Mutua esclusione - 2

Arturo Carpi

Dipartimento di Matematica e Informatica
Università di Perugia

Corso di Sistemi Operativi - a.a. 2021/22

Algoritmo di Dekker

Obiettivo

Un protocollo che permetta a due processi di assicurare la mutua esclusione

- senza supporto hardware (salvo impossibilità di accesso simultaneo al medesimo indirizzo di memoria)
- senza stallo
- senza starvation

Primo tentativo

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

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

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

Quarto tentativo

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

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

Soluzione corretta

```
boolean flag [2];
int turn;

void P0()                                          void P1()
{                                                  {
   while (true) {                                    ...
      flag [0] = true;
      while (flag [1]) {
         if (turn == 1) {
            flag [0] = false;
            while (turn == 1) /* do nothing */;
            flag [0] = true;
         }
      }
      /* critical section */;
      turn = 1;
      flag [0] = false;
      /* remainder */;
   }
}
```

Algoritmo di Peterson

```
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() { . . . }
```

Semafori

Semaforo generico

Una variabile con valore intero, una coda e tre operazioni

- inizializzazione a un valore non negativo
- semWait decrementa il valore del semaforo.
  Se il valore diventa negativo, il processo è sospeso e inserito nella coda
- semSignal incrementa il valore del semaforo.
  Se il valore non diventa positivo, uno dei processi della coda è riattivato

Osservazione

Il processo non ha modo di sapere in anticipo

- se con l'esecuzione di semWait sarà sospeso
- se c'è un processo che attende l'esecuzione di semSignal
- quale fra i processi in attesa sarà riattivato da semSignal

Semaforo generico

```
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 - 2

Semaforo binario

Una variabile con valore binario (0 o 1), una coda e tre operazioni

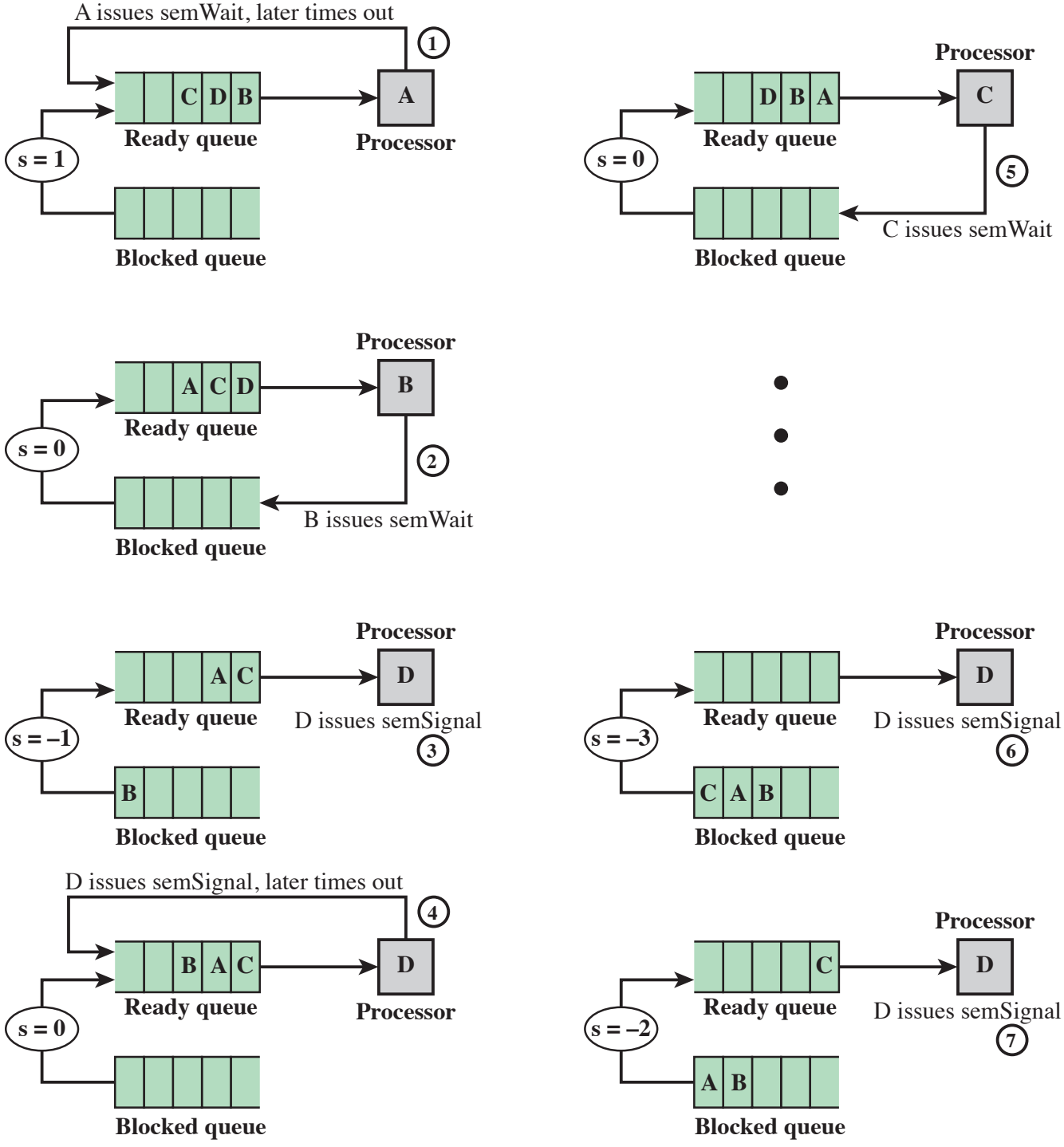
- inizializzazione a zero o uno
- semWaitB verifica il valore del semaforo
  Se il valore è 1 allora è portato a 0, nel caso contrario, il processo è sospeso e inserito nella coda,
- semSignalB verifica se ci sono processi in attesa nella coda.
  In caso affermativo, uno dei processi della coda è riattivato, nel caso opposto, il valore del semaforo è portato a 1.

Gestione della coda

- Deve evitare la starvation
- Generalmente FIFO (semaforo forte)

Semaforo binario

