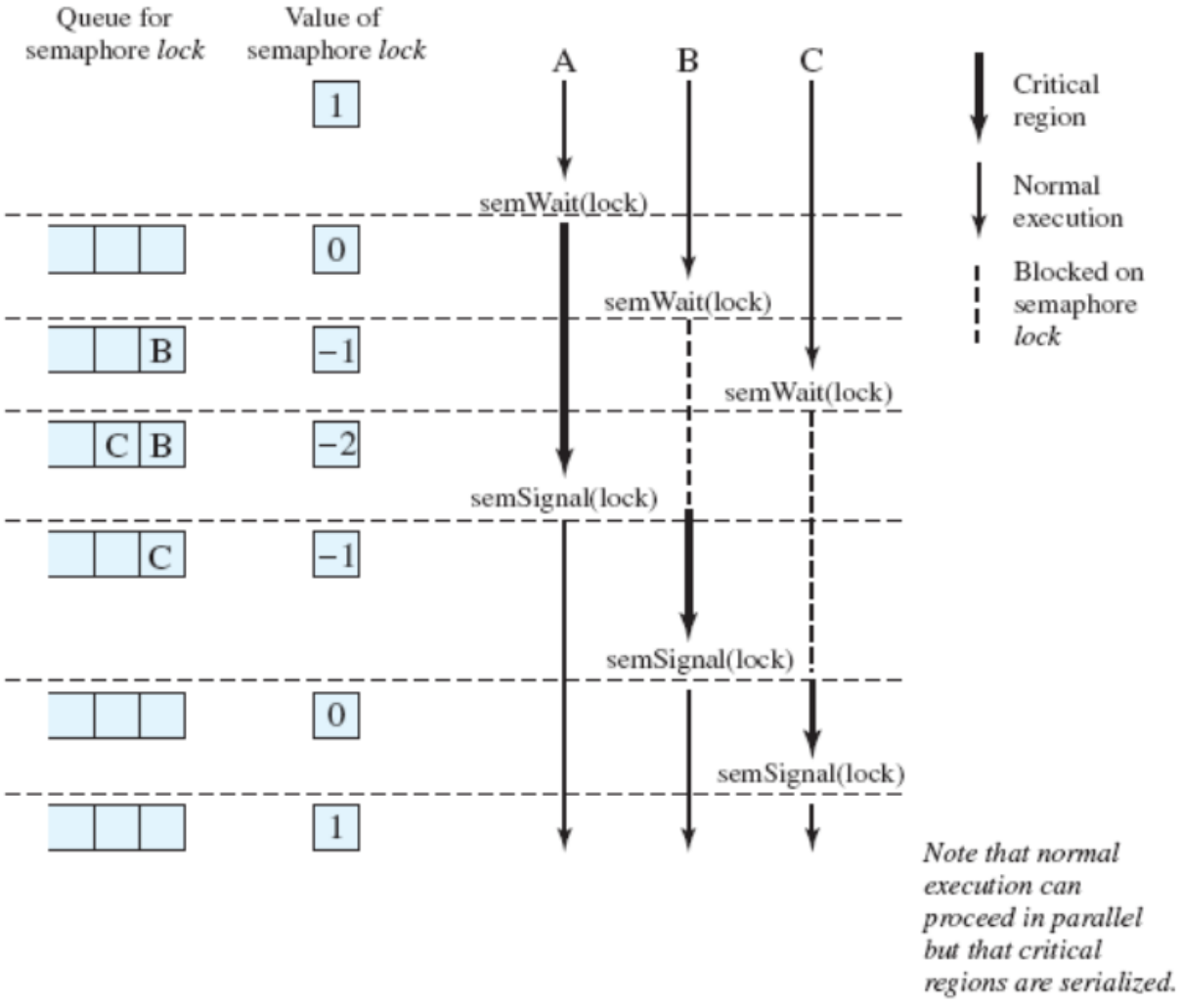
Semafori Forti: Esempio



Mutua Esclusione con i Semafori

```
/* program mutualexclusion */
const int n = /* number of processes  */;
semaphore s = 1;
void P(int i)
{
     while (true) {
          semWait(s);
          /* critical section   */;
          semSignal(s);
          /* remainder   */;
     }
}
void main()
{
     parbegin (P(1), P(2), . . ., P(n));
}
```

Processi e Semafori



Problema del Produttore/Consumatore

- Situazione generale:
  - uno o più (processi) produttori creano dati e li mettono in un buffer
  - un consumatore prende dati dal buffer uno alla volta
  - al buffer può accedere un solo processo, sia esso produttore o consumatore
- Il problema:
  - assicurare che il produttore o i produttori non inseriscano dati quando il buffer è pieno
  - assicurare che il consumatore non prenda dati quando il buffer è vuoto
  - oltre alla mutua esclusione sull'intero buffer
    - in realtà, sarebbe possibile permettere a consumatori e produttori di agire anche in contemporanea
    - per semplicità, non considereremo questo caso

