Concurrency

[A. Warhol – Marilyn Monroe, 1960]

The need for concurrency

- There are many reason for concurrency
  - Functional
  - Performance
  - Expressive power
- Functional
  - Many users may be connected to the same system at the same time
    - Each user can have its own processes that execute concurrently with the processes of the other users
  - Perform many operations concurrently
    - For example, listen to music, write with a word processor, burn a CD, etc...
    - They are all different and independent activities
    - They can be done “at the same time”

the need for concurrency (2)

- Performance
  - Take advantage of blocking time
    - While some thread waits for a blocking condition, another thread performs another operation
  - Parallelism in multi-processor machines
    - On a multi-core machine, independent activities can be carried out on different cores are the same time
- Expressive power
  - Many control application are inherently concurrent
  - Concurrency support helps in expressing concurrency, making application development simpler

Theoretical model

- A system is a set of concurrent activities
  - They can be processes or threads
- They interact in two ways
  - They access the hardware resources
    - processor
    - disk
    - memory, etc.
  - They exchange data
- These activities compete for the resources and/or cooperate for some common objective

Resource

- A resource can be
  - A HW resource like a I/O device
  - A SW resource, i.e. a data structure
  - In both cases, access to a resource must be regulated to avoid interference
- Example 1
  - If two processes want to print on the same printer, their access must be sequentialised, otherwise the two printing could be intermangled!
- Example 2
  - If two threads access the same data structure, the operation on the data must be sequentialized otherwise the data could be inconsistent!

Interaction model

- Activities can interact according to two fundamental models
  - Shared memory
    - All activities access the same memory space
  - Message passing
    - All activities communicate by sending each other messages through OS primitives
  - We will analize both models in the following slides

Cooperative vs Competitive

The interaction between concurrent activities (threads or processes) can be classified into:

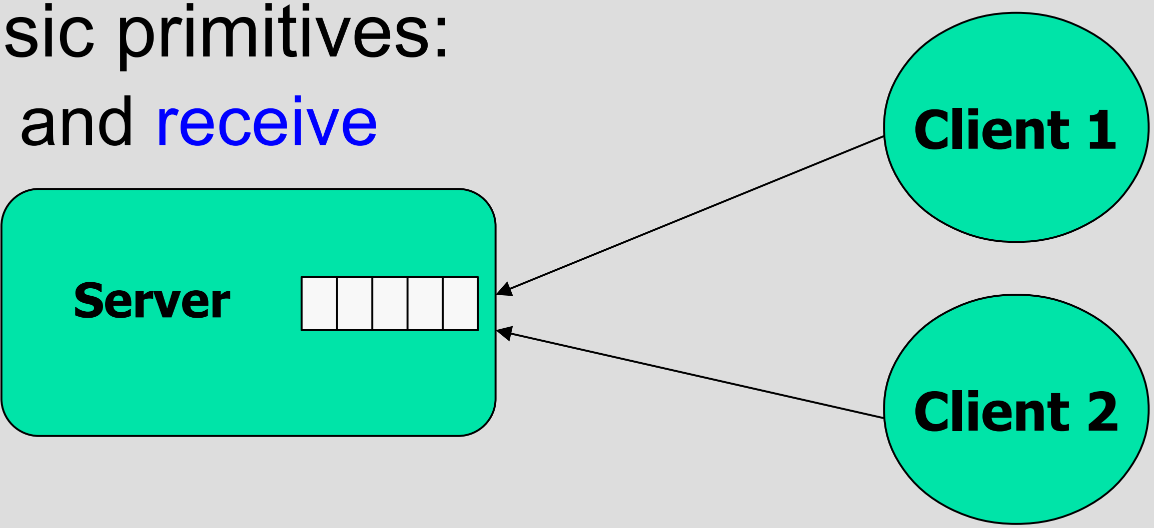
- Competitive concurrency
  - Different activities compete for the resources
  - One activity does not know anything about the other
  - The OS must manage the resources so to
    - Avoid conflicts
    - Be fair
- Cooperative concurrency
  - Many activities cooperate to perform an operation
  - Every activity knows about the others
  - They must synchronize on particular events

Competition

- Cooperative and competitive activities need different models of execution and synchronization
  - Competing activities need to be “protected” from each other
    - Separate memory spaces, as with different processes
  - The allocation of the resource and the synchronization must be centralized
    - Competitive activities request for services to a central manager (the OS or some dedicated process) which allocates the resources in a fair way
  - Client/Server model
    - Communication is usually done through messages
  - More suitable to the process model of execution

Competition (2)

- In a client/server system
  - A server manages the resource exclusively
    - For example, the printer
  - If a process needs to access the resource, it sends a request to the server
    - For example, printing a file, or asking for the status
  - The server can send back the responses
  - The server can also be on a remote system
- Two basic primitives:
  - send and receive



Cooperation

- Cooperative activities know about each other
  - They do not need memory protection
    - Not using memory protection, we have less overhead
  - They need to access the same data structures
  - Allocation of the resource is de-centralized
  - Shared memory model
  - More suitable to the thread model of execution

Cooperation and competion

- Competition is best resolved by using the message passing model
  - However it can be implemented using a shared memory paradigm too
- Cooperation is best implemented by using the shared memory paradigm
  - However, it can be realized by using pure message passing mechanisms
- Shared memory or message passing?
  - In the past, there were OS that supported only shared memory or only message passing

Cooperation and competion (2)

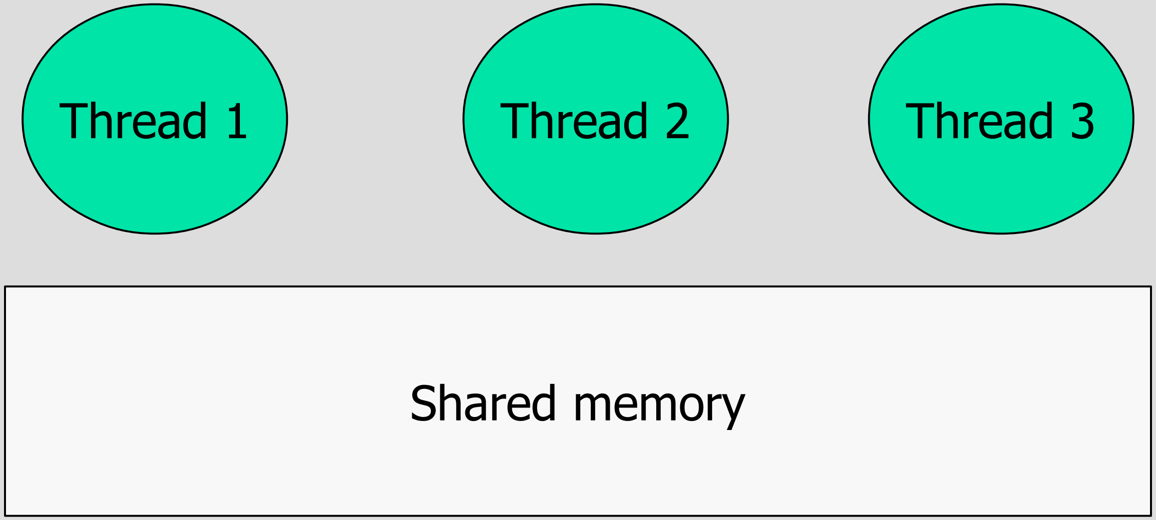
- A general purpose OS needs to support both models
  - Protection for competing activities
  - Client/server models → message passing primitives
  - Shared memory for reducing the overhead
- Some special OS supports only one of the two
  - for example, some RTOS supports only shared memory

Models of concurrency

Shared Memory

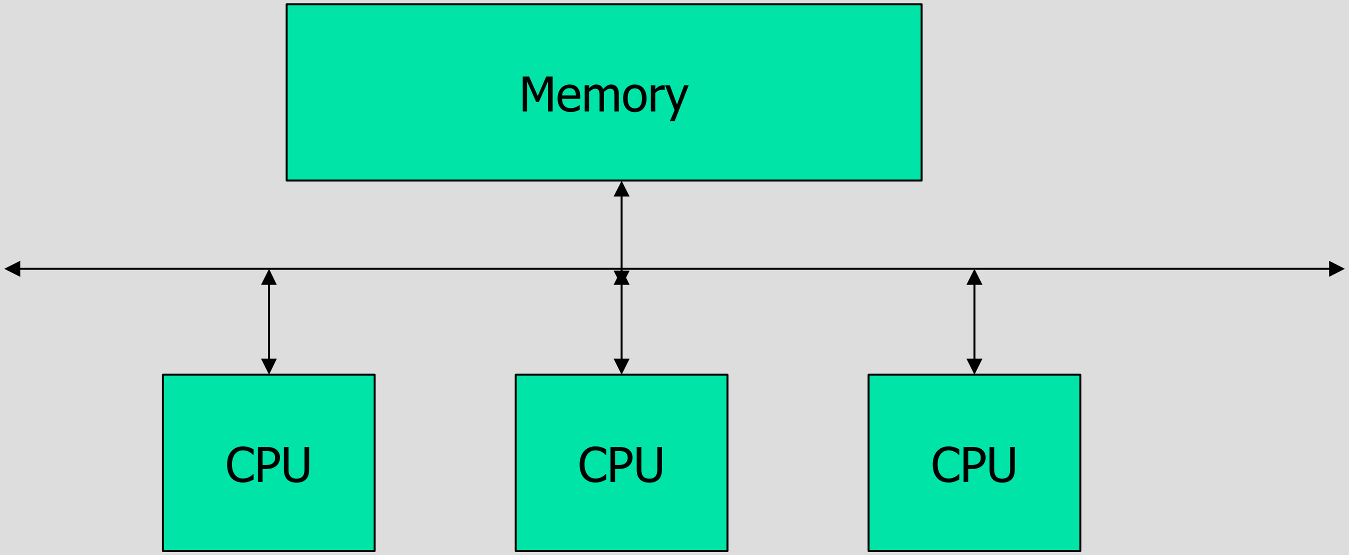
Shared memory

- Shared memory communication
  - The first one being supported in old OS
  - The simplest one and the closest to the machine
  - All threads can access the same memory locations



Hardware analogy

- An abstract model that presents a good analogy is the following
  - Many HW CPU, each one running one activity (thread)
  - One shared memory



Resource allocation

- Allocation of resource can be
  - Static: once the resource is granted, it is never revoked
  - Dynamic: resource can be granted and revoked dynamically
    - Manager
- Access to a resource can be
  - Dedicated: only one activity at a time may request access to the resource
  - Shared: many activities may access the resource at the same time
    - Mutual exclusion

	Dedicated	**Shared**
Static	Compile Time	Manager
Dynamic	Manager	Manager

Mutual exclusion problem

- We do not know in advance the relative speed of the processes
  - Hence, we do not know the order of execution of the hardware instructions

- Example:
  - Incrementing a variable x is NOT an atomic operation

Atomicity

- A hardware instruction is atomic if it cannot be “interleaved” with other instructions
  - Atomic operations are always sequentialized
  - Atomic operations cannot be interrupted
    - They are safe operations
    - For example, transferring one word from memory to register or viceversa
  - Non atomic operations can be interrupted
    - They are not “safe” operations
    - Non elementary operations are not atomic

Non-atomic operations

- Consider a “simple” operation like:

```
x = x+1;
```

- In assembler:

```
LD  R0, x
INC R0
ST  x, RO
```

  - A simple operation like incrementing a memory variable, may be composed by three machine instructions

Example 1

shared memory

```
int x ;
```

```
void *threadA(void *)
{
      ...;
      x = x + 1;
      ...;
}
```

```
void *threadB(void *)
{
      ...;
      x = x + 1;
      ...;
}
```

- Bad interleaving:

```
...
LD   R0, x      TA         x = 0
LD   R0, x      TB         x = 0
INC R0          TB         x = 0
ST   x, R0      TB         x = 1
INC R0          TA         x = 1
ST   x, R0      TA         x = 1
...
```

Example 2

Shared object (sw resource)

```
struct A_t {
      int a;
      int b;
} A;

void A_init(A_t *x) { x->a=1;       x->b=1; }
void A_inc(A_t *x) { x->a++;        x->b++; }
void A_mul(A_t *x){ x->b*=2;        x->a*=2;}
```

consistency: after each operation, a == b

```
void *threadA(void *)
{
      ...
      A_inc(&A);
      ...
}
```

```
void *threadB(void *)
{
      ...
      A_mul(&A);
      ...
}
```

- Bad interleaving

x->a++;	TA	*a = 2*
x->b*=2;	TB	*b = 2*
x->b++;	TA	*b = 3*
x->a*=2;	TB	*a = 4*

resource in a non-consistent state!