```
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 */;
   }
}
```

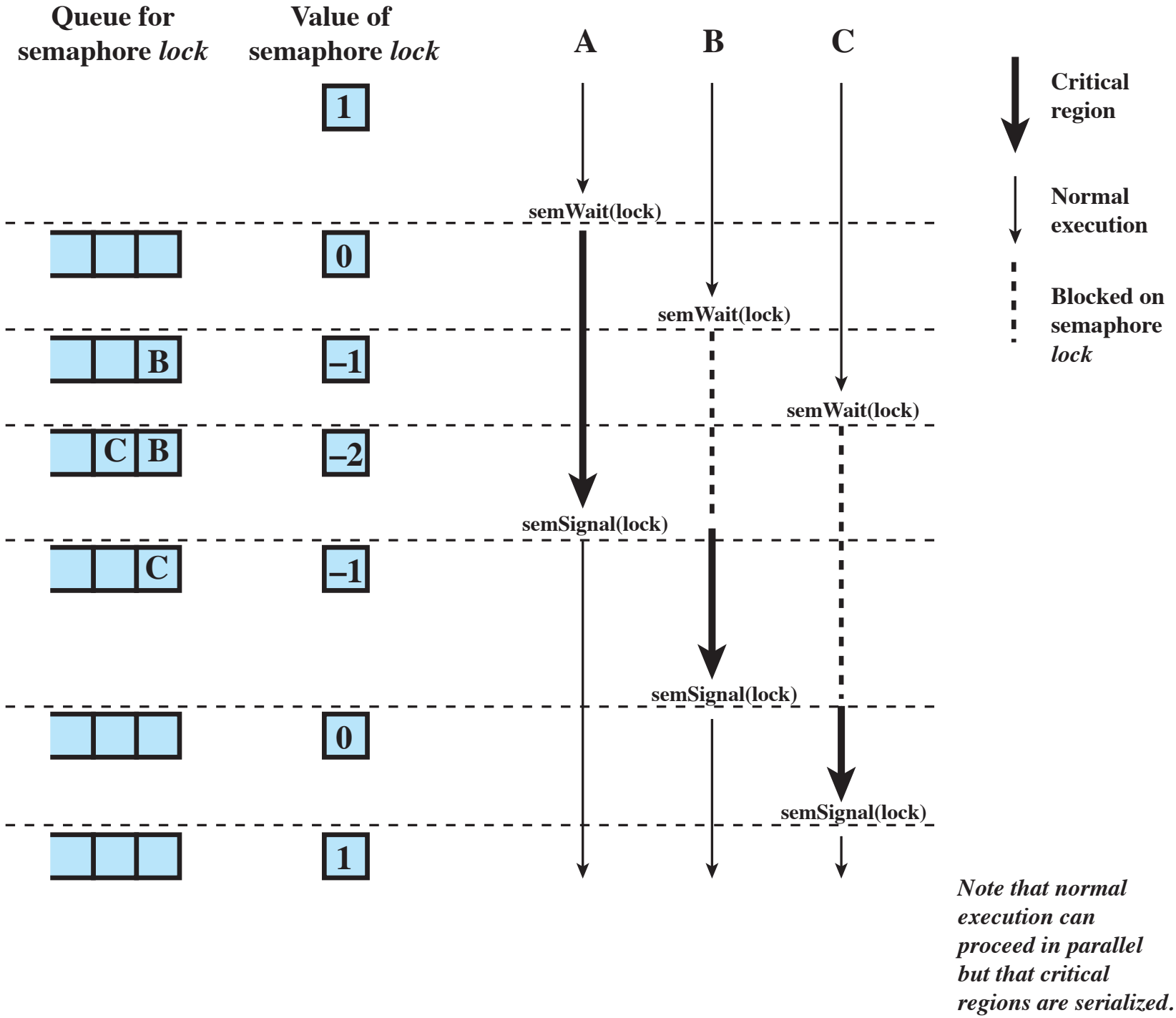

A issues semWait, later times out
1
C D B
A
Ready queue
Processor
s = 1
Blocked queue
Processor
D B A
C
Ready queue
s = 0
5
C issues semWait
Blocked queue
Processor
A C D
B
Ready queue
s = 0
2
B issues semWait
Blocked queue
A C
D
Ready queue
D issues semSignal
3
s = –1
B
Blocked queue
Processor
A C
D
Ready queue
D issues semSignal
6
s = –3
C A B
Blocked queue
D issues semSignal, later times out
4
B A C
D
Ready queue
Processor
s = 0
Blocked queue
C
D
Ready queue
D issues semSignal
7
s = –2
A B
Blocked queue

Mutua esclusione

