Concurrency

Paolo Burgio
paolo.burgio@unimore.it

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA
High Performance Real Time Lab


FRATELLI
QUANDO LA MAMMA ESCE DI CASA
QUANDO TORNA A CASA

Why concurrency?

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

Performance

- Take advantage of blocking time
- While some thread waits for a blocking condition, another thread performs another operation
- On a multi-core machine, independent activities can be carried out on different cores are the same time

Competitive vs. Cooperative

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

Competitive

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

Client/server model

A server manages the resource **exclusively**

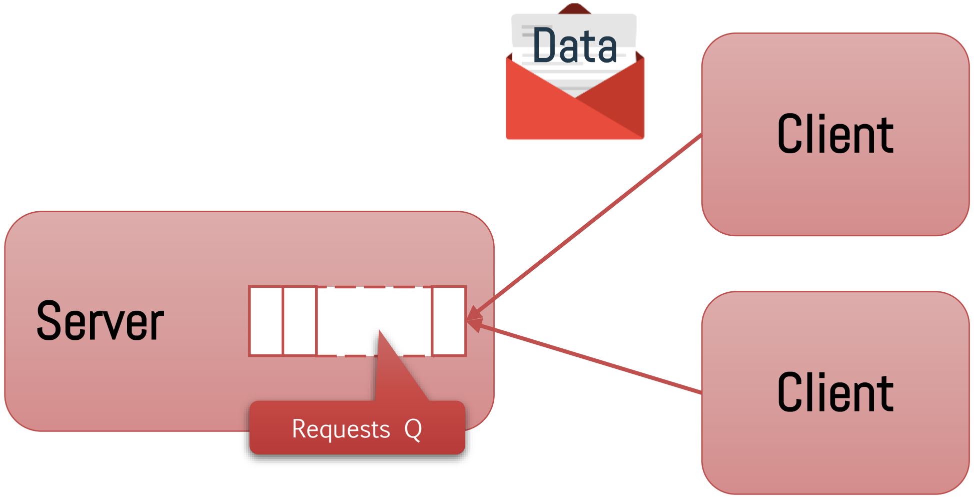
- For example, the printer

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**



Cooperative model

Cooperative activities **know** about each other

› They do not need memory protection (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