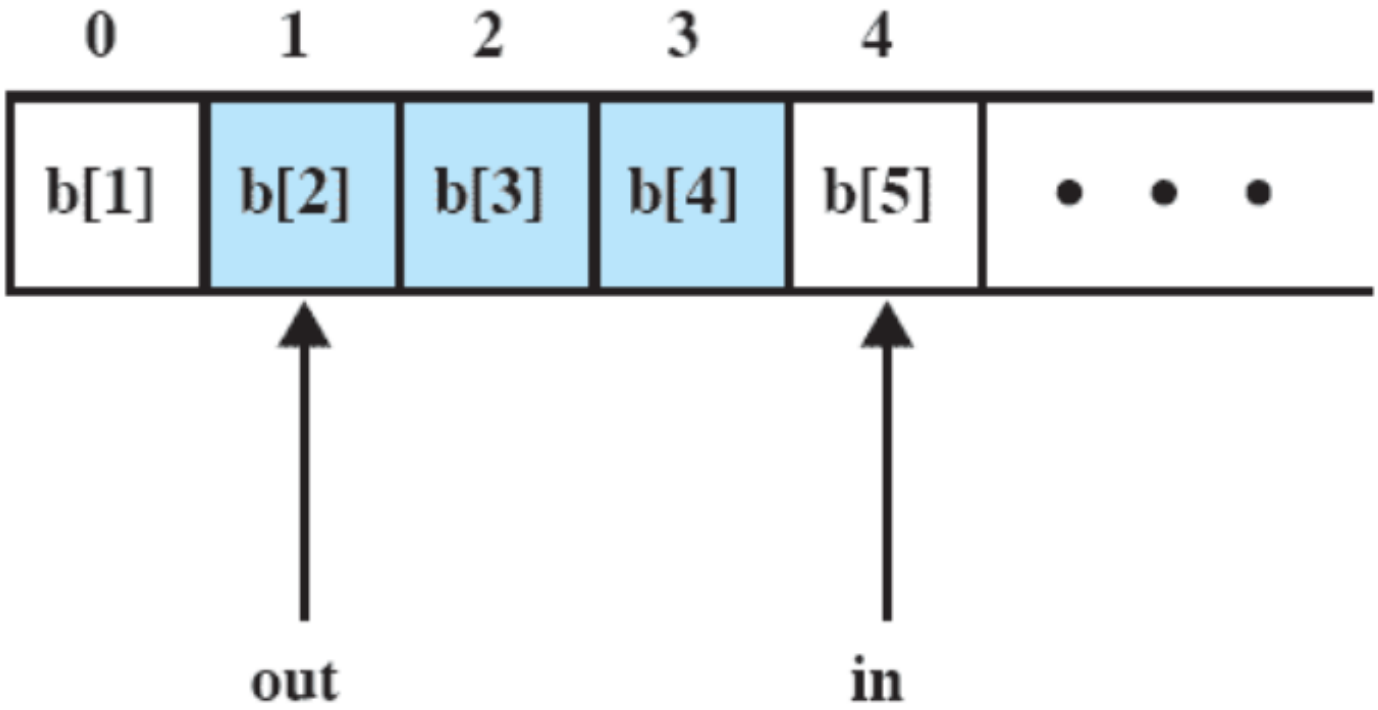
Produttore/Consumatore: Pseudocodici

Per ora, facciamo finta che il buffer sia infinito: il produttore non ha mai motivo di fermarsi

```
while (true) {
    /* produce item v */
    b[in] = v;
    in++;
}
```

```
while (true) {
    while (in <= out)
    /* do  nothing */;
    w = b[out];
    out++;
    /* consume item w */
}
```

Produttori e Consumatori: il Buffer



Produttori e Consumatori: Soluzione Sbagliata

```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
     while (true) {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
void consumer()
{
     semWaitB(delay);
     while (true) {
          semWaitB(s);
          take();
          n--;
          semSignalB(s);
          consume();
          if (n==0) semWaitB(delay);
     }
}
void main()
{
     n = 0;
     parbegin (producer, consumer);
}
```

Soluzione Sbagliata: Possibile Scenario

	Producer	Consumer	s	n	Delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	**if** (n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		semSignalB(s)	1	0	0
10	semWaitB(s)		0	0	0
11	n++		0	1	0
12	**if** (n==1) (semSignalB(delay))		0	1	1
13	semSignalB(s)		1	1	1
14		**if** (n==0) (semWaitB(delay))	1	1	1
15		semWaitB(s)	0	1	1
16		n--	0	0	1
17		semSignalB(s)	1	0	1
18		**if** (n==0) (semWaitB(delay))	1	0	0
19		semWaitB(s)	0	0	0
20		n--	0	–1	0
21		semiSignlaB(s)	1	–1	0

Produttori e Consumatori: Soluzione Corretta

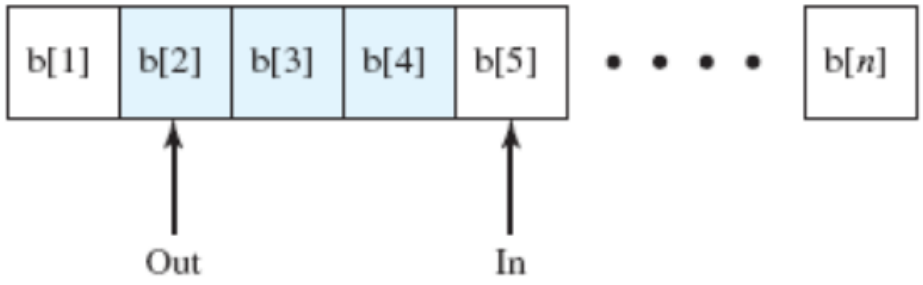
```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
     while (true) {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
void consumer()
{
     int m; /* a local variable */
     semWaitB(delay);
     while (true)  {
          semWaitB(s);
          take();
          n--;
          m = n;
          semSignalB(s);
          consume();
          if (m==0) semWaitB(delay);
     }
}
void main()
{
     n = 0;
     parbegin (producer, consumer);
}
```