```
/* 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));
}
```


Queue for semaphore lock
Value of semaphore lock
A
B
C
1
semWait(lock)
0
semWait(lock)
B
–1
semWait(lock)
C
B
–2
semSignal(lock)
C
–1
semSignal(lock)
0
semSignal(lock)
1
Critical region
Normal execution
Blocked on semaphore lock
Note that normal execution can proceed in parallel but that critical regions are serialized.

Implementazione

- semWait e semSignal devono essere primitive atomiche
  - hardware o firmware
  - con istruzioni speciali
  - con l'algoritmo di Peterson

Messaggi

Nell'interazione fra processi sono necessarie

Sincronizzazione

per realizzare la mutua esclusione

Comunicazione

scambio di informazioni

Scambio di messaggi

Permette di realizzare entrambe

- naturale nei sistemi distribuiti
- utile anche nei sistemi con memoria condivisa

Due primitive

- send (destination, message)
  - un processo invia informazioni in forma di messaggio a un altro processo designato come destinatario
- receive (source, message)
  - un processo riceve informazioni indicando il mittente e l'indirizzo in cui salvare il messaggio

Caratteristiche dei sistemi di messaggistica

Sincronizzazione

- Send
  - bloccante
  - non bloccante
- Receive
  - bloccante
  - non bloccante
  - test presenza messaggi

Indirizzamento

- Diretto
  - send
  - receive
    - esplicito
    - implicito
- Indiretto
  - statico
  - dinamico

Formato

- Contenuto
- Lunghezza
  - fissa
  - variabile

Organizzazione delle code

- FIFO
- Priorità

Sincronizzazione

- Il destinatario non può ricevere il messaggio prima che sia stato inviato
- Quando esegue la Receive:
  - Se è presente un messaggio, il processo lo riceve e continua l'esecuzione
  - In caso contrario, due possibilità:
    - si sospende finchè il messaggio non è inviato
    - continua l'esecuzione senza più ricevere il messaggio

Send bloccante, Receive bloccante

- Mittente e destinatario bloccati fino al completamento dell'operazione
- Rendez-vous
- permette la sincronizzazione dei processi

Send non bloccante, Receive bloccante

- il mittente procede ma il destinatario e bloccato fino alla ricezione del messaggio
- combinazione più comune
- il mittente invia più messaggi a tanti destinatari senza perdita di tempo.
  - Esempio: processo server che fornisce servizi a più processi