Competition vs. cooperation

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

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

- RTOS supports only shared memory

Models of concurrency

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 *(fire-and-forget)*

```
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

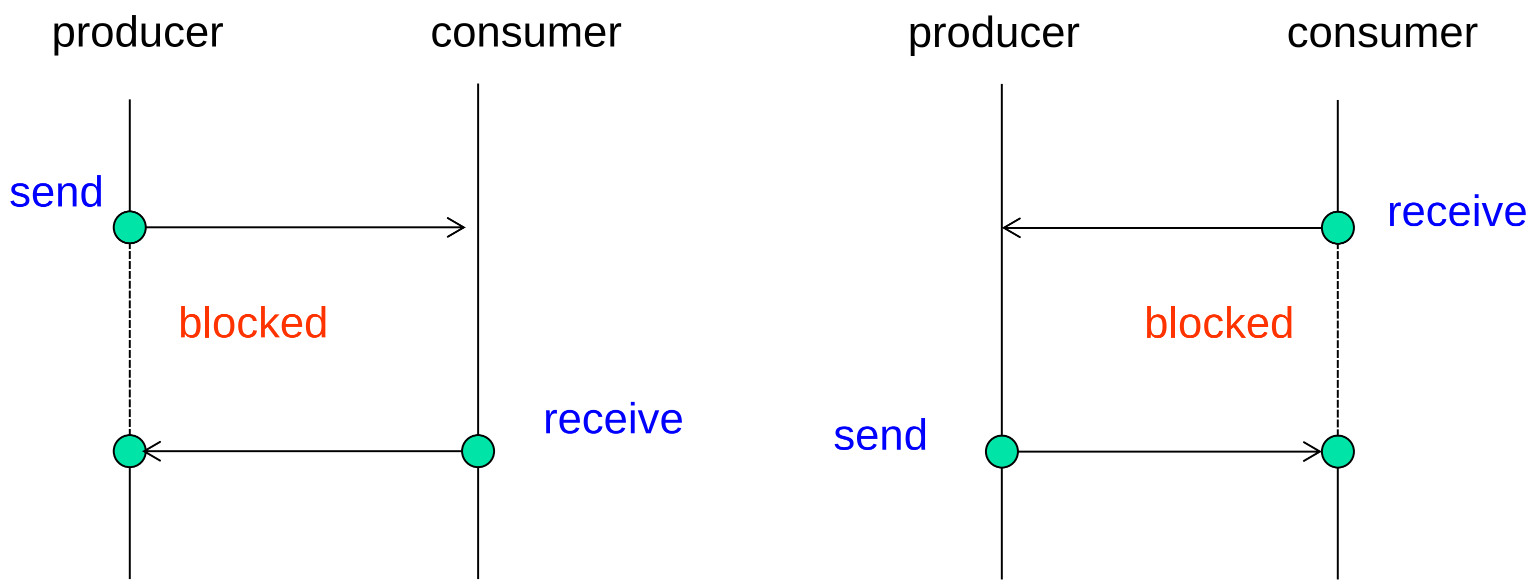
✓ no buffers!

producer:

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

consumer:

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



Async send/ sync receive

Asynchronous send / synchronous receive