Soluzione con Semafori Generali

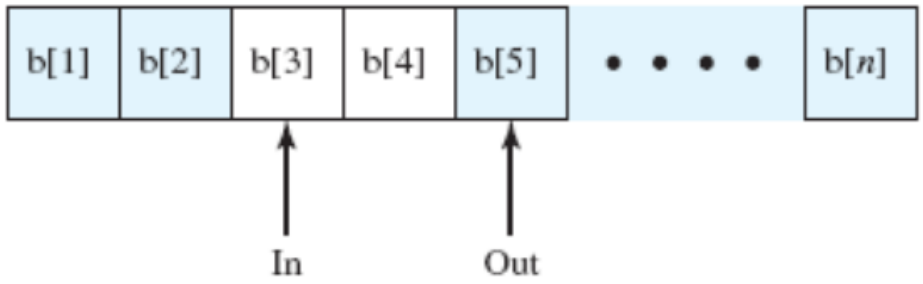
```
/* program  producerconsumer */
semaphore n = 0, s = 1;
void producer()
{
     while (true) {
          produce();
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

Produttori e Consumatori con Buffer Circolare

Block on:	Unblock on:
Producer: insert in full buffer	Consumer: item inserted
Consumer: remove from empty buffer	Producer: item removed



(a)



(b)

Buffer Circolare: Pseudocodici

La dimensione effettiva del buffer è $n - 1$ (altrimenti, non si potrebbe capire se `in == out` implichi buffer pieno o vuoto)

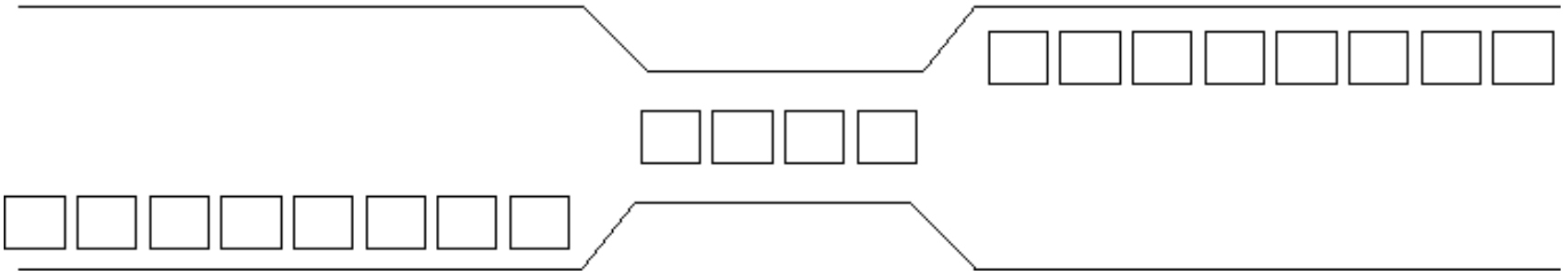
```
while (true) {
  /* produce item v */
  while ((in + 1) % n ==
          out)
    /* do nothing */;
  b[in] = v;
  in = (in + 1) % n
}
```

```
while (true) {
  while (in == out)
    /* do nothing */;
  w = b[out];
  out = (out + 1) % n;
  /* consume item w */
}
```

Produttori e Consumatori con Buffer Circolare

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
void producer()
{
     while (true) {
          produce();
          semWait(e);
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          semSignal(e);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

Trastevere



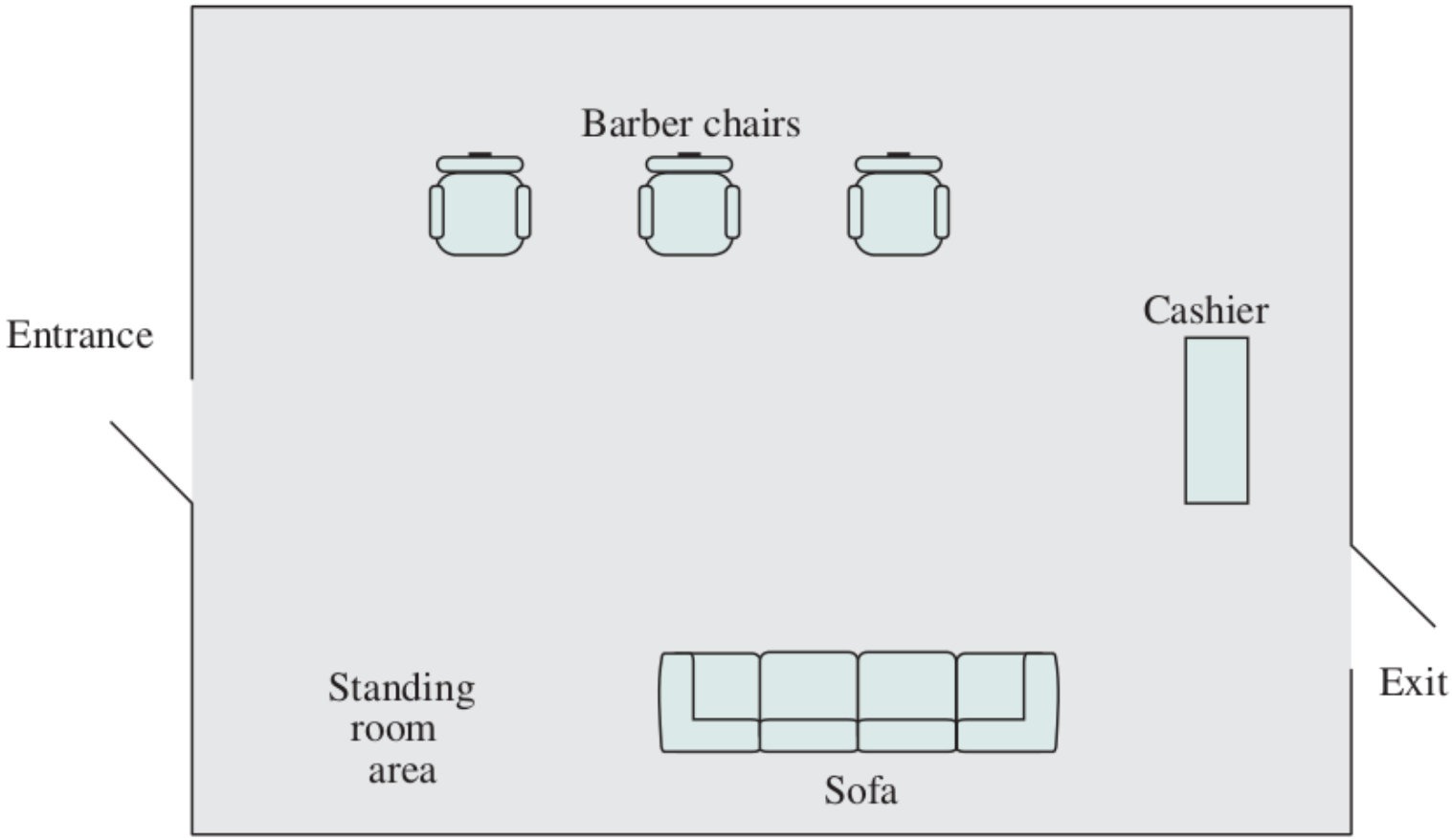
- I blocchetti sono macchine; il tubo è una strada di Trastevere
- La strada si restringe: senso unico alternato, massimo 4 auto per volta
- Vince chi arriva prima, non ci può essere parità
- Assumendo semafori *strong*, le macchine dovrebbero impegnare la strettoia nell'ordine con cui vi arrivano (o si accodano dalla propria parte)

Trastevere