Send non bloccante, Receive non bloccante

- Nessuno dei due processi attende

Indirizzamento

Indirizzamento diretto

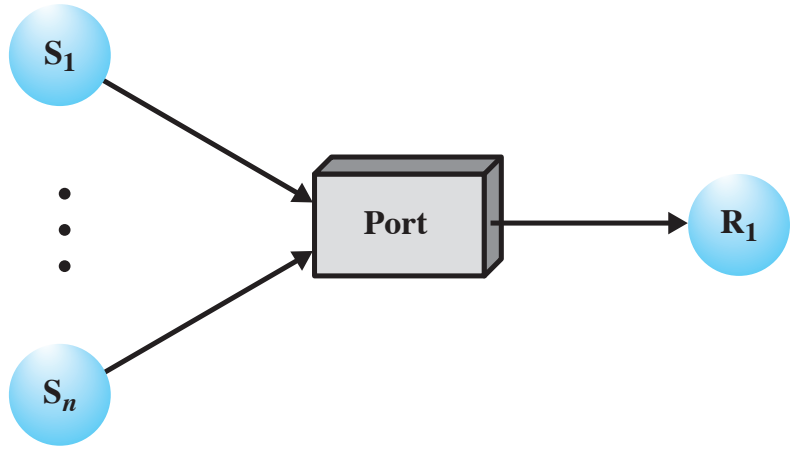
- La primitiva Send include l'identificatore del processo destinatario
- La primitiva Receive può avere
  - Indirizzamento esplicito
    - destinatario designato esplicitamente
    - utile per processi concorrenti cooperanti
  - Indirizzamento implicito
    - sender è un parametro che restituisce l'identificatore del mittente

Indirizzamento indiretto

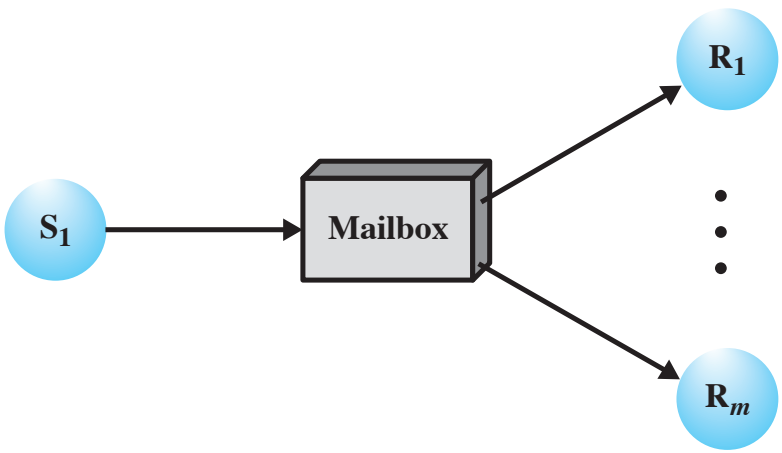
- Messaggi inviati a una struttura dati condivisa consistente in code che contengono temporaneamente i messaggi (mailbox)
- Il mittente invia un messaggio a una mailbox
- Il destinatario preleva un messaggio dalla mailbox
- Flessibilità

Indirizzamento indiretto

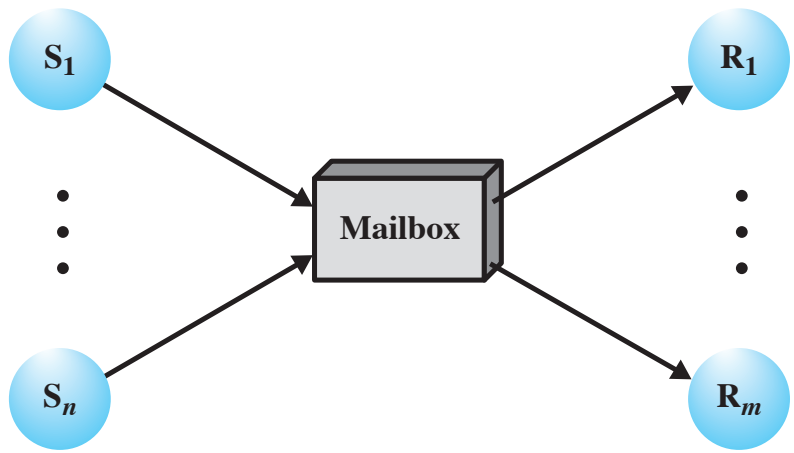
(a) One to one



(b) Many to one

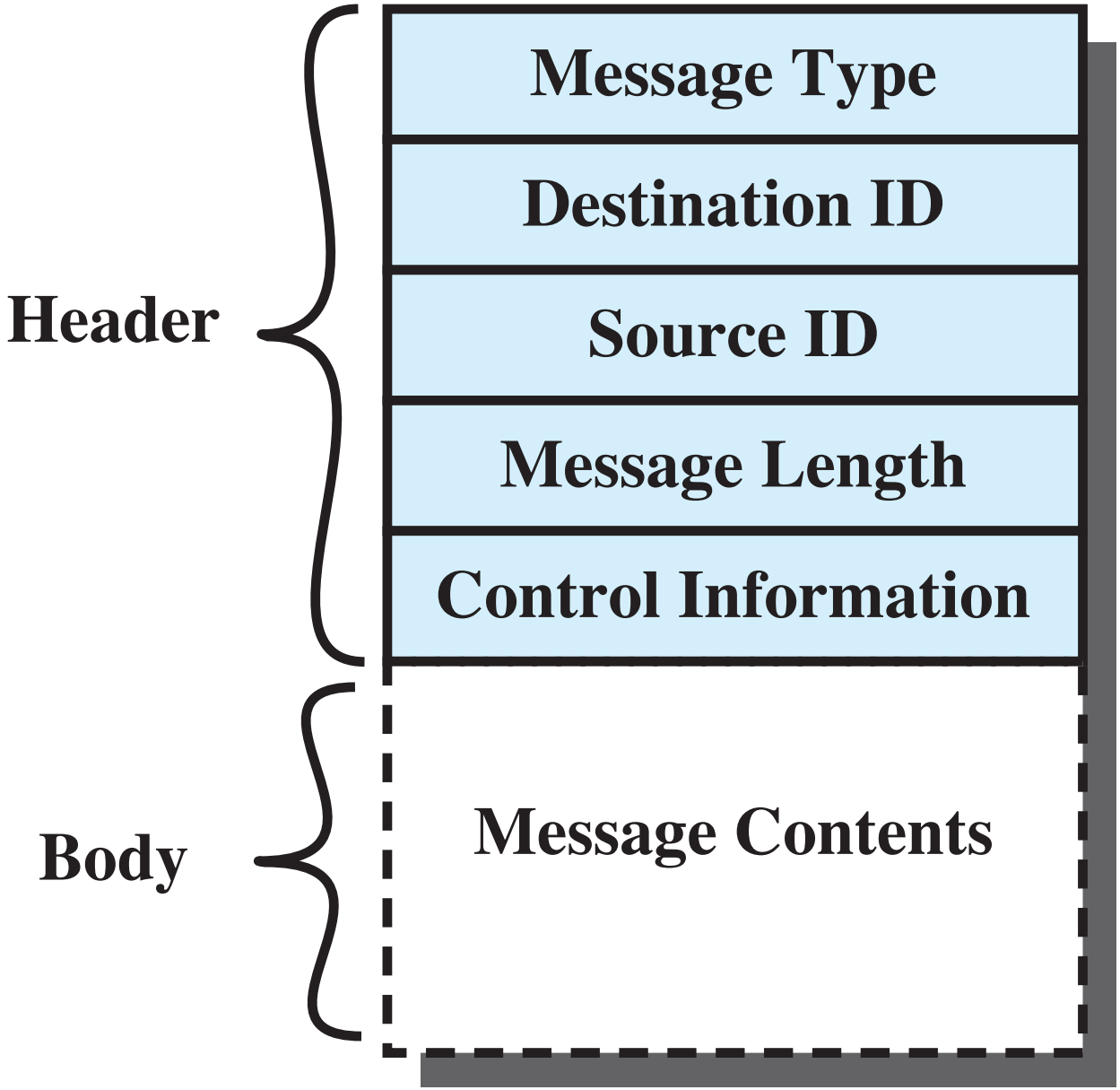


(c) One to many



(d) Many to many

Formato del messaggio



Gestione delle code

- FIFO
- Priorità
  - per tipo di messaggio
  - designata dal mittente
- Il destinatario può ispezionare la coda e scegliere il messaggio

Mutua esclusione