Consistency

- For each resource, we can state some consistency property
  - A consistency property C_i is a boolean expression on the values of the internal variables
  - A consistency property must hold before and after each operation
  - It does not hold during an operation
  - If the operations are properly sequentialized, the consistency properties must hold
- Formal verification
  - Let *R* be a resource, and let *C(R)* be a set of consistency properties on the resource
    - $C(R) = \{ C_i \}$

Definition: a concurrent program is *correct* if, for every possible interleaving of the operations on the resource, the consistency properties hold after each operation

Example 3: circular array

```
struct CircularArray_t {
        int array[10];
        int head, tail, num;
} queue;

void init_CA(struct CircularArray_t *a)
{ a->head=0; a->tail=0; a->num=0; }

int insert_CA(struct CircularArray_t *a,
                int elem)
{       if (a->num == 10) return 0;
        a->array[a->head] = elem;
        a->head = (a->head + 1) % 10;
        a->num++;
        return 1;
}
int extract_CA(struct CircularArray_t *a,
                int *elem)
{       if (a->num == 0) return 0;
        *elem = a->array[a->tail];
        a->tail = (a->tail + 1) % 10;
        a->num--;
        return 1;
}
```

(suppose num++ e num-- atomic)

Consistency properties

(suppose num++ and num-- atomic)

C_1: if (num == 0 || num == 10)
head == tail;

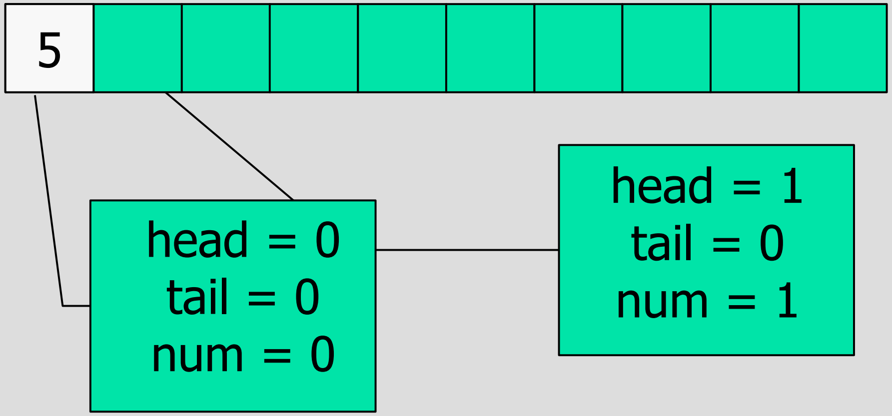
C_2: if (0 < num < 10)
num == (head – tail) % 10

C_3: num == NI - NE

C_4: (insert x)
pre: if (num < 10)
post: num == num + 1 &&
array[(head-1)%10] = x;

C_5: (extract &x)
pre: if (num > 0)
post: num == num –1 &&
x = array[(tail-1)%10];

Example 3: circular array - insert



Initial state:

head = 0; tail = 0; num = 0;

insert_CA (&queue, 5) ;

head = 1; tail = 0; num = 1;

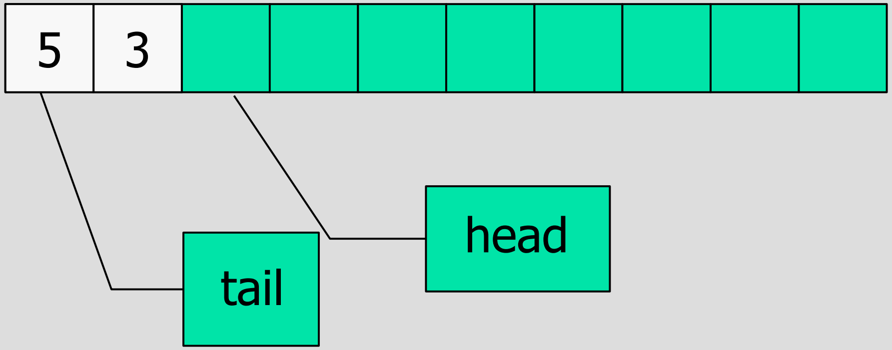
C_2, C_3, C_4 holds

C_2: if (0 < num < 10)
num == (head – tail) % 10

C_3: *num == NI – NE*

C_4: insert_CA(&queue, x)
pre: if (num < 10)
post: num == num + 1 &&
array[(head-1)%10] = x;

Example 3: circular array – insert (2)



Initial state:

head = 0; tail = 0; num = 0;

insert_CA (&queue, 5) ;

head = 1; tail = 0; num = 1;

insert_CA (&queue, 3) ;

head = 2; tail = 0; num = 2;

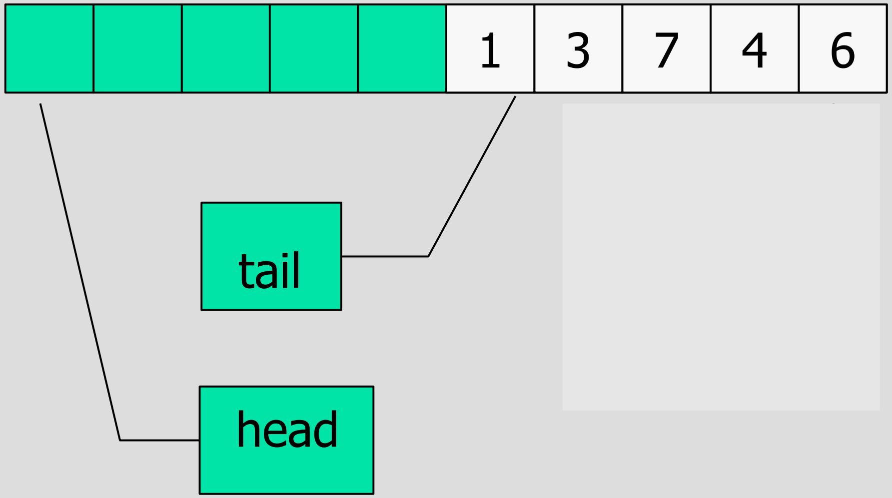
C_2, C_3, C_4 hold

C_2: if (0 < num < 10)
num == (head – tail) % 10

C_3: *num == NI – NE*

C_4: insert_CA(&queue, x)
pre: if (num < 10)
post: num == num + 1 &&
array[(head-1)%10] = x;

Example 3: circular array – insert (3)



Initial state:

head = 9; tail = 5; num = 4;

insert_CA (&queue, 6) ;

head = 0; tail = 5; num = 5

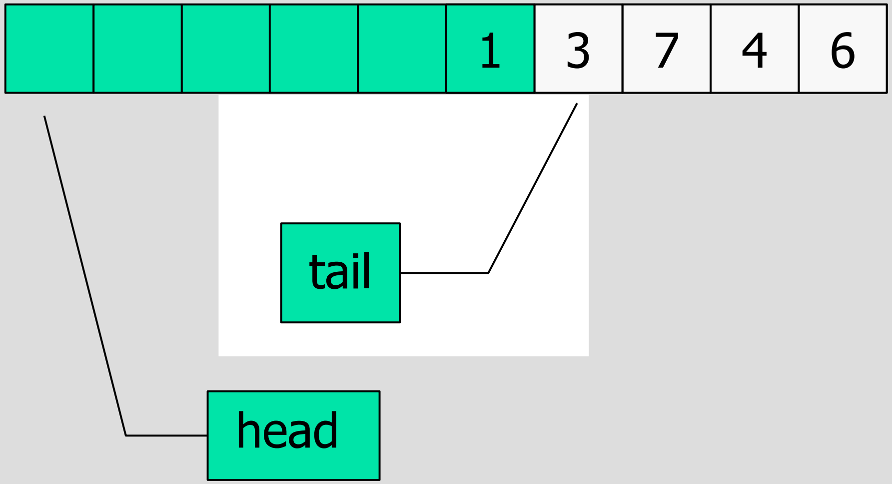
C_2, C_3, C_4 hold

C_2: if (0 < num < 10)
num == (head – tail) % 10

C_3: *num == NI – NE*

C_4: insert_CA (&queue, x)
pre: if (num < 10)
post: num == num + 1 &&
array[(head-1)%10] = x;

Example 3: circular array – extract



Initial state:

head = 0; tail = 5; num = 5;

extract_CA (&queue, &elem) ;

head = 0; tail = 6; num = 4

C_2, C_3, C_5 hold

C_2: if (0 < num < 10)
num == (head – tail) % 10

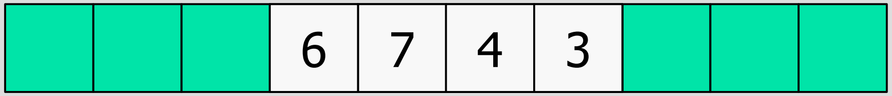
C_3: *num == NI – NE*

C_5: extract_CA (&queue, &x)
pre: if (num > 0)
post: num == num –1 &&
x = array[tail];

Example 3: the problem

- If the insert operation is performed by two processes, some consistency property may be violated!

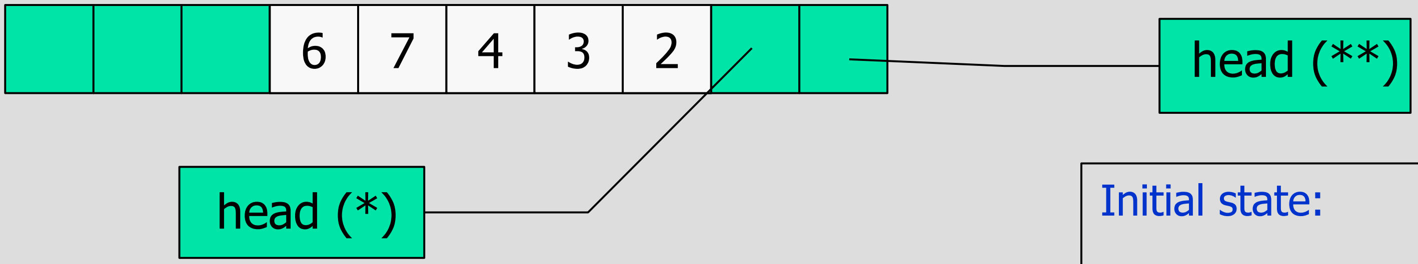
```
struct CircularArray_t queue;
```



```
void *threadA(void *)
{
    ...
    insert_CA( &queue, 5);
    ...
}
```

```
void *threadB(void *)
{
    ...
    insert_CA( &queue, 2);
    ...
}
```

Example 3: interference



C_4 is violated!

5 != array[head – 1]

```
if (a->num == 10) return 0;
a->array[a->head] = 5;
a->head = (a->head + 1) % 10; (**)
a->num ++;
return 1;
```

Initial state:

head = 7; tail = 3; num = 4;

insert_CA (&queue, 5) ; (TA)

insert_CA (&queue, 2) ; (TB)

```
if (a->num == 10) return 0; (TA)
a->array[a->head] = 5;          (TA)
if (a->num == 10) return 0; (TB)
a->array[a->head] = 2;          (TB)
a->head = (a->head + 1) % 10;(TB) (*)
a->num ++;                      (TB)
return 1;                       (TB)
a->head = (a->head + 1) % 10;(TA) (**)
a->num ++;                      (TA)
return 1;                       (TA)
```

```
if (a->num == 10) return 0;
a->array[a->head] = 2;
a->head = (a->head + 1) % 10; (*)
a->num ++;
return 1;
```

Final State:

head = 9; tail = 3; num = 6;

Example 3: correctness

- The previous program is not correct
  - It exist a possible interleaving of two insert operations that leaves the resource in a inconsistent state
- Proving the non-correctness is easy
  - it suffices to find a counter example
- Proving the correctness is not easy
  - it is necessary to prove the correctness for every possible interleaving of every operation

Example 3: problem

- What if an insert and an extract are interleaved?
  - Nothing bad can happen!!
  - Proof
    - if 0<num<10, insert_CA() and extract_CA() are independent
    - if num==0
      - if extract_CA begins before insert_CA, it immediately returns 0, so nothing bad can happen
      - if insert_CA begins before, extract_CA will still return false, so it cannot interfere with insert
    - same thing when num==10
- Question: what happens if we exchange the sequence of instructions in insert or extract?