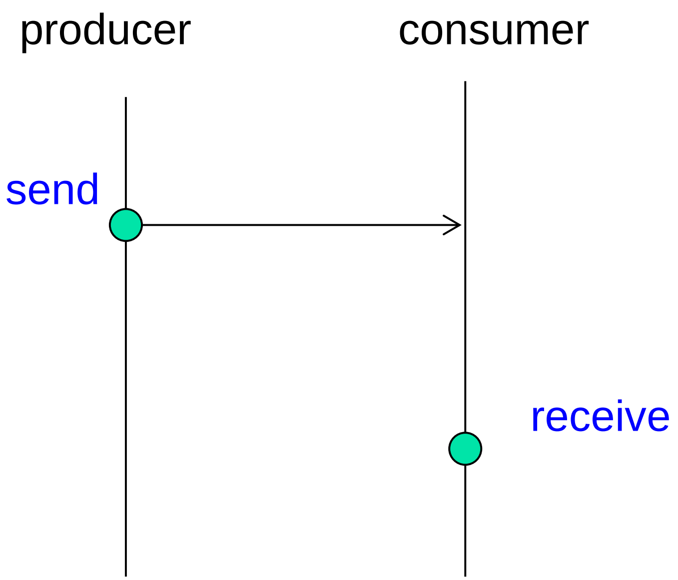
✓ there is probably a send buffer somewhere

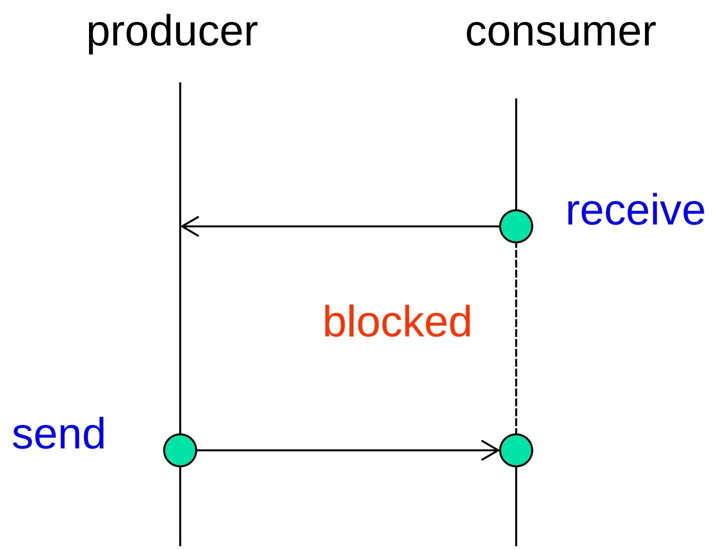
producer:

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

consumer:

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





(A)symmetric 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

E.g., a web server is asymmetric

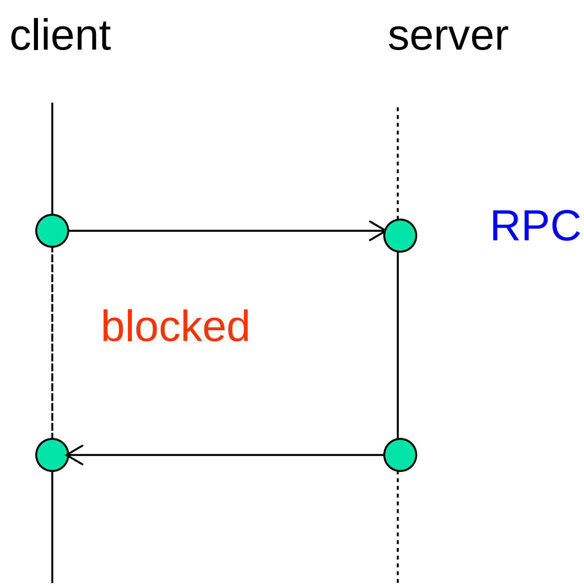
✓ the programmer cannot know in advance the address of the browser that will request the service

✓ many browsers can ask for the same service

Remote procedure call