```
semaphore z = 1;
semaphore strettoia = 4;
semaphore sx = 1;
semaphore dx = 1;
int nsx = 0;
int ndx = 0;

macchina_dal_lato_sinistro() {
  wait(z);
  wait(sx);
  ++nsx;
  if (nsx == 1)
    wait(dx);
  signal(sx);
  signal(z);
  wait(strettoia);
  passa_strettoia();
  signal(strettoia);
  wait(sx);
  --nsx;
  if(nsx == 0)
    signal(dx);
  signal(sx);
}
```

Il Negozio del Barbiere



Il Negozio del Barbiere

- Dimensione massima del negozio: `max_cust`
- Prima ci si siede sul divano, poi da lì si accede ad una delle sedie
  - si compete per entrambe le risorse: un certo numero (`sofa`) sul divano, le sedie sono di meno
  - tra tutti quelli nel negozio si compete per il divano, tra tutti quelli sul divano per le sedie
- Il barbiere o dorme, o taglia capelli, o prende soldi dopo il taglio
- Prima soluzione ("equa", nelle slide che seguono):
  - si possono servire, nel corso dell'intero periodo, un massimo numero di clienti
    - dimensione fissa di `finish`
  - c'è una sola cassa, per la quale competono i barbieri
    - ovvero, un barbiere libero a caso può fare le veci del cassiere

Il Negozio del Barbiere

```
semaphore
 max_cust=20,sofa=4,chair=3,
 coord=3,ready=0,leave_ch=0,
 paym=0,recpt=0,finish[50]={0};
 mutex1=1, mutex2=1;
int count = 0;
void customer() {
    int cust_nr;
    wait(max_cust);
    enter_shop();
    wait(mutex1);
    cust_nr = count;
    count++;
    signal(mutex1);
    wait(sofa);
    sit_on_sofa();
    wait(chair);
    get_up_from_sofa();
    signal(sofa);
    sit_in_chair();
    wait(mutex2);
    enqueue1(cust_nr);
    signal(mutex2);
    signal(ready);
    wait(
     finish[cust_nr]);
    leave_chair();
    signal(leave_ch);
    pay();
    signal(paym);
    wait(recpt);
    exit_shop();
    signal(max_cust);
}
```

Il Negozio del Barbiere

```
void barber()
{
  int b_cust;
  while (true) {
    wait(ready);
    wait(mutex2);
    dequeue1(b_cust);
    signal(mutex2);
    wait(coord);
    cut_hair();
    signal(coord);
    signal(
      finish[b_cust]);
    wait(leave_ch);
    signal(chair);
  }
}
```

```
void cashier()
{
  while (true)  {
    wait(paym);
    wait(coord);
    accept_pay();
    signal(coord);
    signal(recpt);
  }
}
```

Una Soluzione Migliore

- Niente limite massimo al numero di clienti servibili in un giorno
- Niente processo separato per pagare, ma resta il fatto che si paga un barbiere qualsiasi (purché libero)
- Un solo semaforo `mutex`
- Il semaforo `coord` non serve più
- Ci sono tanti semafori `finish` quanti sono i *barbieri*
- Niente semaforo `leave_ch` ()
  - che tanto era anche inefficiente: solo un cliente alla volta poteva alzarsi, anche su sedie diverse
- Il semaforo `chair` è inizializzato a 0: viene incrementato ogni volta che un barbiere torna libero...
- Piccola inefficienza: solo un barbiere alla volta può preparare la *propria* sedia

Una Soluzione Migliore