Example 3: CircularArray properties

- a) if more than one thread executes insert_CA()
  - inconsistency!!
- b) if we have only two threads
  - one threads calls insert_CA() and the other thread calls extract_CA()
  - no inconsistency!
- The order of the operations is important!
  - a wrong order can make the object inconsistency even under the assumption b)
    - the case is when num is incremented but the data has not yet been inserted
    - in any case, the final result depends on the timings of the dfferent requests (e.g, an insertion with the buffer full)

Example 3: questions

- Problem:
  - In the previous example, we supposed that num++ and num-- are atomic operations
  - What happens if they are not atomic?
- Question:
  - Assuming that operation -- and ++ are not atomic, can we make the circularArray safe under the assumption b) ?
    - Hint: try to substitute variable num with two boolean variables: bool empty and bool full;

Critical sections

- Definitions
  - The shared object where the conflict may happen is a "resource"
  - The parts of the code where the problem may happen are called "critical sections"
    - A critical section is a sequence of operations that cannot be interleaved with other operations on the same resource
  - Two critical sections on the same resource must be properly sequentialized
  - We say that two critical sections on the same resource must execute in MUTUAL EXCLUSION
  - There are two ways to obtain mutual exclusion
    - Disabling the preemption (valid only for single-core systems)
    - Implementing the critical section as an atomic operation, using semaphores and mutexes

Critical sections: disabling preemption

- Single core systems
  - In some scheduler, it is possible to disable preemption for a limited interval of time
  - Problems:
    - If a high priority critical thread needs to execute, it cannot make preemption and it is delayed
    - Even if the high priority task does not access the resource!

```
<disable preemption>
<critical section>
<enable preemption>
```

no context switch may happen during the critical section

Critical sections: atomic operations

- There exist some general mechanisms to implement mutual exclusion only between the processes that uses a resource:
  - semaphores
  - mutexes
- Define a flag s for each resource
- Use lock(s)/unlock(s) around the critical section

```
int s;
...
lock(s);
<critical section>
unlock(s);
...
```

Synchronisation

- Mutual exclusion is not the only problem
  - We need a way of synchronise two or more threads
- Example: producer/consumer
  - Suppose we have two threads,
    - One produces some integers and sends them to another thread (PRODUCER)
    - Another one takes the integer and elaborates it (CONSUMER)

Producer/consumer

- The two threads have different speeds
  - For example the producer is much faster than the consumer
  - We need to store the integers in a queue, so that no data is lost
  - Let’s use the CircularArray_t structure

Producer/consumer (2)

```
struct CircularArray_t queue;
```

```
void *producer(void *)
{
    bool res;
    int data;
    while(1) {
        <obtain data>
        while (!insert_CA(&queue, data));
    }
}
```

```
void *consumer(void *)
{
    bool res;
    int data;
    while(1) {
        while (!extract_CA(&queue, &data));
        <use data>
    }
}
```

- Problems with this approach:
  - If the queue is full, the producer actively waits
  - If the queue is empty, the consumer actively waits

A more general approach

- We need to provide a general mechanism for synchonisation and mutual exclusion
- Requirements
  - Provide mutual exclusion between critical sections
    - Avoid two insertions operation to interleave
  - Synchronise two threads on one condition
    - For example, block the producer when the queue is full

General mechanism: semaphores

- Djikstra proposed the semaphore mechanism
  - A semaphore is an abstract entity that consists of
    - A counter
    - A blocking queue
    - Operation wait
    - Operation signal
  - The operations on a semaphore are considered atomic

Semaphores

- Semaphores are basic mechanisms for providing synchronization
  - It has been shown that every kind of synchronization and mutual exclusion can be implemented by using semaphores
  - We will analyze possible implementation of the semaphore mechanism later

```
typedef struct {
    <blocked queue> blocked;
    int counter;
} sem_t;

void sem_init      (sem_t *s, int n);

void sem_wait      (sem_t *s);
void sem_post      (sem_t *s);
```

Note:
the real prototype
of sem_init is
slightly different!

Wait and signal

- A wait operation has the following behavior
  - If counter == 0, the requiring thread is blocked
    - It is removed from the ready queue
    - It is inserted in the blocked queue
  - If counter > 0, then counter--;
- A post operation has the following behavior
  - If counter == 0 and there is some blocked thread, unblock it
    - The thread is removed from the blocked queue
    - It is inserted in the ready queue
  - Otherwise, increment counter

Semaphores

```
void sem_init (sem_t *s, int n)
{
  s->count=n;
  ...
}

void sem_wait(sem_t *s)
{
  if (counter == 0)
        <block the thread>
  else
        counter--;
}

void sem_post(sem_t *s)
{
  if (<there are blocked threads>)
        <unblock a thread>
  else
        counter++;
}
```

Signal semantics

- What happens when a thread blocks on a semaphore?
  - In general, it is inserted in a BLOCKED queue
- Extraction from the blocking queue can follow different semantics:
  - Strong semaphore
    - The threads are removed in well-specified order
    - For example, FIFO order, priority based ordering, ...
  - Signal and suspend
    - After the new thread has been unblocked, a thread switch happens
  - Signal and continue
    - After the new thread has been unblocked, the thread that executed the signal continues to execute
- Concurrent programs should not rely too much on the semaphore semantic

Mutual exclusion with semaphores

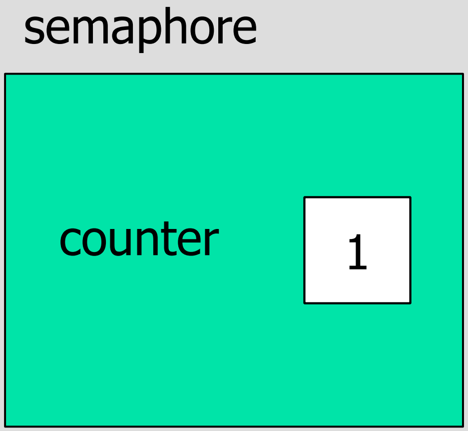
- How to use a semaphore for critical sections
  - Define a semaphore initialized to 1
  - Before entering the critical section, perform a wait
  - After leaving the critical section, perform a post

```
sem_t s;
...
sem_init(&s, 1);
```

```
void *threadA(void *arg)
{
     ...
     sem_wait(&s);
       <critical section>
     sem_post(&s);
     ...
}
```

```
void *threadB(void *arg)
{
     ...
     sem_wait(&s);
       <critical section>
     sem_post(&s);
     ...
}
```

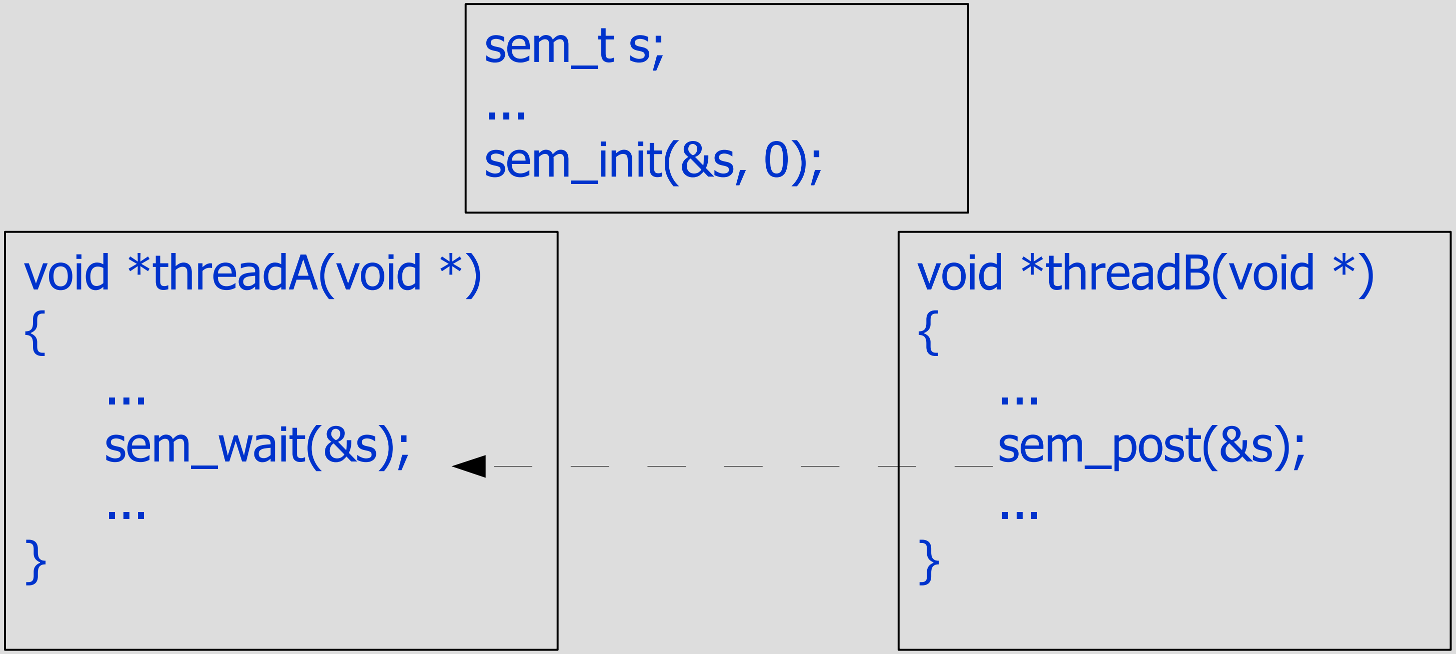
Mutual exclusion with semaphores (2)



sem_wait();	(TA)
<critical section (1)>	(TA)
sem_wait()	(TB)
<critical section (2)>	(TA)
sem_post()	(TA)
<critical section>	(TB)
sem_post()	(TB)

Synchronization

- How to use a semaphore for synchronization
  - Define a semaphore initialized to 0
  - At the synchronization point, follower performs a wait
  - At the synchronization point, producer performs a post
  - In the example, threadA blocks until threadB wakes it up



```
sem_t s;
...
sem_init(&s, 0);
```

```
void *threadA(void *)
{
     ...
     sem_wait(&s);
     ...
}
```

```
void *threadB(void *)
{
     ...
     sem_post(&s);
     ...
}
```

  - How can both A and B synchronize at the same point?

Producer/consumer

- Consider a producer/consumer system
  - One producer executes insert_CA()
    - We want the producer to be blocked when the queue is full
    - The producer will be unblocked when there is some space again
  - One consumer executes extract_CA()
    - We want the consumer to be blocked when the queue is empty
    - The consumer will be unblocked when there is some space again
  - First attempt: one producer and one consumer only

Producer/consumer (2)

```
struct CircularArray_t {
	int array[10];
	int head, tail;
	sem_t empty, full;
}
void init_CA(struct CircularArray_t *c)
{ c->head=0; c->tail=0;
   sem_init(&c->empty, 0); sem_init(&c->full, 10); }

void insert_CA(struct CircularArray_t *c, int elem) {
	sem_wait(&c->full);
		c->array[c->head] = elem;
		c->head = (c->head + 1) % 10;
	sem_post(&c->empty);
}
void extract_CA(struct CircularArray_t *c, int &elem) {
	sem_wait(&c->empty);
		elem = c->array[c->tail];
		c->tail = (c->tail + 1) % 10;
	sem_post(&c->full);
}
```

Note: there is no member called *num* as we had in Example 3 (slide 23)

Producer/consumer: properties

- Notice that
  - The value of the counter of empty is the number of elements in the queue
    - It is the number of times we can call extract without blocking
  - The value of the counter of full is the complement of the elements in the queue
    - It is the number of times we can call insert without blocking

- Exercise
  - Prove that the implementation is correct
    - insert_CA() never overwrites elements
    - extract_CA() always gets an element of the queue

Producers/consumers

- Now let’s combine mutual exclusion and synchronization
  - Consider a system in which there are
    - Many producers
    - Many consumers
  - We want to implement synchronization
  - We want to protect the data structure

Producers/consumers: does it work?

```
struct CircularArray_t {
      int array[10];
      int head, tail;
      sem_t full, empty;
      sem_t mutex;
}
void init_CA(struct CircularArray_t *c)
{
      c->head=0; c->tail=0;
      sem_init(&c->empty, 0); sem_init(&c->full, 10); sem_init(&c->mutex, 1);
}
```

```
void insert_CA(struct CircularArray_t *c,
              int elem)
{
      sem_wait(&c->mutex);
        sem_wait(&c->full);
          c->array[c->head]=elem;
          c->head = (c->head+1)%10;
        sem_post(&c->empty);
      sem_post(&c->mutex);
}
```

```
void extract_CA(struct CircularArray_t *c,
              int *elem)
{
      sem_wait(&c->mutex);
        sem_wait(&c->empty);
          elem = c->array[c->tail];
          c->tail = (c->tail+1)%10;
        sem_post(&c->full);
      sem_post(&c->mutex);
}
```

Producers/consumers: correct solution

```
struct CircularArray_t {
	int array[10];
	int head, tail;
	sem_t full, empty;
	sem_t mutex;
}
void init_CA(struct CircularArray_t *c)
{
	c->head=0; c->tail=0;
	sem_init(&c->empty, 0); sem_init(&c->full, 10); sem_init(&c->mutex, 1);
}
```

```
void insert_CA(struct CircularArray_t *c,
		int elem)
{
	sem_wait(&c->full);
	  sem_wait(&c->mutex);
	    c->array[c->head]=elem;
	    c->head = (c->head+1)%10;
	  sem_post(&c->mutex);
	sem_post(&c->empty);
}
```

```
void extract_CA(struct CircularArray_t *c,
		int *elem)
{
	sem_wait(&c->empty);
	  sem_wait(&c->mutex);
	    elem = c->array[c->tail];
	    c->tail = (c->tail+1)%10;
	  sem_post(&c->mutex);
	sem_post(&c->full);
}
```

