Communication and Synchronization between tasks

Juan José Costa and Alejandro Pajuelo

Escola Tècnica Superior de Telecomunicacions de Barcelona (ETSETB)

Universitat Politècnica de Catalunya (UPC)

2021-2022Q2

Index

- Why communicate and synchronize?
- Shared memory
- Distributed memory

Introduction

- A single process is easy, but no practical
- More than one process is desirable
  - To cooperate in some problem
  - To accelerate some computation
  - For modularity (different roles)
- But then new needs arise:
  - How do we pass info from one to the other?
  - What if we want to execute one before the other?

Introduction

- We need ways to:
  - share information
  - synchronize processes
- We will see different ways to communicate processes
  - Inter Process Communication (IPC)
- There are 2 communication models:
  - Shared memory
    - Processes use some variables that can be read/written
  - Message passing
    - Processes use specific functions to send/receive data

Index

- Why communicate and synchronize?
- Shared memory
- Distributed memory

Shared Memory

- Introduction
- Problems
  - Critical Region
  - Serialization
- Mutual exclusion
  - Busy-waiting
  - Token [Harder18]sec 9.3 [sync]
  - Test-Set [Harder18]sec 9.4
  - Mutex [Harder18]sec 9.5.1
  - Semaphores [Harder18]sec 9.5 [sync]

Introduction

- A single machine has the following resources:
  - memory
  - secondary memory
  - network
- They may be shared by different tasks

Introduction: Example

```
int first = 1; /* GLOBAL VAR */


int f() {
  if (first) {
    first --;
    task1();
  else {
    task2();
  }
}
```

Introduction: Example

```
int first = 1; /* GLOBAL VAR */


int f() {
  if (first) {
    first --;
    task1();
  else {
    task2();
  }
}
```

With a single task we are fine ☺

- No sharing at all

Introduction: Example

```
int first = 1; /* GLOBAL VAR == SHARED */
create_2_threads( f );

int f() {
  if (first) {
    first --;
    task1();
  else {
    task2();
  }
}
```

Introduction: Example

```
int first = 1; /* GLOBAL VAR == SHARED */
create_2_threads( f );
```

1st thread

```
int f() {
  if (first) {
    first --;
    task1();
  else {
    task2();
  }
}
```

2nd thread

```
int f() {
  if (first) {
    first --;
    task1();
  else {
    task2();
  }
}
```

Possible outcomes?

Introduction: Example

```
int first = 1; /* GLOBAL VAR == SHARED */
create_2_threads( f );
```

1st thread

```
int f() {
  if (first) {
    first --;
    task1();
  else {
    task2();
  }
}
```

2nd thread

```
int f() {
  if (first) {
    first --;
    task1();
  else {
    task2();
  }
}
```

Possible outcomes?

task1()	task2()
1st thread	2nd thread

Introduction: Example

```
int first = 1; /* GLOBAL VAR == SHARED */
create_2_threads( f );
```

1st thread

```
int f() {
  if (first) {
    first --;
    task1();
  else {
    task2();
  }
}
```

2nd thread

```
int f() {
  if (first) {
    first --;
    task1();
  else {
    task2();
  }
}
```

Possible outcomes?

task1()	task2()
1st thread	2nd thread
2nd thread	1st thread

Introduction: Example

```
int first = 1; /* GLOBAL VAR == SHARED */
create_2_threads( f );
```

1st thread

```
int f() {
  if (first) {
    first --;
    task1();
  else {
    task2();
  }
}
```

2nd thread

```
int f() {
  if (first) {
    first --;
    task1();
  else {
    task2();
  }
}
```

Possible outcomes?

task1()	task2()
1st thread	2nd thread
2nd thread	1st thread
1st and 2nd	

Problem

- Shared variable
  - `first`
- Non atomic operations:
  - LOAD
  - SUB
  - STORE
- **Concurrent accesses** to **shared resources** can lead to unexpected or erroneous behavior
  - Known as *race conditions*
- Another example: [Harder18] Section 9.1.1

Critical Region

- **Section of a program** where a shared resource is accessed concurrently and needs to be protected
- This section is the **critical region.**
- A mechanism is needed to ensure that it can only be executed by a single task at a time.
  - **Mutual exclusion**