```
int next_barber;
void Barber(i) {
  while (true) {
    wait(mutex);
    next_barber = i;
    signal(chair);
    wait(ready);
    signal(mutex);
    cut_hair();
    signal(finish[i]);
    wait(paym);
    accept_pay();
    signal(recpt);
  }
}
```

Una Soluzione Migliore

```
void Customer() { int my_barber;
  wait(max_cust);
  enter_shop();
  wait(sofa);
  sit_on_sofa();
  wait(chair);
  get_up_from_sofa();
  signal(sofa);
  my_barber = next_barber;
  sit_in_chair();
  signal(ready);
  wait(finish[my_barber]);
  leave_chair();
  pay();
  signal(paym);
  wait(recpt);
  exit_shop();
  signal(max_cust); }
```

Roadmap

- Concetti basilari di concorrenza
- Mutua esclusione: supporto hardware
- Semafori
- Mutua esclusione: soluzioni software
- Passaggio di messaggi
- Problema dei lettori/scrittori
- Equivalenze

Primo Tentativo

```
   /* PROCESS 0 /*                    /* PROCESS 1 */
.                                  .
.                                  .
while (turn != 0)                  while (turn != 1)
   /* do nothing */ ;                 /* do nothing */;
/* critical section*/;              /* critical section*/;
turn = 1;                          turn = 0;
.                                  .
```

Secondo Tentativo

```
   /* PROCESS 0 */
.
.
while (flag[1])
   /* do nothing */;
flag[0] = true;
/*critical section*/;
flag[0] = false;
.
```

```
   /* PROCESS 1 */
.
.
while (flag[0])
   /* do nothing */;
flag[1] = true;
/* critical section*/;
flag[1] = false;
.
```

Terzo Tentativo

```
   /* PROCESS 0 */                  /* PROCESS 1 */
.                                .
.                                .
flag[0] = true;                  flag[1] = true;
while (flag[1])                  while (flag[0])
   /* do nothing */;                 /* do nothing */;
/* critical section*/;            /* critical section*/;
flag[0] = false;                 flag[1] = false;
.                                .
```

Quarto Tentativo

```
   /* PROCESS 0 */                  /* PROCESS 1 */
.                                   .
.                                   .
flag[0] = true;                     flag[1] = true;
while (flag[1]) {                   while (flag[0]) {
   flag[0] = false;                    flag[1] = false;
   /*delay */;                         /*delay */;
   flag[0] = true;                     flag[1] = true;
}                                   }
/*critical section*/;               /* critical section*/;
flag[0] = false;                    flag[1] = false;
.
```

Livelock

P0 sets `flag[0]` to `true`.

P1 sets `flag[1]` to `true`.

P0 checks `flag[1]`.

P1 checks `flag[0]`.

P0 sets `flag[0]` to `false`.

P1 sets `flag[1]` to `false`.

P0 sets `flag[0]` to `true`.

P1 sets `flag[1]` to `true`.

Algoritmo di Dekker

```
p0:
   wants_to_enter[0] ← true
   while wants_to_enter[1] {
      if turn ≠ 0 {
         wants_to_enter[0] ← false
         while turn ≠ 0 {
           // busy wait
         }
         wants_to_enter[0] ← true
      }
   }

   // critical section
   ...
   turn ← 1
   wants_to_enter[0] ← false
   // remainder section
```

```
p1:
   wants_to_enter[1] ← true
   while wants_to_enter[0] {
      if turn ≠ 1 {
         wants_to_enter[1] ← false
         while turn ≠ 1 {
           // busy wait
         }
         wants_to_enter[1] ← true
      }
   }

   // critical section
   ...
   turn ← 0
   wants_to_enter[1] ← false
   // remainder section
```

Algoritmo di Dekker

- Vale solo per 2 processi; estensione a N processi possibile ma non banale
  - che valore dare a `turn` dopo la sezione critica?
- Flag inizializzati a 0; `turn` può essere inzializzato a 0 o ad 1
- Garantisce la non-starvation (grazie a `turn`)
- Garantisce il non-deadlock
  - ma è busy-waiting: se ci sono le priorità fisse, ci può essere deadlock
- Non richiede nessun supporto dal SO
  - alcune moderne architetture hanno ottimizzazioni hardware che riordinano le istruzione da eseguire, nonché gli accessi in memoria
  - è necessario disabilitare tali ottimizzazioni

Algoritmo di Peterson

Peterson's Algorithm