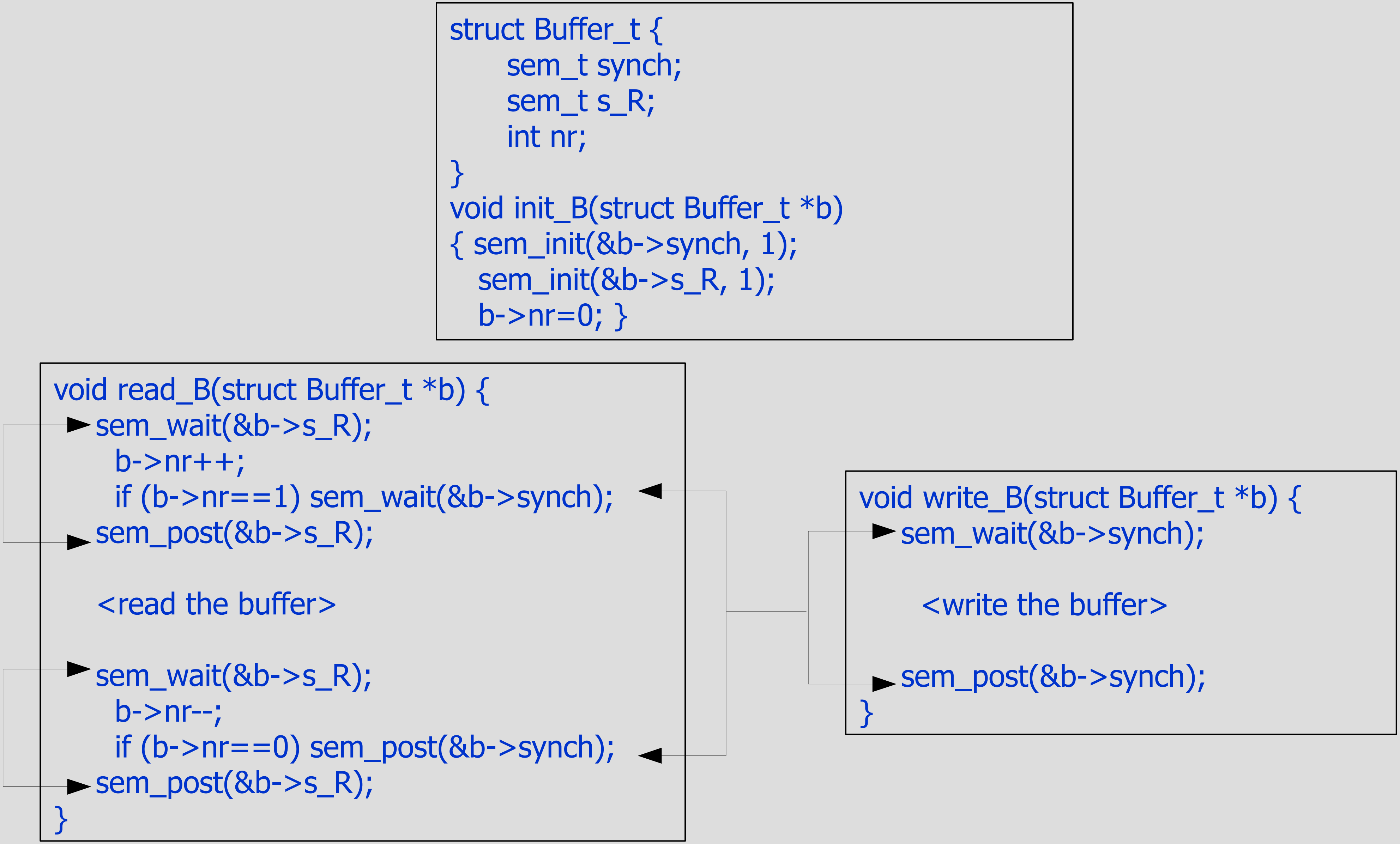
Producers/consumers: deadlock situation

- Deadlock situation
  - A thread executes sem_wait(&c->mutex) and then blocks on a synchronisation semaphore
  - To be unblocked another thread must enter a critical section guarded by the same mutex semaphore!
  - So, the first thread cannot be unblocked and free the mutex!
  - The situation cannot be solved, and the two threads will never proceed
- As a rule, never insert a blocking synchronization inside a critical section!!!

Readers/writers

- One shared buffer
- Readers:
  - They read the content of the buffer
  - Many readers can read at the same time
- Writers
  - They write in the buffer
  - While one writer is writing no other reader or writer can access the buffer
- Use semaphores to implement the resource

Readers/writers: simple implementation



```
struct Buffer_t {
        sem_t synch;
        sem_t s_R;
        int nr;
}
void init_B(struct Buffer_t *b)
{ sem_init(&b->synch, 1);
   sem_init(&b->s_R, 1);
   b->nr=0; }
```

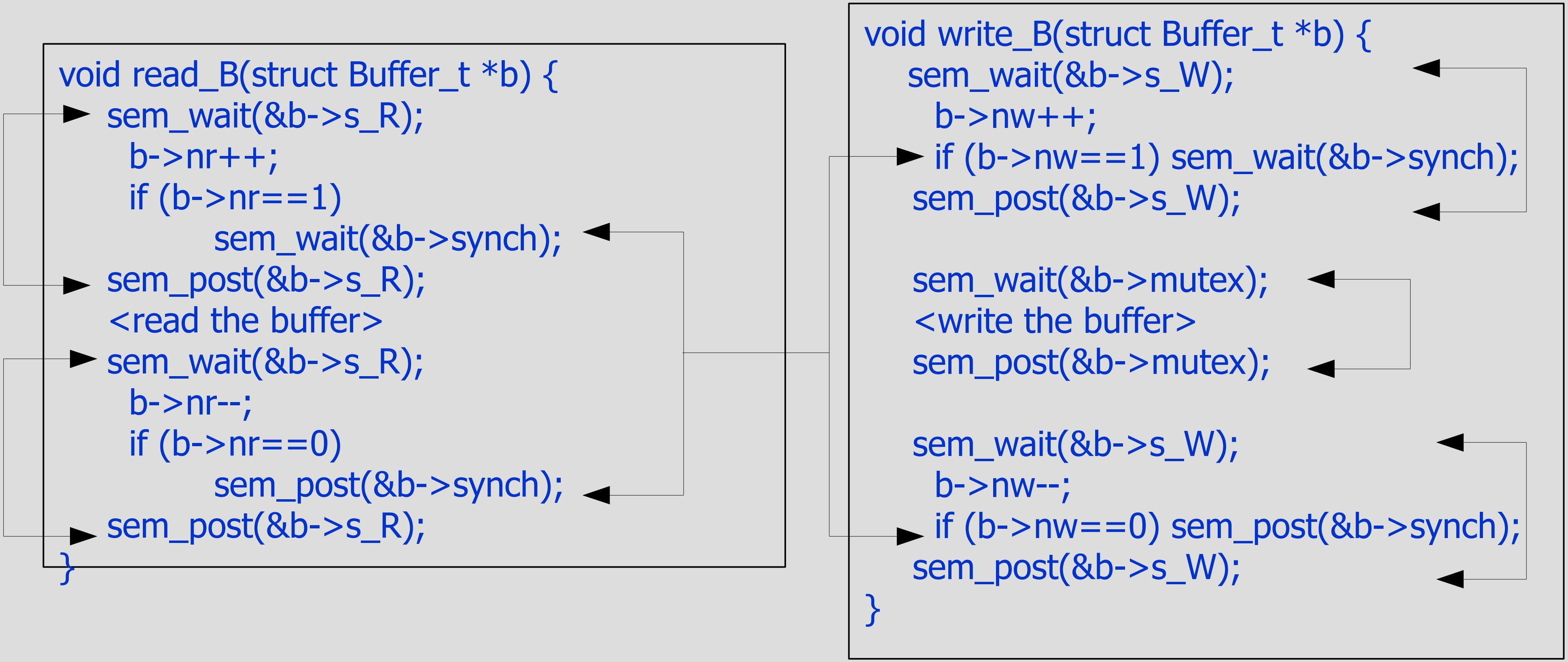
```
void read_B(struct Buffer_t *b) {
  sem_wait(&b->s_R);
    b->nr++;
    if (b->nr==1) sem_wait(&b->synch);
  sem_post(&b->s_R);

  <read the buffer>

  sem_wait(&b->s_R);
    b->nr--;
    if (b->nr==0) sem_post(&b->synch);
  sem_post(&b->s_R);
}
```

```
void write_B(struct Buffer_t *b) {
  sem_wait(&b->synch);

    <write the buffer>

  sem_post(&b->synch);
}
```

Readers/writers: more than one pending writer

```
struct Buffer_t {
      sem_t synch, mutex;
      sem_t s_R, s_W;
      int nr, nw;
};
```

```
void init_B(struct Buffer_t *b)
{
  sem_init(&b->synch, 1); sem_init(&b->mutex(1);
  sem_init(&b->s_R, 1); sem_init(&b->s_W, 1);
  b->nr=0; b->nw=0;
}
```



```
void read_B(struct Buffer_t *b) {
    sem_wait(&b->s_R);
      b->nr++;
      if (b->nr==1)
              sem_wait(&b->synch);
    sem_post(&b->s_R);
    <read the buffer>
    sem_wait(&b->s_R);
      b->nr--;
      if (b->nr==0)
              sem_post(&b->synch);
    sem_post(&b->s_R);
}
```

```
void write_B(struct Buffer_t *b) {
    sem_wait(&b->s_W);
      b->nw++;
      if (b->nw==1) sem_wait(&b->synch);
    sem_post(&b->s_W);

    sem_wait(&b->mutex);
    <write the buffer>
    sem_post(&b->mutex);

    sem_wait(&b->s_W);
      b->nw--;
      if (b->nw==0) sem_post(&b->synch);
    sem_post(&b->s_W);
}
```

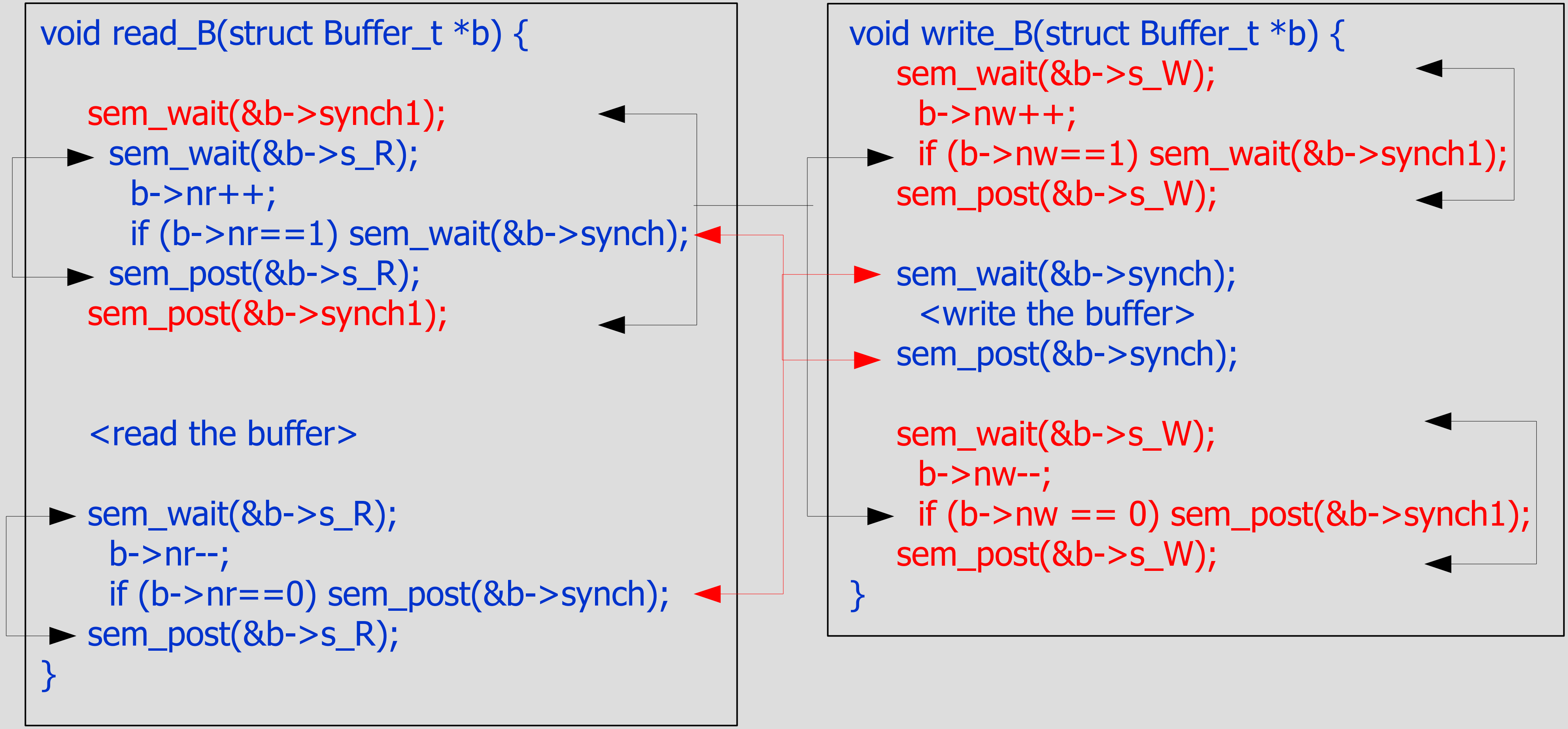
Readers/writers: starvation

- A reader will be blocked for a finite time
- The writer suffers starvation
- Suppose we have 2 readers (R1 and R2) and 1 writer W1
  - Suppose that R1 starts to read
  - While R1 is reading, W1 blocks because it wants to write
  - R2 starts to read
  - R1 finishes, but, since R2 is reading, W1 cannot be unblocked
  - Before R2 finishes to read, R1 starts to read again
  - When R2 finishes, W1 cannot be unblocked because R1 is reading
- A solution
  - Readers should not be counted whenever there is a writer waiting for them