From low-level (MP) to more «programmer friendly» (RPC) mechanism

- ✓ Increase expressiveness, higher level of abstraction
- ✓ In a client-server system, a client wants to request an action to a server
- ✓ Typically done using a remote procedure call (RPC)



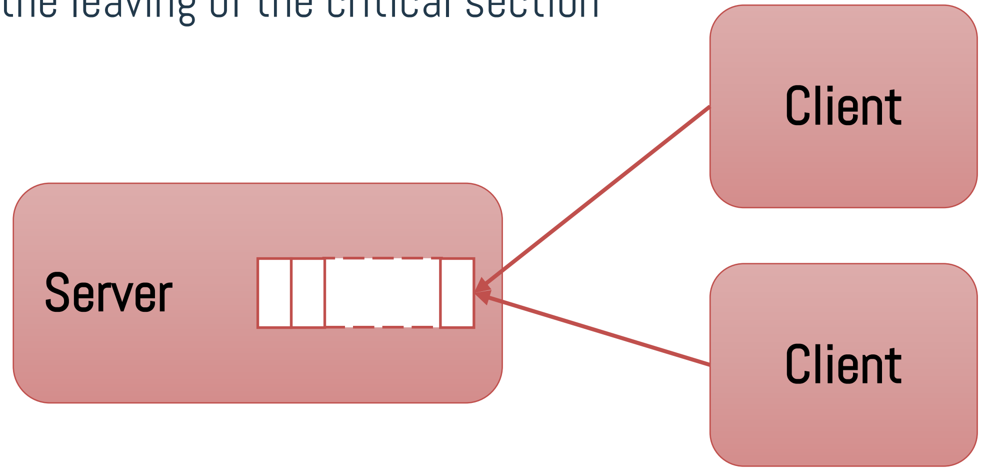
Massage passing systems

In message passing

- ✓ each resource needs one threads manager (often called **daemon thread**)
- ✓ the threads manager is responsible for giving access to the resource

Example: 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



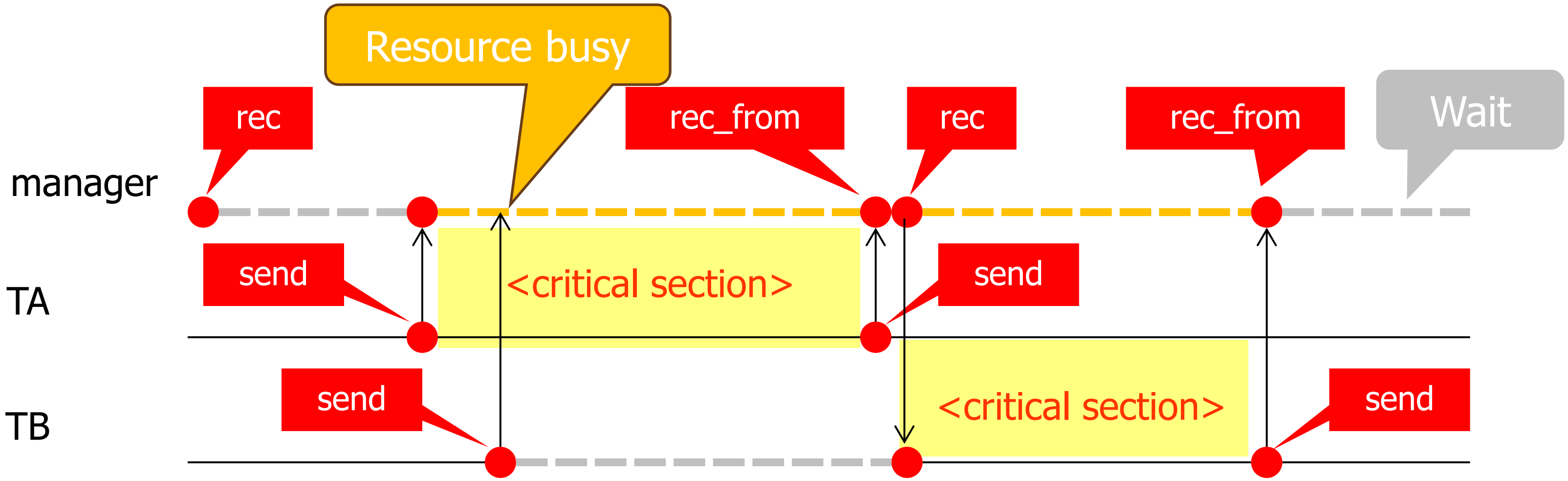
Sync send / sync receive