```
boolean flag [2];
int turn;
void P0()
{
      while (true) {
            flag [0] = true;
            turn = 1;
            while (flag [1] && turn == 1) /* do nothing */;
            /* critical section */;
            flag [0] = false;
            /* remainder */;
      }
}
void P1()
{
      while (true) {
            flag [1] = true;
            turn = 0;
            while (flag [0] && turn == 0) /* do nothing */;
            /* critical section */;
            flag [1] = false;
            /* remainder */
      }
}
void main()
{
      flag [0] = false;
      flag [1] = false;
      parbegin (P0, P1);
}
```

Algoritmo di Peterson

- Vale solo per 2 processi; estensione a N processi più facile che per Dekker
- Inizializzazioni non necessarie
- Come Dekker per starvation, deadlock e problemi con le CPU che riordinano gli accessi in memoria
- Anche il bounded-waiting: un processo può aspettare l'altro al più una volta
  - vale anche per Dekker
  - non vale per la generalizzazione ad $N > 2$ processi

Roadmap

- Concetti basilari di concorrenza
- Mutua esclusione: supporto hardware
- Semafori
- Mutua esclusione: soluzioni software
- Passaggio di messaggi
- Problema dei lettori/scrittori
- Equivalenze

Interazione tra Processi

- Quando un processo interagisce con un altro, due requisiti fondamentali devono essere soddisfatti:
  - sincronizzazione (mutua esclusione)
  - comunicazione
- Lo scambio di messaggi (*message passing*) è una soluzione al secondo requisito
  - funziona sia con memoria condivisa che distribuita
  - può essere usata anche per la sincronizzazione
- Funzionalità fornita tramite due primitive
  - `send(destination, message)`
  - `receive(source, message)`
  - spesso c'è anche il test di ricezione
  - notare che `message` è un input per la `send` ed un output per la `receive`

Sincronizzazione

- La comunicazione richiede anche la sincronizzazione
  - il mittente deve inviare prima che il ricevente riceva
- Cosa succede dopo che un processo ha effettuato un invio od una ricezione?
  - queste operazioni possono essere bloccanti oppure no
  - il test di ricezione non è mai bloccante

Send e Receive Bloccanti

- Sia mittente che destinatario sono bloccati finché il messaggio non è completamente ricevuto
- Tipicamente viene chiamato *rendezvous*
- Sincronizzazione molto stretta

Send Non Bloccante

- Più naturale per molti programmi concorrenti
- La indicheremo come `nbsend`
- Con ricezione bloccante (caso molto comune):