Readers/writers: priority to writers!

```
struct Buffer_t {
        sem_t synch, synch1;
        sem_t s_R, s_W;
        int nr, nw;
};
```

```
void init_B(struct Buffer_t *b) {
  sem_init(&b->synch, 1); sem_init(&b->synch1, 1);
  sem_init(&b->s_R, 1); sem_init(&b->s_W, 1);
  b->nr=0; b->nw=0;
}
```



```
void read_B(struct Buffer_t *b) {

    sem_wait(&b->synch1);
      sem_wait(&b->s_R);
        b->nr++;
        if (b->nr==1) sem_wait(&b->synch);
      sem_post(&b->s_R);
    sem_post(&b->synch1);


    <read the buffer>

    sem_wait(&b->s_R);
      b->nr--;
      if (b->nr==0) sem_post(&b->synch);
    sem_post(&b->s_R);
}
```

```
void write_B(struct Buffer_t *b) {
    sem_wait(&b->s_W);
      b->nw++;
      if (b->nw==1) sem_wait(&b->synch1);
    sem_post(&b->s_W);

    sem_wait(&b->synch);
      <write the buffer>
    sem_post(&b->synch);

    sem_wait(&b->s_W);
      b->nw--;
      if (b->nw == 0) sem_post(&b->synch1);
    sem_post(&b->s_W);
}
```

Readers/writers: problem

- Now, there is starvation for readers
- The readers/writers problem can be solved in general?
  - No starvation for readers
  - No starvation for writers
- Solution
  - Maintain a FIFO ordering with requests
    - If at least one writer is blocked, every next reader blocks
    - If at least one reader is blocked, every next writer blocks
- We can do that using the private semaphores technique

Private semaphores: when to use it

- The private semaphores technique can be used every time the system wants to specify the policy to be used when waking up a particular thread/process
- Examples
  - In the readers/writers problem, we want to avoid starvation of both readers and writers
  - When a resource becomes free, and there are more than one process waiting, we want to activate a *particular* proces following a given *policy*

Private semaphore: what is it?

- In general, when using a resource, a process will block on a synchronization point because some kind of test fails
  - Example, a process tries to insert an element in a full buffer. The “buffer full” is the synchronization test
- When the the process will block, it will block on a private semaphore
- A private semaphore is a semaphore used only by one process
  - (or only by a class of processes)
- There are two ways for using a private semaphore

Private semaphores: solution 1

```
struct myresource_t {
      sem_t mutex;
      sem_t priv[MAXPROC];
      ...
}
```

```
void myresource_init(...)
{
       <mutex initialized to 1>
       <private semaphores initialized to 0>
       ...
}
```

Checking a condition to eventually block

```
void f1(struct myresource_t *r)
{
    sem_wait(&r->mutex);

    if <condition> {
              <resource allocation to i>
              sem_post(&r->priv[i]);
    }
    else
              <record that i is suspended >

    sem_post(&r->mutex);
    sem_wait(&r->priv[i]);
}
```

Changing a blocking condition

```
void f2(struct myresource_t *r) {
{
    int i;
    sem_wait(&r->mutex);
    <release the resource>

    if <wake up someone> {
              i = <process to wake up>
              <resource allocation to i>
              <record that i is no more
                            suspended>
              sem_post(&r->priv[i]);
    }
    sem_post(&r->mutex);
}
```

Private semaphores: notes to solution 1

- The wait on the private semaphore is outside the mutex critical region
- Each process blocks on a separate private semaphore
  - In this way, the release can choose exactly which is the task to wake up
- Disadvantages
  - When acquiring, the wait on the private semaphore is always done
  - The resource allocation is done both in the acquisition and in the release

Private semaphores: solution2

```
struct myresource_t {
      sem_t mutex;
      sem_t priv[MAXPROC];
      ...
}
```

```
void myresource_init(...)
{
       <mutex initialized to 1>
       <private semaphores initialized to 0>
       ...
}
```

Checking a condition to eventually block

```
void f1(struct myresource_t *r)
{
    sem_wait(&r->mutex);

    if <not condition> {
              <record that i is suspended>
              sem_post(&r->mutex);
              sem_wait(&r->priv[i].wait);
              <record that i has been
                woken up>
    }

    <resource allocation to i>

    sem_post(&r->mutex);
```

Changing a blocking condition

```
void f2(struct myresource_t *r) {
{
    int i;
    sem_wait(&r->mutex);
     <release the resource>

     if <wake up someone> {
               i = <process to wake up>
               sem_post(&r->priv[i]);
     }
     else
               sem_post(&r->mutex);
}
```

Private semaphores: notes to solution 2

- Using solution 2, it is hard to wake up more than one process at the same time

Readers/writers: solution

```
struct Buffer_t {
    int nbr, nbw;
    int nr, nw;
    sem_t priv_r, priv_w;
    sem_t m;
}

void Buffer_init(struct Buffer_t *b)
{
    b->nbw=0; b->nbr=0;
    b->nr=0; b->nw=0;
    sem_init(&b->priv_r,0);
    sem_init(&b->priv_w,0);
    sem_init(&b->m,1);
}
```

Readers/writers: solution (2)

```
void Buffer_read(struct Buffer_t b)
{
    sem_wait(&b->m);
    if (nw>0 || nbw>0)
       nbr++;
    else {
       nr++;
       sem_post(&b->priv_r);
    }
    sem_post(&b->m);
    sem_wait(&b->priv_r);

    <read buffer>;

    sem_wait(&b->m);
    nr--;
    if (nbw>0 && nr == 0) {
       nbw--; nw++;
       sem_post(&b->priv_w);
    }
    sem_post(&b->m);
}
```

```
void Buffer_write(struct Buffer_t b)
{
    sem_wait(&b->m);
    if (nr>0 || nw>0)
       nbw++;
    else {
       nw++;
       sem_post(&b->priv_w);
    }
    sem_post(&b->m);
    sem_wait(&b->priv_w);

    <write buffer>;

    sem_wait(&b->m);
    nw--;
    if (nbr>0)
       while (nbr>0)
           { nbr--; nr++; sem_post(&b->priv_r); }
    else if (nbw>0)
       { nbw--; nw++; sem_post(&b->priv_w); }
    sem_post(&b->m);
}   // NB: nw can have only values 0 or 1!!
```

Private semaphores: final notes

- These general rules apply
  - When a process blocks on a private semaphore, it has to leave some information in the shared structure saying it has blocked
  - Also the fact that a resource is used by a process is recorded in the internal data structures
  - The assignment of a resource to a process is separated from the use of the resource
  - Note that the usage of the resource does not need anymore to be protected by a mutex

Semaphore implementation

- System calls
  - sem_wait() and sem_post() involve a possible thread-switch
  - Therefore they must be implemented as system calls!
    - One blocked thread must be removed from state RUNNING and be moved in the semaphore blocking queue
- Protection:
  - A semaphore is itself a shared resource
  - sem_wait() and sem_post() are critical sections!
  - They must run with interrupt disabled and by using lock() and unlock() primitives

Semaphore implementation (2)

```
void sem_wait(sem_t *s)
{
    spin_lock_irqsave();
    if (counter==0) {
        <block the thread>
        schedule();
    } else s->counter--;
    spin_lock_irqrestore();
}
```

```
void sem_post(sem_t *s)
{
    spin_lock_irqsave();
    if (counter== 0) {
        <unblock a thread>
        schedule();
    } else s->counter++;
    spin_lock_irqrestore();
}
```

Monitors

- Monitors are a language structure equivalent to semaphores, but cleaner
  - A monitor is similar to an object in a OO language
  - It contains variables and provides procedures to other software modules
  - Only one thread can execute a procedure at a certain time
    - Any other thread that has invoked the procedure is blocked and waits for the first threads to exit
    - Therefore, a monitor implicitely provides mutual exclusion
  - The source code that is used to implement the mutual exclusion is automatically inserted by the compiler

Condition variables

- Monitors support synchronization with *Condition Variables*
  - A condition variable is a blocking queue
  - Two operations are defined on a condition variable
    - cond_wait() -> suspends the calling thread on the queue
    - cond_signal() -> resumes execution of one thread blocked on the queue
- Important note:
  - cond_wait() and cond_signal() operation on a condition variable are different from sem_wait and sem_post on a semaphore!
  - There is not any counter in a condition variable!
  - If we do a signal on a condition variable with an empty queue, the signal is lost
  - There are 6 ways to implementa monitor construct
    - we will only look at the POSIX approach
      (that is the same used by the MESA language)

Condition variables (2)

- When a process blocks on a condition variable, the mutual exclusion is released to let someone else modify the shared data structure
- When it is then woken up by someone, it has to check again for the blocking condition
  - Because someone could have modified the data structure
- That is, condition variables are always used inside a while()

CircularArray with monitors

```
struct CircularArray_t {
      int array[10];
      int head, tail, num;
      Condition empty, full;
} queue;
```

```
void CircularArray_init(struct CircularArray_t *ca)
{
       ca->head = 0;
       ca->tail = 0;
       ca->num =0;
}
```

```
void CircularArray_insert(
         struct CircularArray_t *ca,
         int elem
       ) synchronized
{
      while (num==10)  cond_wait(&full);
      array[head]=elem;
      head = (head+1)%10;
      num++;
      if (num==1) cond_signal(empty);
}
```

```
void CircularArray_extract(int &elem)
 synchronized
{
      while (num== 0) cond_wait(&empty);
      elem = array[tail];
      tail = (tail+1)%10;
      num--;
      if (num == 9) cond_signal(&full);
}
```

Monitors and POSIX

- POSIX is an interface, not a language
- For that reason, a POSIX program has to explicitly say where a critical section starts and ends
- Mutexes are used to bound a critical section
  - A mutex is a binary semaphore with two functions, lock and unlock
- Condition variables must stay inside a while loop

```
void CircularArray_insert(struct CircularArray_t *ca,
        int elem)
{
      pthread_mutex_lock(&ca->mymutex);
      while (num==10)  pthread_cond_wait(&ca->full,&ca->mymutex);
      array[head]=elem;
      head = (head+1)%10;
      num++;
      if (num==1) pthread_cond_signal(&ca->empty);
      pthread_mutex_unlock(&ca->mymutex);
}
```

Policies and monitors

- Waking up policies can be implemented using private conditions
- The idea is the same of the private semaphores, but implemented using condition variables :-)

Problem

- Implement the readers/writers problem with monitors
  - Hint: follow the previous solution with semaphores!

Models of concurrency

Message Passing

Message passing

- Message passing systems are based on the basic concept of message
- Two basic operations
  - send(destination, message);
    - send can be synchronous or asynchronous

  - receive(source, &message);
    - receive can be symmetric or asymmetric

Producer/Consumer with MP

- The producer executes `send(consumer, data)`
- the consumer executes `receive(producer, data)`
- no need for a special communication structure (already contained in the send/receive semantic)

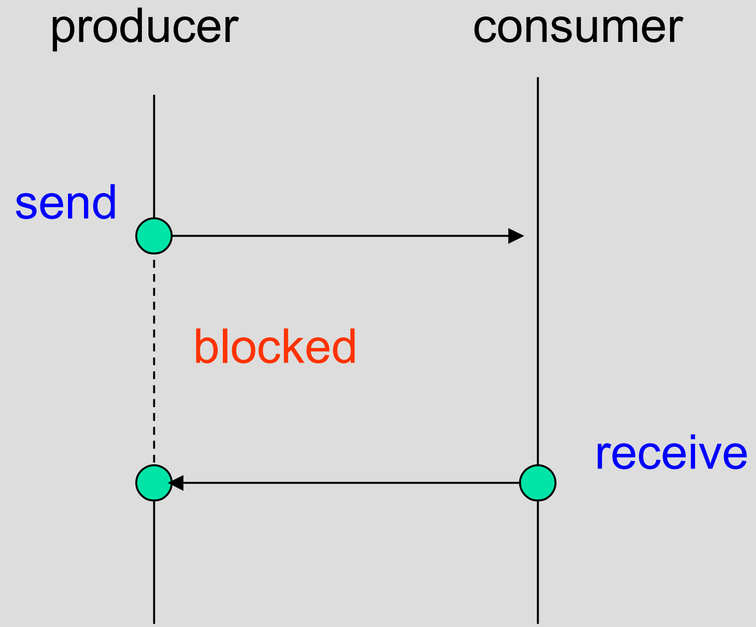
Resources and message passing

- There are no shared resources in the message passing model
  - all the resources are allocated statically, accessed in a dedicated way
- Each resource is handled by a manager process that is the only one that has right to access to a resource
- The consistency of a data structure is guaranteed by the manager process
  - there is no more competition, only cooperation!!!