```
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);
  }
}
```

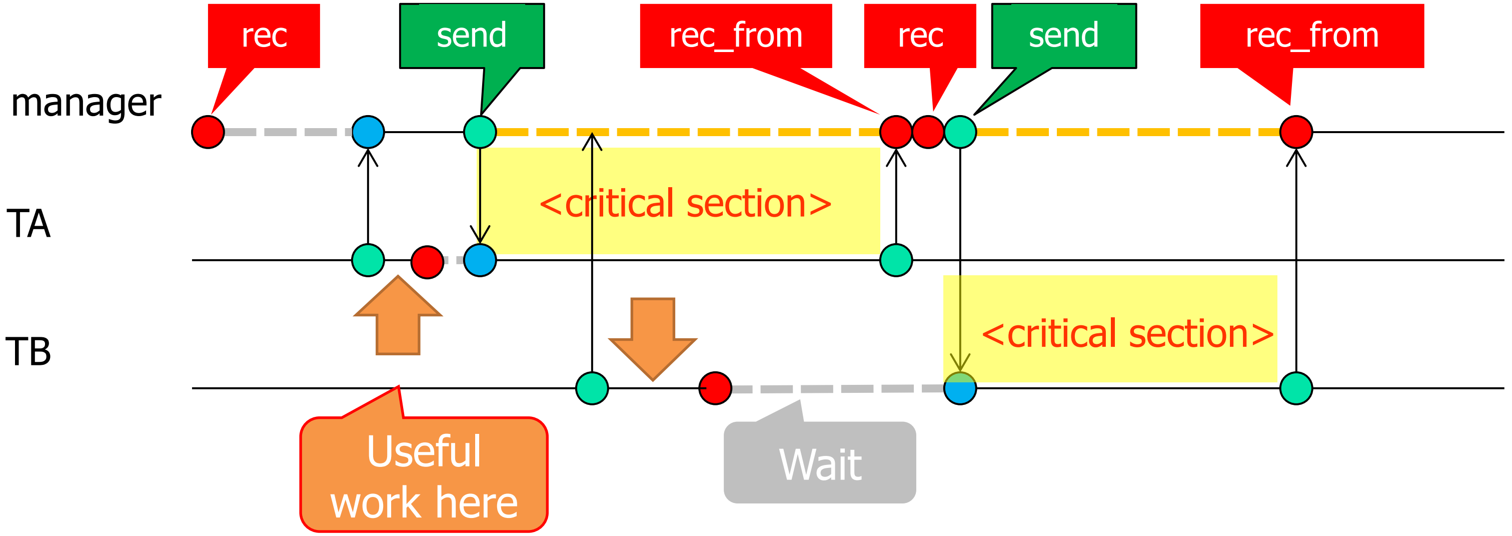
Blocking