  - il mittente continua (nel senso: non viene messo in blocked)
  - il destinatario rimane bloccato finché non ha ricevuto il messaggio
- Con ricezione non bloccante: se il messaggio c'è viene ricevuto, altrimenti si va avanti
- La indicheremo come `nbreceive`
  - può settare un bit dentro il messaggio per dire se la ricezione è avvenuta oppure no
  - se la ricezione è non bloccante, allora tipicamente non lo è neanche l'invio
- Sempre *atomiche*
  - un solo processo per volta le esegue
  - finché l'operazione non è stata completata, oppure il processo non viene bloccato

Indirizzamento

- Il mittente deve poter dire a quale processo (o quali processi) vuole mandare il messaggio
- Lo stesso per il destinatario, anche se non sempre
- Si possono usare:
  - indirizzamento diretto
  - indirizzamento indiretto

Indirizzamento Diretto

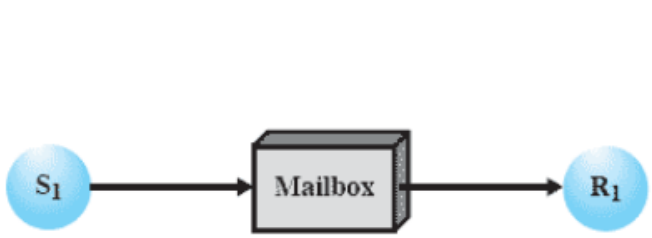
- La primitiva di `send` include uno specifico identificatore per il destinatario
  - o per un gruppo di destinatari
- Per la `receive`, ci può essere oppure no
  - `receive(sender, msg)`: ricevi solo se il mittente coincide con `sender`
    - utile per applicazioni fortemente cooperative
  - `receive(null, msg)`: ricevi da chiunque
    - dentro `msg`, come vedremo, c'è anche il mittente
    - es.: il processo che gestisce le stampe può accettare messaggi da tutti
- Ogni processo ha una sua coda; una volta piena, solitamente il messaggio si perde o viene ritrasmesso
  - es.: syscall `listen` di Linux

Indirizzamento Indiretto

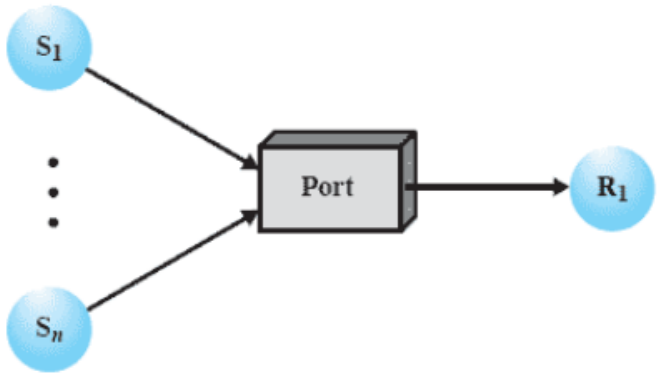
- I messaggi sono inviati ad una casella di posta (*mailbox*) *condivisa*
  - va esplicitamente creata da un qualche processo
- Il mittente manda messaggi alla mailbox, da dove il destinatario se li va a prendere
- Se la ricezione è bloccante, e ci sono più processi in attesa su una ricezione, un solo processo viene svegliato
- Evidenti analogie con il produttore/consumatore
  - in particolare, se la mailbox è piena allora anche `nbsend` si deve bloccare
  - se serve solo per le versioni bloccanti e per comunicazioni x-to-one, la mailbox può avere dimensione 1

Comunicazione Indiretta

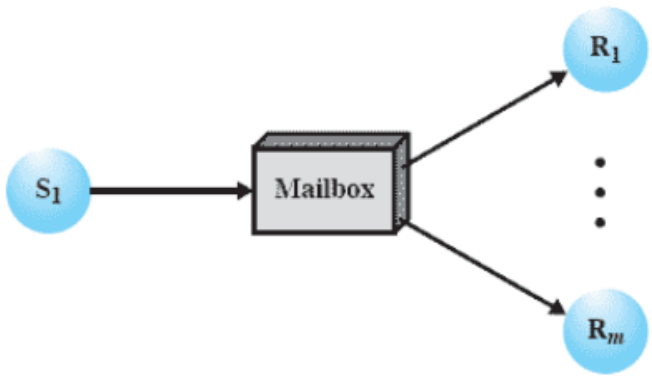
La prima riga, in realtà, è la comunicazione diretta



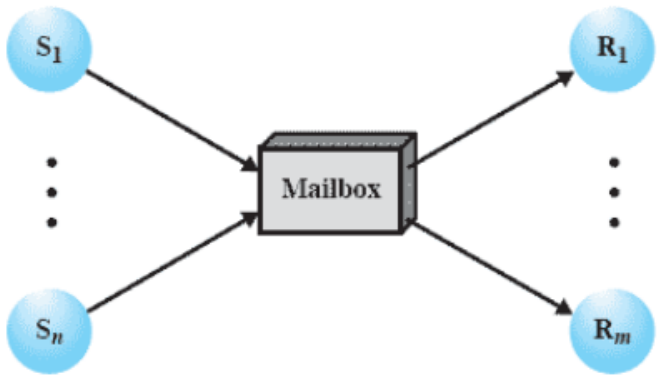
(a) One to one



(b) Many to one

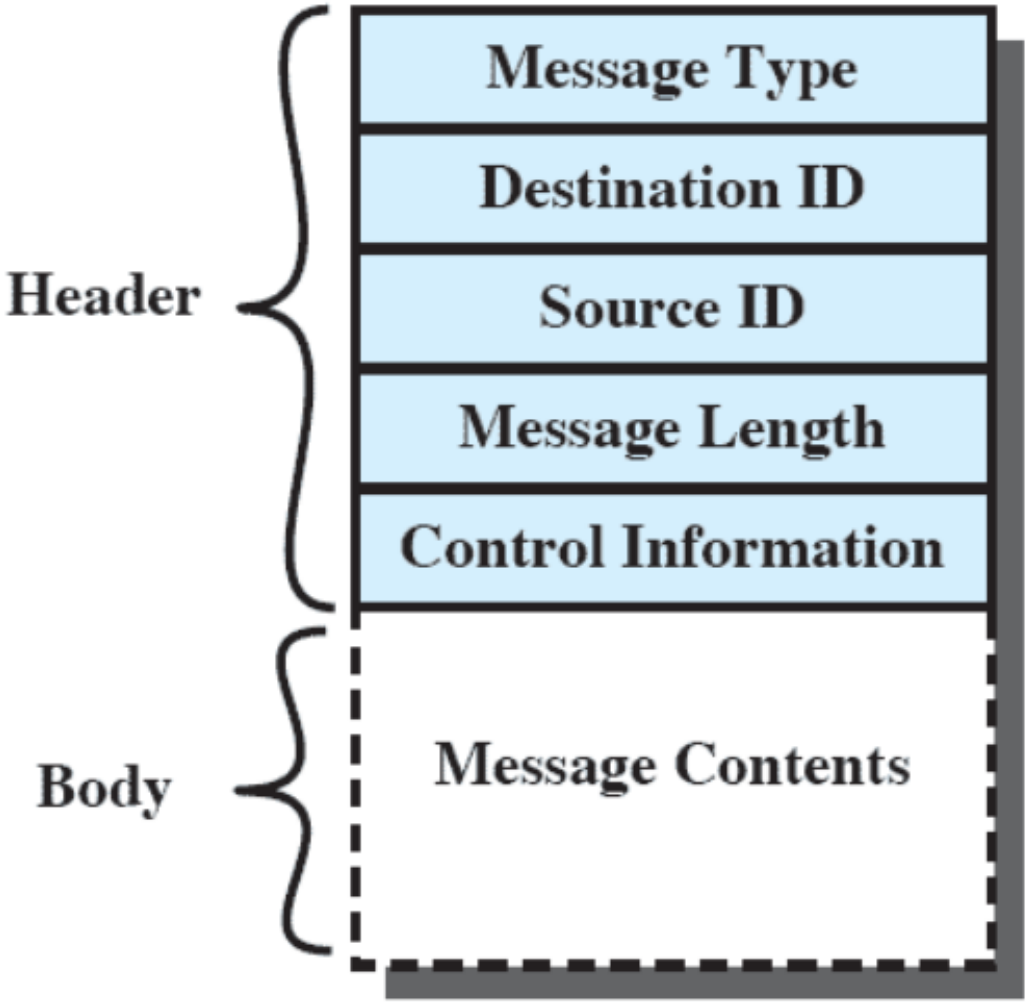


(c) One to many



(d) Many to many

Tipico Formato dei Messaggi



Mutua Esclusione con i Messaggi