Synchronous communication

- synchronous send/receive
  - no buffers!

```
producer:
   s_send(consumer, d);

consumer:
   s_receive(producer, &d);
```



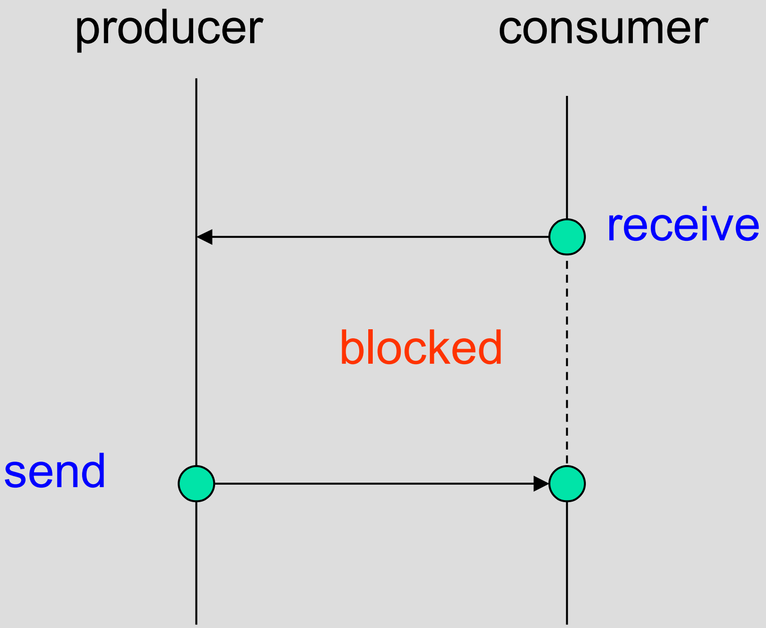
producer

consumer

send

blocked

receive



producer

consumer

receive

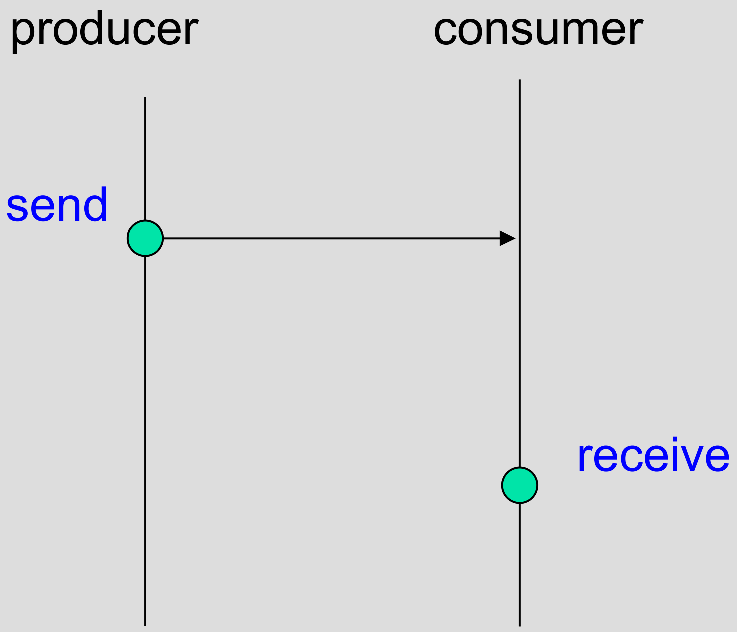
blocked

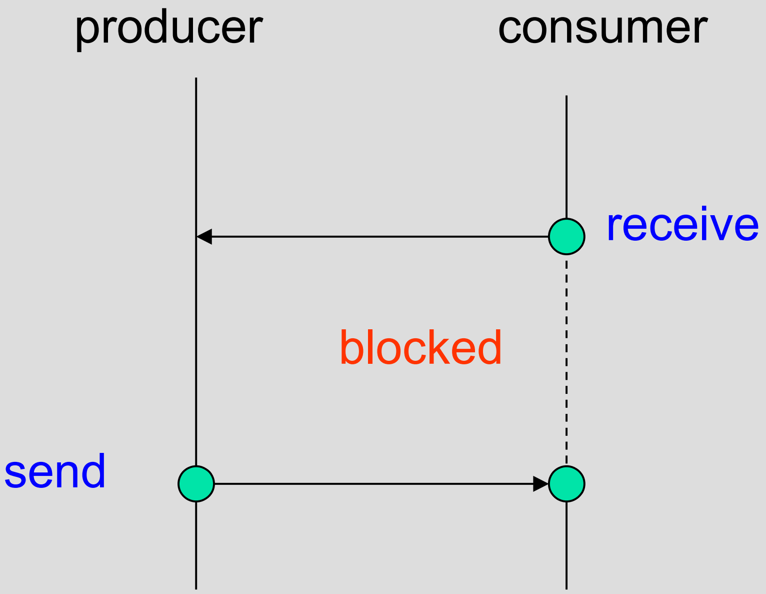
send

Async send/ sync receive

- asynchronous send / synchronous receive
  - there is probably a send buffer somewhere

```
producer:
   a_send(consumer, d);

consumer:
   s_receive(producer, &d);
```



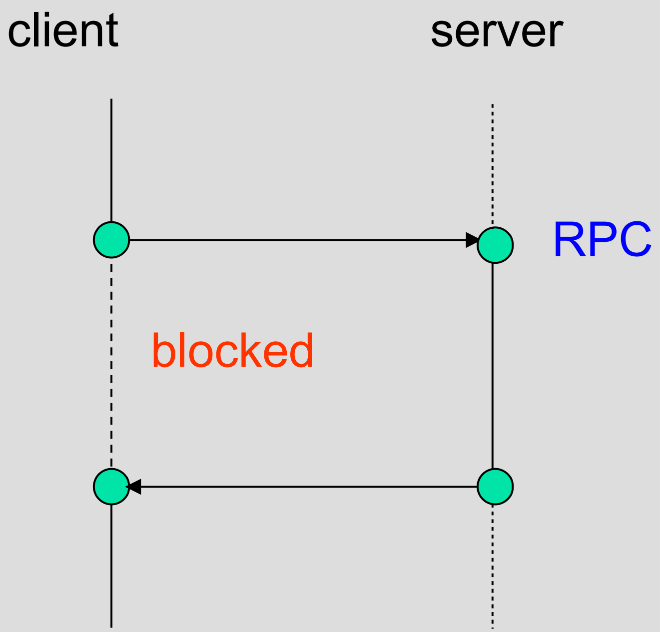


Asymmetric receive

- Symmetric receive
  - receive(source, &data);
  - the programmer wants a message from a given producer
- Asymmetric receive
  - source = receive(&data);
  - often, we do not know who is the sender
    - imagine a web server;
    - the programmer cannot know in advance the address of the browser that will request the service
    - many browsers can ask for the same service

Remote procedure call

- In a client-server system, a client wants to request an action to a server
  - that is typically done using a remote procedure call (RPC)



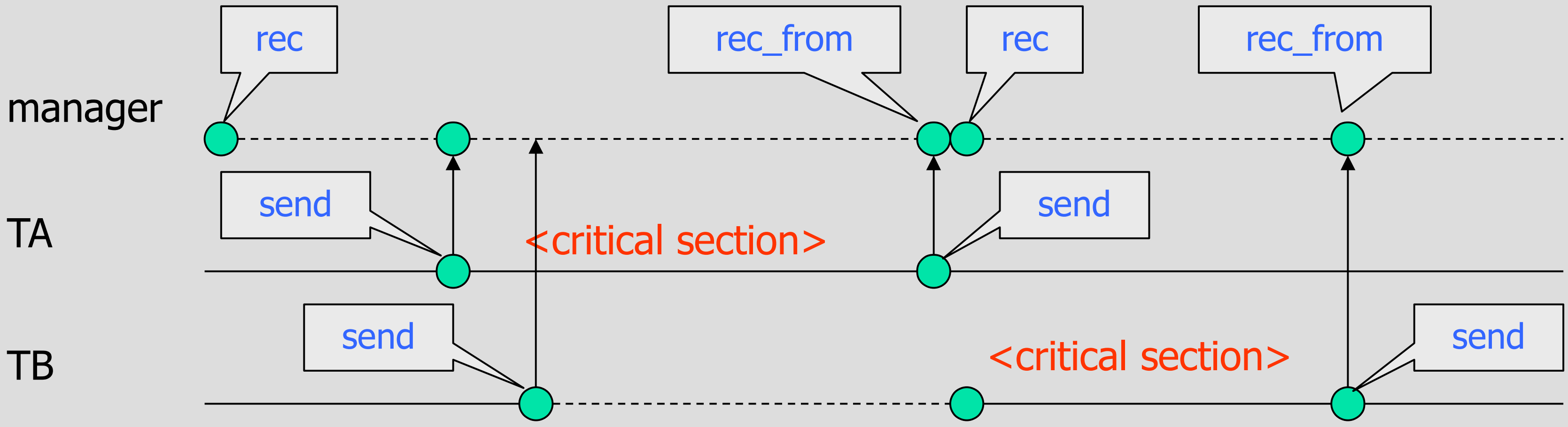
Massage passing systems

- In message passing
  - each resource needs one threads manager
  - the threads manager is responsible for giving access to the resource
- Example: let’s try to implement mutual exclusion with message passing primitives
  - one thread will ensure mutual exclusion
  - every thread that wants to access the resource must
    - send a message to the manager thread
    - access the critical section
    - send a message to signal the leaving of the critical section

Sync send / sync receive

```
void * manager(void *)
{
       thread_t source;
       int d;
       while (true) {
            source = s_receive(&d);
            s_receive_from(source, &d);
       }
}
```

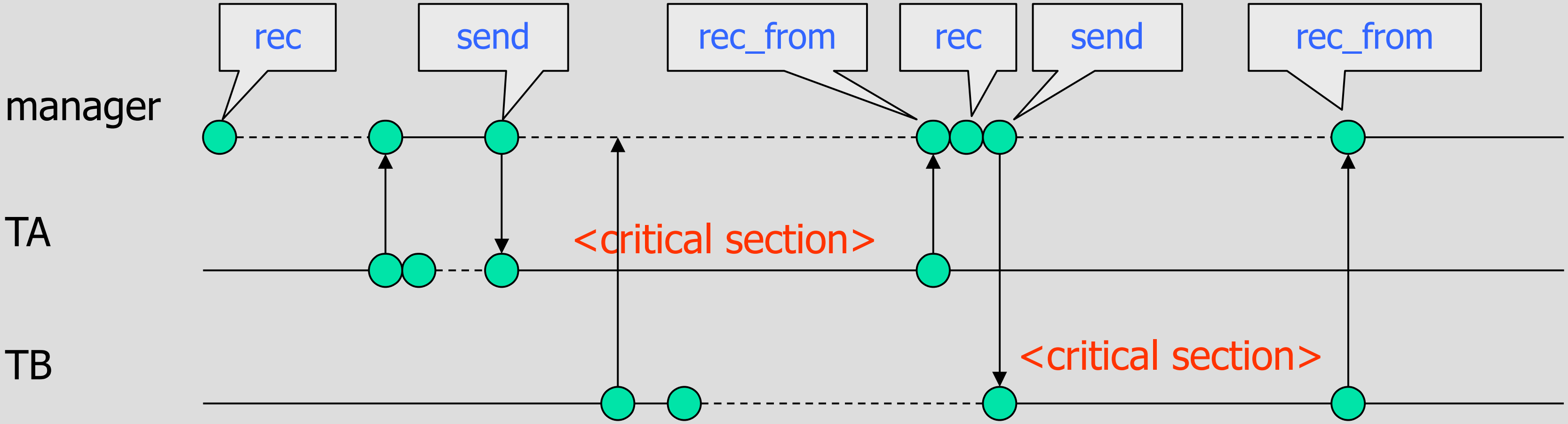
```
void * thread(void *)
{
       int d;
       while (true) {
            s_send(manager, d);
            <critical section>
            s_send(manager, d);
       }
}
```



With async send and sync receive

```
void * manager(void *)
{
      thread_t source;
      int d;
      while (true) {
           source = s_receive(&d);
           a_send(source,d);
           s_receive_from(source,&d);
      }
}
```

```
void * thread(void *)
{
      int d;
      while (true) {
           a_send(manager, d);
           s_receive_from(manager, &d);
           <critical section>
           a_send(manager, d);
      }
}
```



Problem

- Implement readers/writers with message passing
- Hints:
  - define a manager thread
  - the service type (read/write) can be passed as data
  - use asynchronous send and synchronous receive
  - use symmetric and asymmetric receive