```
void * thread(void *) {
  int d;
  while (true) {
    a_send (manager, d);
    s_receive_from (manager, d);

    <critical section>

    a_send (manager, d);
  }
}
```

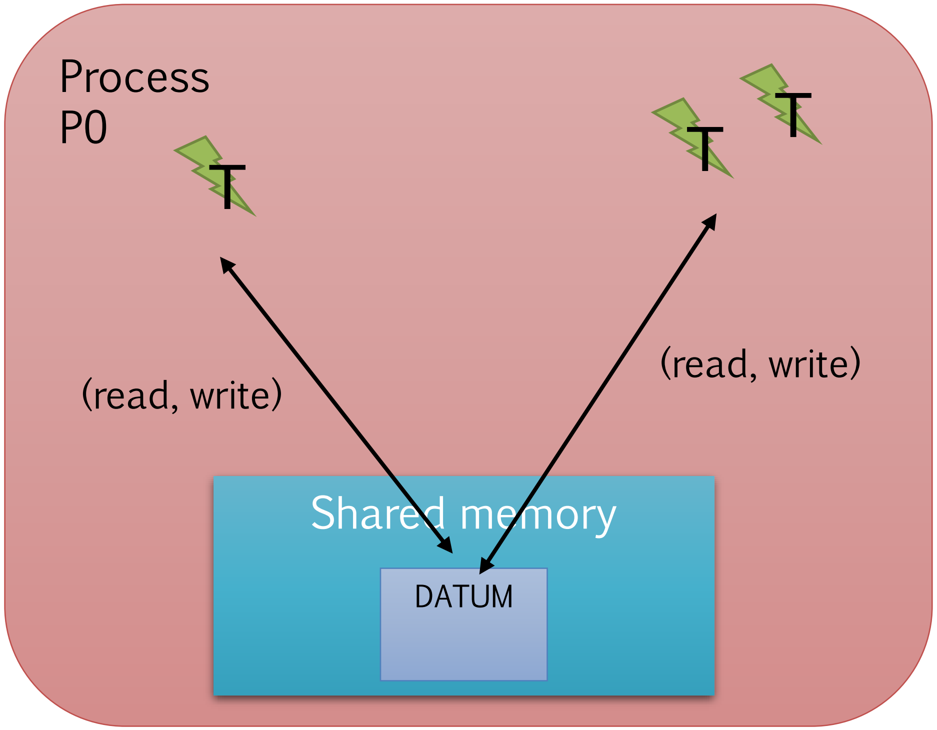
Non blocking



Shared memory model

Shared memory abstraction

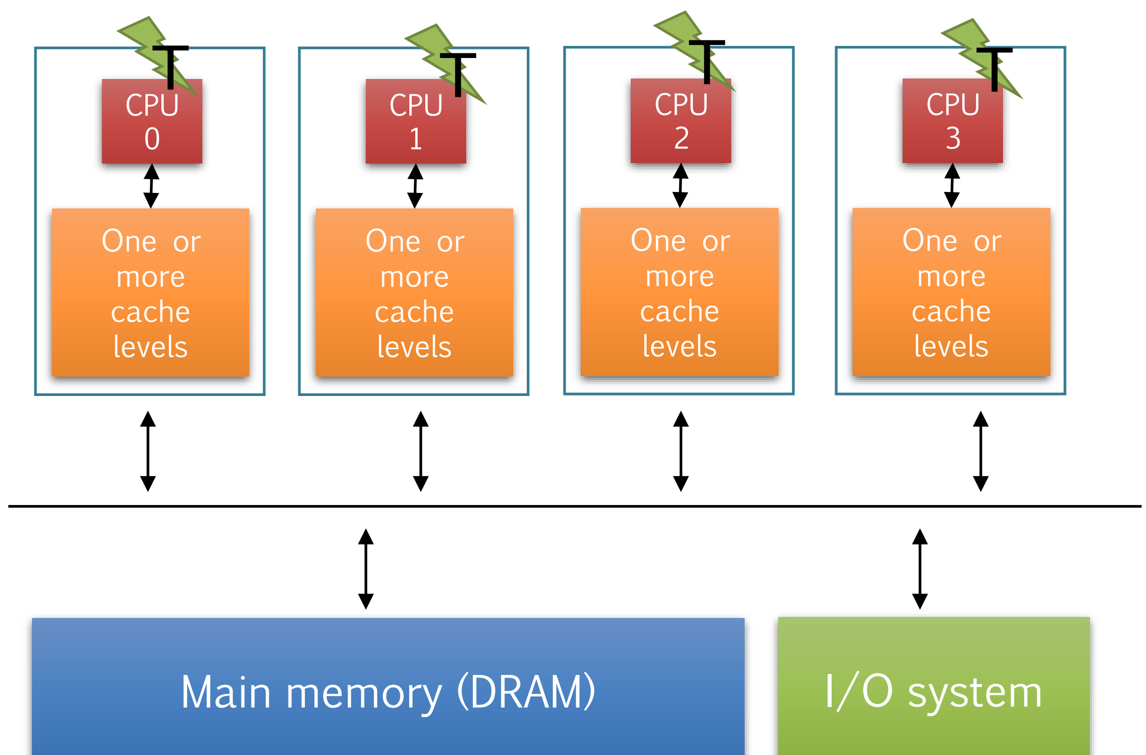
- ✓ The first one being supported in old OS
- ✓ The simplest one and the closest to the machine
- ✓ All threads can access the same memory locations



Analogy with hardware