```
const message null = /* null message */;
mailbox box;
void P(int i) {
  message msg;
  while (true) {
    receive(box, msg);
    /* critical section */;
    nbsend(box, msg);
    /* remainder */;
} }

void main() {
  box = create_mailbox();
  nbsend(box, null);
  parbegin (P(1), P(2), . . ., P(n));
}
```

Produttore/Consumatore con i Messaggi

```
const int capacity = /* buffering capacity */ ;
mailbox mayproduce, mayconsume;
const message null = /* null message */;

void main()
{
  mayproduce = create_mailbox(),
  mayconsume = create_mailbox();
  for (int i = 1; i <= capacity; i++)
    nbsend (mayproduce,null);
  parbegin (producer,consumer);
}
```

Produttore/Consumatore con i Messaggi

```
void producer() {
  message pmsg;
  while (true) {
    receive (mayproduce,pmsg);
    pmsg = produce();
    nbsend (mayconsume,pmsg);
} }

void consumer() {
  message cmsg;
  while (true) {
    receive (mayconsume,cmsg);
    consume (cmsg);
    nbsend (mayproduce,null);
} }
```

Soluzioni con Messaggi

- Mutua esclusione: ok
- Deadlock: ok
- Starvation: ok solo se le code di processi bloccati su una receive sono gestite in modo "forte"
  - ovvero, i processi si sbloccano secondo un ordine FIFO
  - nel caso di 1 produttore ed un consumatore, ovviamente, starvation non c'è in nessun caso...
- Se ci sono più di 1 produttore ed 1 consumatore, la soluzione appena vista funziona ugualmente

Roadmap

- Concetti basilari di concorrenza
- Mutua esclusione: supporto hardware
- Semafori
- Mutua esclusione: soluzioni software
- Passaggio di messaggi
- Problema dei lettori/scrittori
- Equivalenze

Problema dei Lettori/Scrittori

- Un'area dati è condivisa tra molti processi
- Alcuni la leggono, altri la scrivono
- Condizioni da soddisfare:
  - più lettori possono leggere l'area contemporaneamente
  - solo uno scrittore può scrivere nell'area
  - se uno scrittore è all'opera sull'area, nessun lettore può effettuare letture
- Differenza con i produttori/consumatori: l'area condivisa si accede *per intero*
  - niente problemi di buffer pieno o vuoto
  - ma è importante permettere ai lettori di accedere contemporaneamente

Soluzione Con Precedenza ai Lettori

```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true) {
      semWait (x);
      readcount++;
      if (readcount == 1) semWait (wsem);
      semSignal (x);
      READUNIT();
      semWait (x);
      readcount--;
      if (readcount == 0) semSignal (wsem);
      semSignal (x);
    }
 }
void writer()
{
    while (true) {
      semWait (wsem);
      WRITEUNIT();
      semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader, writer);
}
```

Soluzione Con Precedenza agli Scrittori

```
/*program readersandwriters*/
int  readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true) {
     semWait (z);
          semWait (rsem);
               semWait (x);
                    readcount++;
                    if (readcount == 1) semWait (wsem);
               semSignal (x);
          semSignal (rsem);
     semSignal (z);
     READUNIT();
     semWait (x);
          readcount--;
          if (readcount == 0) semSignal (wsem);
     semSignal (x);
    }
}
```

Soluzione Con Precedenza agli Scrittori

```
void writer ()
{
    while (true) {
      semWait (y);
           writecount++;
           if (writecount == 1) semWait (rsem);
      semSignal (y);
      semWait (wsem);
      WRITEUNIT();
      semSignal (wsem);
      semWait (y);
           writecount--;
           if (writecount == 0) semSignal (rsem);
      semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

Soluzione Con Precedenza agli Scrittori

- Cosa potrebbe succedere se si togliesse il semaforo `x` nell'incremento di `readcount`?
  - dopotutto, si è già sicuri che lì c'è un solo processo lettore...
- Cosa potrebbe succedere se si togliesse `z`?
- Cosa potrebbe succedere se si usasse `x` al posto di `z`?
- Ma il `parbegin` messo così ha effettivamente senso?

Soluzione Con i Messaggi

```
void reader(int i)
{
  while (true) {
    nbsend (readrequest, null);
    receive (controller_pid, null);
    READUNIT ();
    nbsend (finished, null);
} }

void writer(int j)
{
  while (true) {
    nbsend (writerequest, null);
    receive (controller_pid, null);
    WRITEUNIT ();
    nbsend (finished, null);
} }
```

Soluzione Con i Messaggi

```
void controller() {
  int count = MAX_READERS;
  while (true) {
    if (count > 0) {
      if (!empty (finished)) { /* da reader! */
        receive (finished, msg);
        count++;
      }
      else if (!empty (writerequest)) {
        receive (writerequest, msg);
        writer_id = msg.sender;
        count = count - MAX_READERS;
      }
      else if (!empty (readrequest)) {
        receive (readrequest, msg);
        count--;
        nbsend (msg.sender, "OK");
    }  }
```

Soluzione Con i Messaggi

```
      if (count == 0) {
        nbsend (writer_id, "OK");
        receive (finished, msg); /* da writer! */
        count = MAX_READERS;
      }
      while (count < 0) {
        receive (finished, msg); /* da reader! */
        count++;
      } /* while (count < 0) */
    } /* while (true) */
  } /* controller */
```

Soluzione Con i Messaggi

- Ovviamente, occorre un processo di inizializzazione
  - crea le 3 mailbox
  - lancia 1 controller più reader e writer a piacimento
- Se ci sono più di `MAX_READERS-1` richieste *contemporanee* da lettori, la soluzione non funziona
- Se `count > 0`, cosa è sempre vero?
- Se `count == 0`, cosa è sempre vero?
- Se `count < 0`, cosa è sempre vero?

Roadmap

- Concetti basilari di concorrenza
- Mutua esclusione: supporto hardware
- Semafori
- Mutua esclusione: soluzioni software
- Passaggio di messaggi
- Problema dei lettori/scrittori
- Equivalenze

Equivalenze

- La condivisione di risorse può quindi essere implementata con 3 metodi diversi
  - istruzioni hardware
  - sincronizzazione (semafori)
  - message passing
- Se si può implementare un'applicazione con uno qualsiasi dei 3 metodi, allora lo si può fare anche con gli altri 2
  - un particolare meccanismo potrà rivelarsi più conveniente degli altri, in termini di facilità di sviluppo, di prestazioni, e di gestione