Deadlocks

Deadlock and livelock

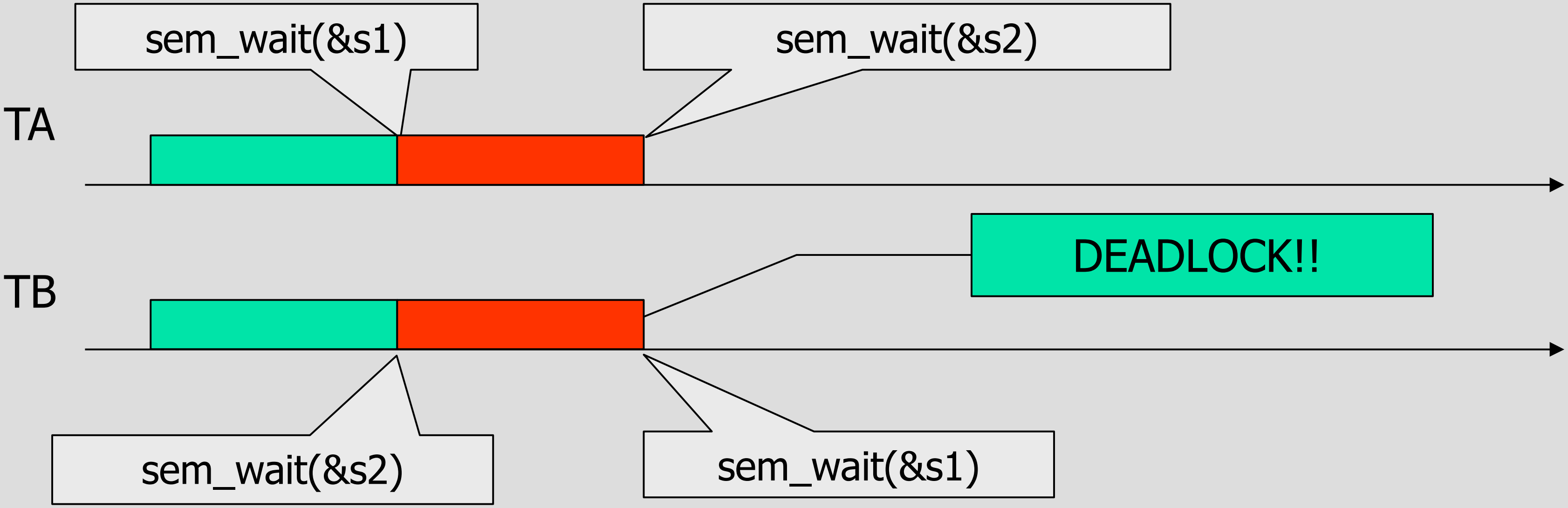
- Deadlock is a situation where a group of threads is permanently blocked waiting for a resource
- Deadlock can happen in many subtle cases
  - example: dining philosophers
- Here we will study ways of avoiding deadlock
- Livelock is the situation where a group of threads tries to get some resource, but they never succeed
  - the idea is that they have a non-blocking wait
  - example: dining philosophers with non-blocking wait
- Deadlocks and livelocks can be total or partial

Example of deadlock

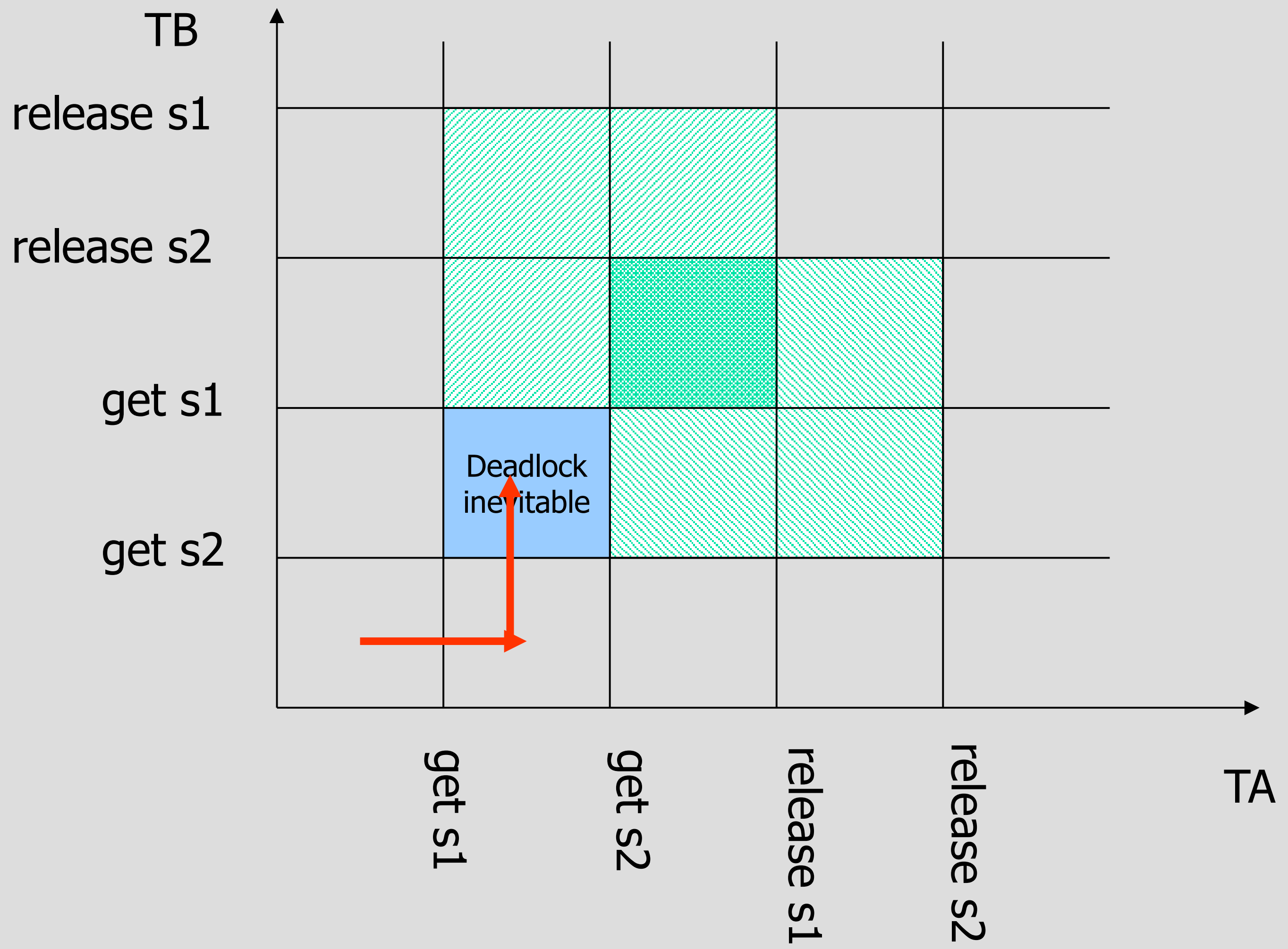
```
Semaphore s1(1);
Semaphore s2(1);
```

```
void *threadA(void *)
{
      ...
      sem_wait(&s1);
      sem_wait(&s2);
      ...
      sem_post(&s2);
      sem_post(&s1);
      ...
}
```

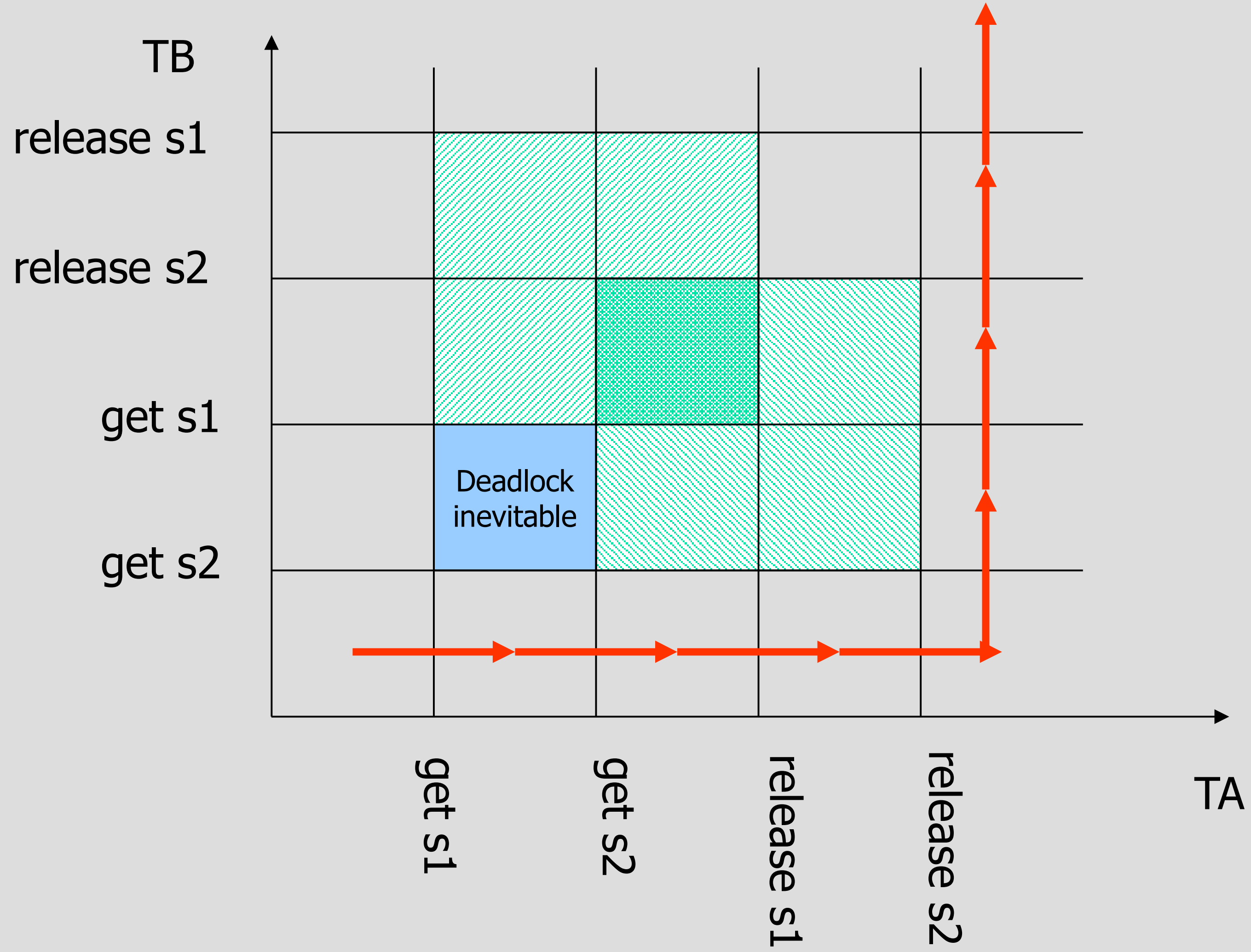
```
void *threadB(void *)
{
      ...
      sem_wait(&s2);
      sem_wait(&s1);
      ...
      sem_post(&s2);
      sem_post(&s1);
      ...
}
```