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: a (big) 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

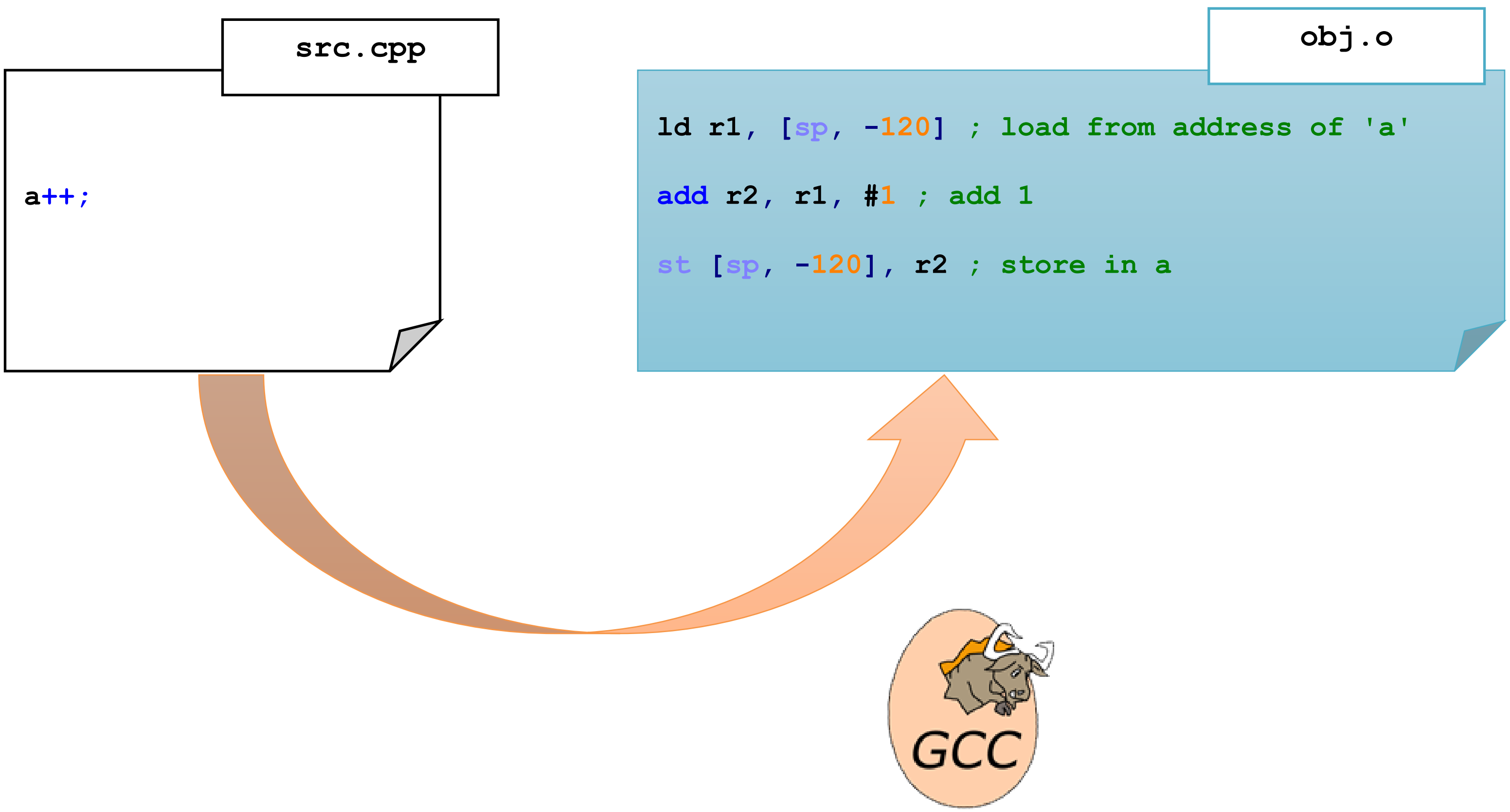
- ✓ They are <u>thread safe</u> operations
  - For example, transferring one word from memory to register or viceversa

Non atomic operations can be interrupted

- ✓ They are not “safe” operations
- ✓ Non-elementary operations are not atomic

The big problem

- a++ is <u>not</u> an unique instruction



When things go well

a++;

11