```
/* program mutualexclusion */
const int n = /* number of process */
void P(int i)
{
    message msg;
    while (true) {
        receive (box, msg);
        /* critical section */;
        send (box, msg);
        /* remainder */;
    }
}

void main()
{
    create mailbox (box);
    send (box, null);
    parbegin (P(1), P(2),..., P(n));
}
```

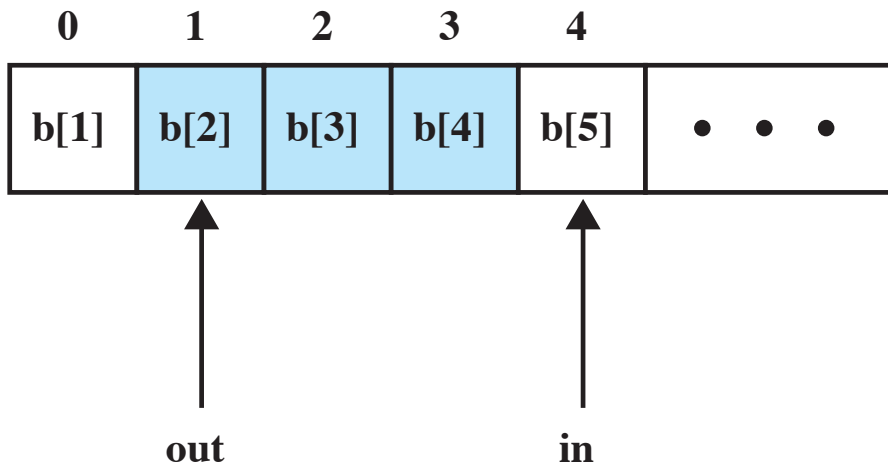
Problema produttore/consumatore

Condizioni

- Uno o più produttori inseriscono dati in un buffer
- Un consumatore estrae i dati dal buffer, uno alla volta
- Un solo produttore o consumatore alla volta accede al buffer

Problema

- Assicurare la mutua esclusione,
- evitare che un produttore aggiunga dati a un buffer pieno,
- evitare che il consumatore cerchi di estrarre dati da un buffer vuoto.



Soluzione errata

(buffer infinito)