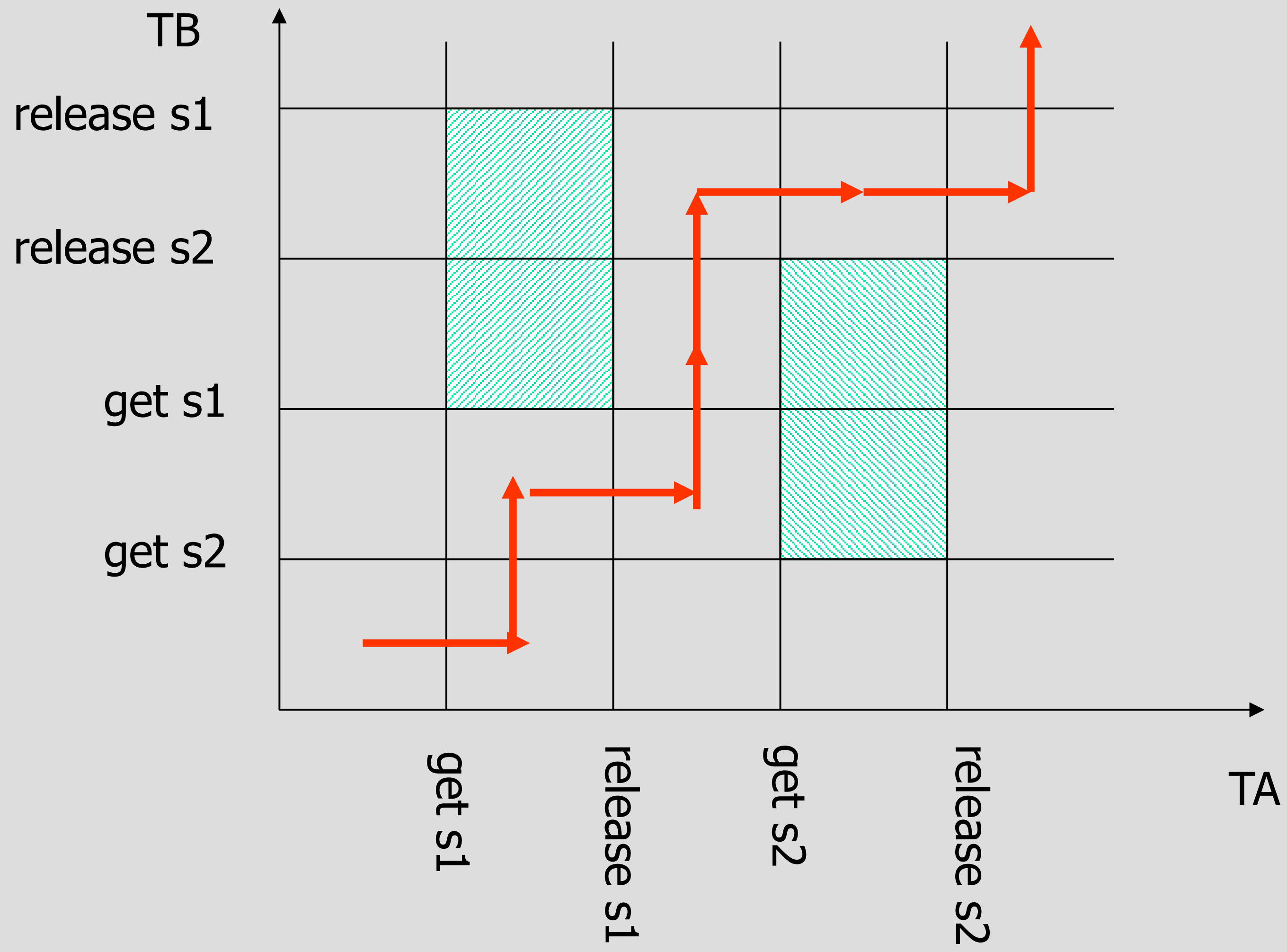
Graphical situation



Graphical situation



Example with no deadlock



Consumable and reusable resources

- Reusable resources
  - it can be safely used by only one thread at time and is not depleted by the use
  - threads must request the resource and later release it, so it can be *reused* by other threads
  - examples are processor, memory, semaphores, etc.
- Consumable resources
  - it is created and destroyed dynamically
  - once the resource is acquired by a thread, it is immediately “destroyed” and cannot be reused
  - examples are messages in a FIFO queue, interrupts, I/O data, etc.

Deadlock with reusable resources

- Bad situations can happen even when the resource is not “on-off”
- Consider a memory allocator
  - suppose that the maximum memory allocable is 200 Kb

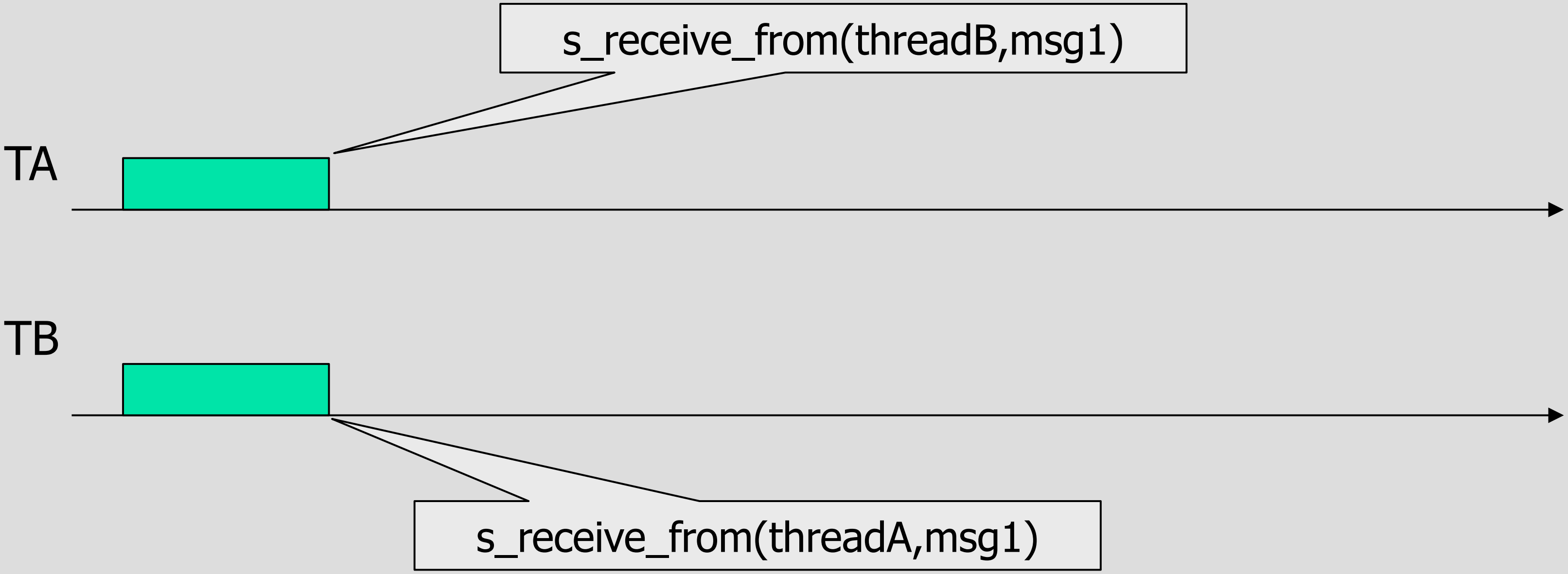
```
void * threadA(void *)
{
    request(80kb);
    ...
    request(60kb);
    ...
    release(140kb);
}
```

```
void * threadB(void *)
{
    request(70kb);
    ...
    request(80kb);
    ...
    release(150kb);
}
```

Deadlock with consumable resources

```
void *threadA(void *)
{
    s_receive_from(threadB, msg1);
    ...
    s_send(threadB, msg2);
    ...
}
```

```
void *threadB(void *)
{
    s_receive_from(threadA, msg1);
    ...
    s_send(threadA, msg2);
    ...
}
```



s_receive_from(threadB,msg1)

TA

TB

s_receive_from(threadA,msg1)

Conditions for deadlock

- Three conditions
  - dynamic allocation of dedicated resources (in mutual exclusion)
    - only one process may use the resource at the same time
  - hold and wait
    - a process may hold allocated resources when it blocks
  - no preemption
    - the resource cannot be revoked
      (note: the CPU is a revokable resource)

Conditions for deadlock

- If the three above conditions hold and
    - circular wait
        - a closed chain of threads exists such that each thread holds at least one resources needed by the next thread in the chain
- Then a deadlock can occur!
- These are necessary and sufficient conditions for a deadlock

How to solve the problem of deadlock

- The basic idea is to avoid that one of the previous conditions hold
- To prevent deadlock from happening we can distinguish two class of techniques
  - static: we impose strict rules in the way resources may be requested so that a deadlock cannot occur
  - dynamic: dynamically, we avoid the system to enter in dangerous situations
- Three strategies
  - deadlock prevention (static)
  - deadlock avoidance (dynamic)
  - deadlock detection (dynamic)

Deadlock prevention: three methods

- Take all the resources at the same time
- Preempt a thread and give the resource to someone else
- Resource allocation in a given order

Deadlock prevention: conditions

- Hold and wait
  - we can impose the tasks to take all resources at the same time with a single operation
  - this is very restrictive! Even if we use the resource for a small interval of time, we must take it at the beginning!
  - reduces concurrency

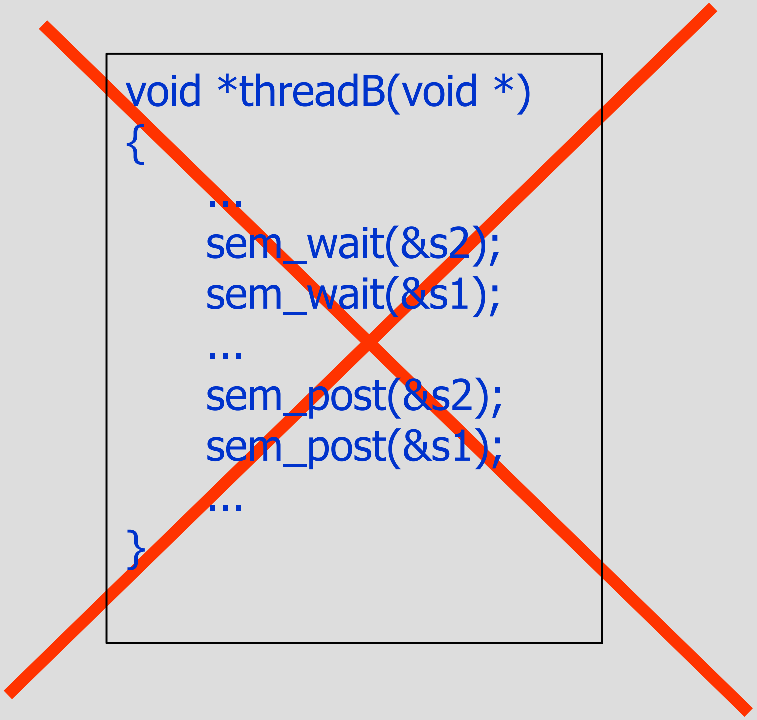
Deadlock prevention: conditions

- No preemption
  - this technique can be done only if we can actually suspend what we are doing on a resource and give it to another thread
  - for the “processor” resource, this is what we do with a thread switch!
  - for other kinds of resources, we should “undo” what we were doing on the resource
  - this may not be possible in many cases!

Deadlock prevention: conditions

- Circular wait
  - This condition can be prevented by defining a linear ordering of the resources
  - for example: we impose that each thread must access resources in a certain well-defined order

```
void *threadA(void *)
{
        ...
        sem_wait(&s1);
        sem_wait(&s2);
        ...
        sem_post(&s2);
        sem_post(&s1);
        ...
}
```



```
void *threadB(void *)
{
        ...
        sem_wait(&s2);
        sem_wait(&s1);
        ...
        sem_post(&s2);
        sem_post(&s1);
        ...
}
```

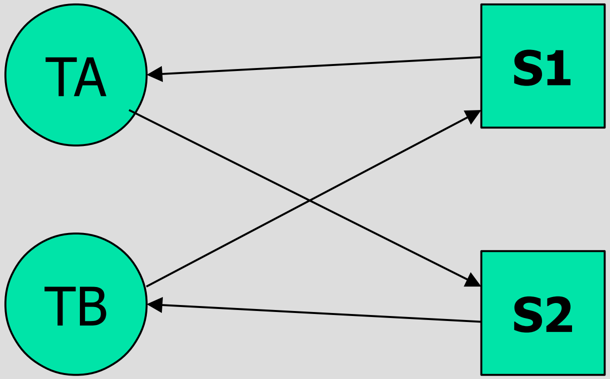
Deadlock prevention: why this strategy works?

- Let us define an oriented graph
  - a vertex can be
    - a thread (round vertex)
    - a resource (square vertex)
  - an arrow from a thread to a resource denotes that the thread requires the resource
  - an arrow from a resource to a thread denotes that the resource is granted to the thread
- Deadlock definition
  - a deadlock happens if at some point in time there is a cycle in the graph

Deadlock prevention: graph

```
void *threadA(void *)
{
      ...
      sem_wait(&s1);
      sem_wait(&s2);
      ...
      sem_post(&s2);
      sem_post(&s1);
      ...
}
```

```
void *threadB(void *)
{
      ...
      sem_wait(&s2);
      sem_wait(&s1);
      ...
      sem_post(&s2);
      sem_post(&s1);
      ...
}
```



Deadlock prevention: theorem

- If all threads access resources in a given order, a deadlock cannot occur
- Proof (by contradiction):
  - suppose a deadlock occurs. Then, there is a cycle
  - by hypothesis all threads access resources in order
  - each thread is blocked on a resource that has an order number grater than the resources it holds
  - starting from a thread and following the cycle, the order number of the resource increases. However, since there is a cycle, we go back to the first thread. Then there must be a thread T that holds a resource Ra and requests a Resource Rb with Ra < Rb
  - this is a contradiction!

Deadlock avoidance

- This technique consists in monitoring the system to avoid deadlock
  - we check the behaviour of the system
  - if we see that we are going into a dangerous situation, we block the thread that is doing the request, even if the resource is free
  - that algorithm is called the Banker's algorithm
    - we skip it :-)

Deadlock detection

- In this strategy, we monitor the system to check for deadlocks *after* they happen
  - we look for cycles between threads and resources
  - how often should we look?
    - it is a complex thing to do, that takes processing time
    - a good point to do that is when we lock (but it is computationally expensive)
  - once we discover deadlock, we must *recover*
- The idea is to
  - kill some blocked thread
  - return an error in the wait statement if there is a cycle
    - that is the POSIX approach

Recovery strategies

1. Abort all threads
   - used in almost all OS: the simplest thing to do.
2. Check point
   - all threads define safe *check points:* when the OS discovers a deadlock, all involved threads are restarted to a previous check point
     - Problem: they can go in the same deadlock again!
3. Abort one thread at time
   - threads are aborted one after the other until deadlock disappears
4. Successively preempt resources
   - preempt resources one at time until the deadlock disappears