```
ld r1, [sp, -120]
add r1, r1, #1
st [sp, -120] r1
```

12

a++;

12

```
ld r1, [sp, -120]
add r1, r1, #1
st [sp, -120] r1
```

13

t

When things go less well

a++;

11

```
ld r1, [sp, -120]
add r1, r1, #1
```

a++;

11

```
ld r1, [sp, -120]
add r1, r1, #1
st [sp, -120] r1
```

12

```
st [sp, -120] r1
```

t

12

Critical sections

Definitions

- ✓ The shared object (e.g., *x*) 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

Multiple critical sections on the same resource must execute in **MUTUAL EXCLUSION**

- ✓ atomic operation (hardware extension within the memory banks)
- ✓ semaphores
- ✓ mutexes
- ✓ *locks*

General mechanism: semaphores

Proposed by Djikstra

A semaphore is an abstract entity that consists of
- ✓ A counter
- ✓ A blocking queue (of threads)

Can perform two atomic operations
- ✓ Blocking **Wait** for a given condition
- ✓ **Signal** that the condition becomes true (aka **Post**)

We can also use them to implement mutual exclusion (we'll see this)

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 **Signal (aka: 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 (pseudo-code)

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

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

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

- We assume that `s->count` (and its operation `++`) are atomic

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 **signal/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

semaphore

counter 0

(TA) sem_wait();
(TA) <critical section (1)>

sem_wait() (TB)

(TA) <critical section (2)>
(TA) sem_post()

<critical section> (TB)
sem_post() (TB)

t

Synchronisation

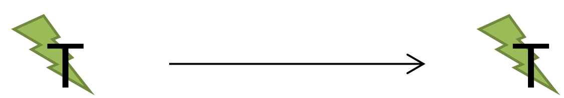
Mutual exclusion is not the only problem: we need a way of synchronise two or more threads

✓ Example: producer/consumer

We have two threads,

✓ One produces some integers and sends them to another thread (**PRODUCER**)

✓ Another one takes the integer and elaborates it (**CONSUMER**)



Synchronization with semaphores

Define a semaphore initialized to 0 (**blocked**)

- ✓ At the synchronization point, consumer performs a **Wait**
- ✓ At the synchronization point, producer performs a **Signal/Post**
- ✓ In the example, *threadA* blocks until *threadB* wakes it up

```
sem_t s;

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

```
void *threadA(void *arg) {

  sem_wait (&s);

  process (a);
}
```

<<unlock>>

```
void *threadB(void *arg) {

  a = 11;

  sem_post (&s);
}
```

Producer/consumer: how to do it naively

Share a queue of data/objects/anything you might need to produce&consume

- ✓ If the queue is full, the producer actively waits
- ✓ If the queue is empty, the consumer actively waits
- ✓ Aka: **busy-waiting**

Very inefficient!

```
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>
  }
}
```

Naive, polling-based producer/consumer

Consider a producer/consumer system

Producer(s) execute `insert_CA()`

- ✓ We want the producers to be blocked when the queue is full
- ✓ The producers will be unblocked when there is some space again

Consumer(s) execute `extract_CA()`

- ✓ We want the consumers to be blocked when the queue is empty
- ✓ The consumers will be unblocked when there is some space again
- ✓ First attempt: one producer and one consumer only

One producer, one consumer

```
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);
}
```

Block if queue is full

Release those who're waiting for extraction

Block if queue is empty

Release those who're waiting for insertion

Multiple producers/consumers

Combine mutual exclusion and synchronization

- ✓ Semaphore to implement synchronization
- ✓ Semaphore to protect the data structure
  - ✓ Make the primitives (`insert_CA`, `extract_CA`) <u>THREAD-SAFE</u>

Producers/consumers: does this 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);
}
```

Enter critical section

Exit critical section

```
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);
}
```

...of course NOT!

- Why? (red => protects critical section/mutual exclusion;
  other colors => synchronization - producer/consumer)

The deadlock explained

- ✓ 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!!!

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);
}
```

...we were mixing producer/consumer and synchronizastion

- NEVER NEST MUTEXES

References

Course website

- http://hipert.unimore.it/people/paolob/pub/Industrial_Informatics/index.html

My contacts

- paolo.burgio@unimore.it
- http://hipert.mat.unimore.it/people/paolob/

Resources

- Giorgio Buttazzo, “Hard Real-Time Computing Systems : Predictable Scheduling Algorithms and Applications“. 3rd Edition. 2011. Springer
- “Real-Time Embedded Systems” course by Prof. Bertogna @UNIMORE
- A "small blog"
  - http://www.google.com