```
/* 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);
  }
}
```

Soluzione corretta

(buffer infinito)

```
/* 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);
  }
}
```

Soluzione con semafori generici

(buffer infinito)

```
/* 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();
  }
}
```

Soluzione con semafori generici

(buffer finito)

```
/* 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();
  }
}
```

Monitor

Costrutto di alcuni linguaggi di programmazione con funzionalità analoghe ai semafori

Disponibile in numerosi linguaggi (Concurrent Pascal, Modula-2, Modula-3, Java) e in moduli di libreria

Modulo software costituito da

- dati locali
- sequenza di inizializzazione
- una o più procedure

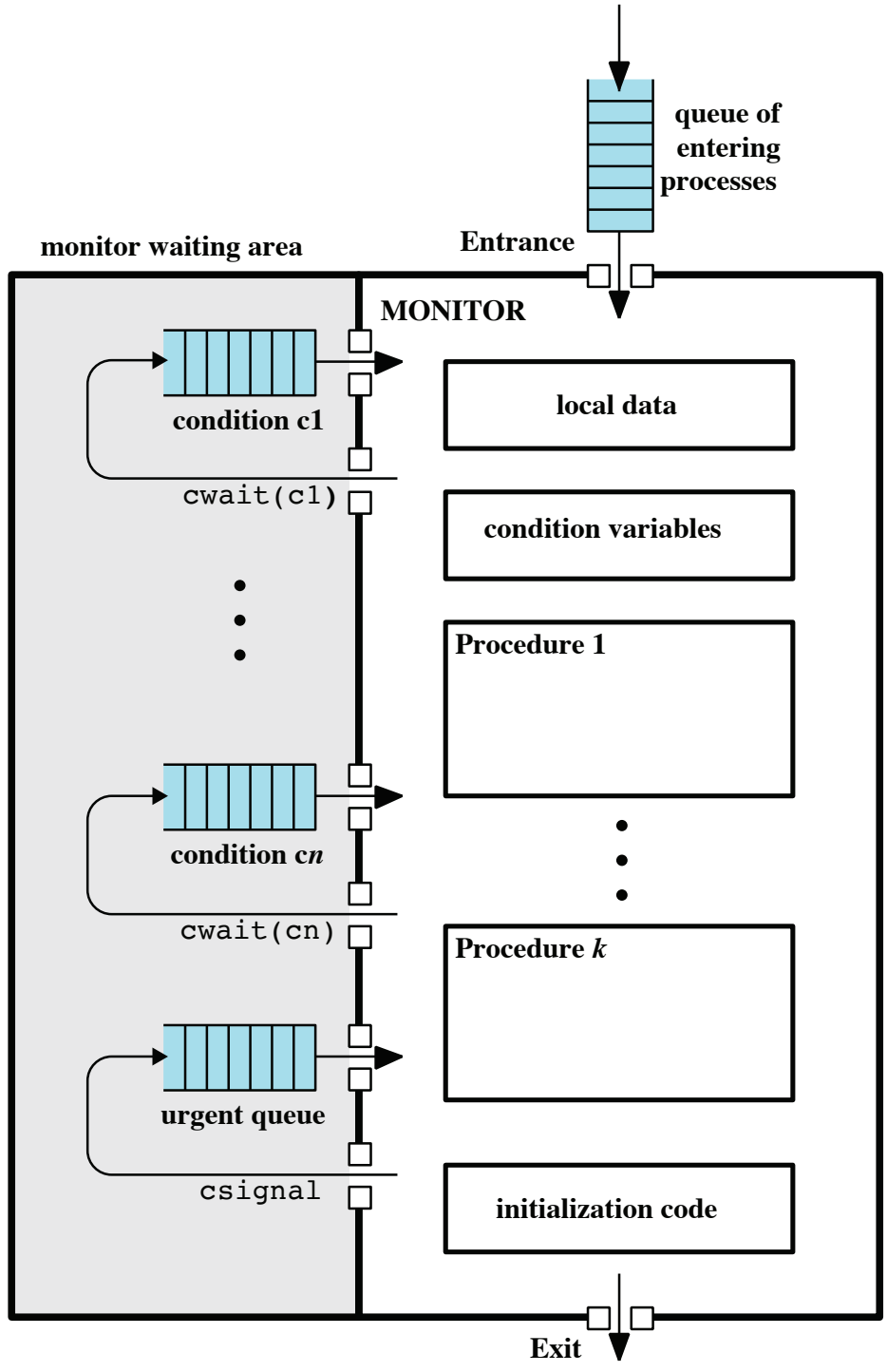
Caratteristiche

- I dati locali sono accessibili solo attraverso le procedure del monitor,
- I processi accedono invocando una delle procedure
- Un solo processo alla volta può essere in esecuzione nel monitor

Variabili di condizione

Variabili accessibili solo all'interno del monitor attraverso due funzioni:

- cwait(c) sospende l'esecuzione del processo che la ha eseguita sulla condizione c
- csignal(c) riattiva uno dei processi in attesa sulla condizione c; se non ce ne sono non fa nulla.


queue of
entering
processes
Entrance
monitor waiting area
MONITOR
condition c1
cwait(c1)
local data
condition variables
Procedure 1
condition cn
cwait(cn)
Procedure k
urgent queue
csignal
initialization code
Exit

Problema produttore/consumatore

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];          /* space for N items
int nextin, nextout;      /* buffer pointers
int count;                /* number of items in buffer
cond notfull, notempty;   /* condition variables for synchronization

void append (char x)
{
   if (count == N) cwait(notfull);    /* buffer is full; avoid overflow */
   buffer[nextin] = x;
   nextin = (nextin + 1) % N;
   count++;                           /* one more item in buffer */
   csignal(notempty);                 /* resume any waiting consumer */
}
void take (char x)
{
   if (count == 0) cwait(notempty);  /* buffer is empty; avoid underflow */
   x = buffer[nextout];
   nextout = (nextout + 1) % N;
   count--;                          /* one fewer item in buffer */
   csignal(notfull);                 /* resume any waiting producer */
}
{                                       /* monitor body */
   nextin = 0; nextout = 0; count = 0;  /* buffer initially empty */
}
```

Problema dei lettori e scrittori

Dati condivisi

- Alcuni processi leggono soltanto (lettori),
- gli altri scrivono soltanto.

Condizioni

- Qualsiasi numero di lettori può leggere simultaneamente i dati
- un solo scrittore alla volta può scrivere i dati
- durante la scrittura, nessun lettore può leggere

Precedenza ai lettori (buffer infinito)

```
/* 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); }
```

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);
  }
}
```

```
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); }
```

Scambio di messaggi

```
void reader(int i)
{
  message rmsg;
  while (true) {
    rmsg = i;
    send (readrequest, rmsg);
    receive (mbox[i], rmsg);
    READUNIT ();
    rmsg = i;
    send (finished, rmsg);
  }
}

void writer(int j)
{
  message rmsg;
  while(true) {
    rmsg = j;
    send (writerequest, rmsg);
    receive (mbox[j], rmsg);
    WRITEUNIT ();
    rmsg = j;
    send (finished, rmsg);
  }
}
```

```
void controller ()
{
  while (true)
  {
    if (count > 0) {
      if (!empty (finished)) {
        receive (finished, msg);
        count++;
      }
      else if (!empty (writerequest)) {
        receive (writerequest, msg);
        writer_id = msg.id;
        count = count - 100;
      }
      else if (!empty (readrequest)) {
        receive (readrequest, msg);
        count--;
        send (msg.id, "OK");
      }
    }
    if (count == 0) {
      send (writer_id, "OK");
      receive (finished, msg);
      count = 100;
    }
  }
  while (count < 0) {
    receive (finished, msg);
    count++;
  }
}
```