Two tasks communicating information

- Two iterative tasks trying to share information.
- Task 1 (*Producer*) prepares data
  - which is to be sent to and used by Task 2.
- Data should not be overwritten by Task 1
  - until Task 2 has completed using it.
- Task 2 (*Consumer*) should not access data
  - until Task 1 has finished writing to it.
- (Courtesy of Harder18 Sec. 9.1)

Two tasks communicating information

```
Data *result;
// Producer
void task_1( void ) {
  while ( 1 ) {
   // Executing...
   // Prepare something for
   // task 2 and assign to result

   write( result );

   // Continue executing...
  }
}
```

```
// Consumer
void task_2( void ) {
  while( 1 ) {
   // Executing...

   read( result );

   // Continue executing...
  }
}
```

Two tasks communicating information: 1st try

```
#include <stdbool.h>
bool result_is_produced = false;
Data *result;
// Producer
void task_1( void ) {
  while ( 1 ) {
   // Executing...
   // Prepare something for
   // task 2 and assign to result

   write( result );
   result_is_produced = true;
   // Continue executing...
  }
}
```

Add a flag

```
// Consumer
void task_2( void ) {
  while( 1 ) {
   // Executing...
   while( !result_is_produced );
   read( result );

   // Continue executing...
  }
}
```

Two tasks communicating information: 1st try

```
#include <stdbool.h>
bool result_is_produced = false;
Data *result;
// Producer
void task_1( void ) {
  while ( 1 ) {
   // Executing...
   // Prepare something for
   // task 2 and assign to result
   while( result_is_produced );
   write( result );
   result_is_produced = true;
   // Continue executing...
  }
}
```

Add a flag

```
// Consumer
void task_2( void ) {
  while( 1 ) {
   // Executing...
   while( !result_is_produced );
   read( result );
   result_is_produced = false;
   // Continue executing...
  }
}
```

Two tasks communicating information: 1st try

Different case as before
(both codes are mutually exclusive)

Add a flag

```
#include <stdbool.h>
bool result_is_produced = false;
Data *result;
// Producer
void task_1( void ) {
  while ( 1 ) {
   // Executing...
   // Prepare something for
   // task_2 and assign to result
   while( result_is_produced );
   write( result );
   result_is_produced = true;
   // Continue executing...
  }
}
```

```
// Consumer
void task_2( void ) {
  while( 1 ) {
   // Executing...
   while( !result_is_produced );
   read( result );
   result_is_produced = false;
   // Continue executing...
  }
}
```

Two tasks communicating information: 1st try

```
#include <stdbool.h>
bool result_is_produced = false;
Data *result;
// Producer
void task_1( void ) {
  while ( 1 ) {
   // Executing...
   // Prepare something for
   // task 2 and assign to result
   while( result_is_produced );
   write( result );
   result_is_produced = true;
   // Continue executing...
  }
}
```

```
// Consumer
void task_2( void ) {
  while( 1 ) {
   // Executing...
   while( !result_is_produced );
   read( result );
   result_is_produced = false;
   // Continue executing...
  }
}
```

What happens in a single-processor machine?

Two tasks communicating information: 1st try

```
#include <stdbool.h>
bool result_is_produced = false;
Data *result;
// Producer
void task_1( void ) {
  while ( 1 ) {
   // Executing...
   // Prepare something for
   // task 2 and assign to result
   while( result_is_produced );
   write( result );
   result_is_produced = true;
   // Continue executing...
  }
}
```

```
// Consumer
void task_2( void ) {
  while( 1 ) {
   // Executing...
   while( !result_is_produced );
   read( result );
   result_is_produced = false;
   // Continue executing...
  }
}
```

What happens in a single-processor machine? **Busy Waiting**

Two tasks communicating information: 2nd try

```
#include <stdbool.h>
bool result_is_produced = false;
Data *result;
// Producer
void task_1( void ) {
  while ( 1 ) {
   // Executing...
   // Prepare something for
   // task 2 and assign to result
   while( result_is_produced ) {
     pthread_yield();
   }
   write( result );
   result_is_produced = true;
   // Continue executing...
  }
}
```

```
// Consumer
void task_2( void ) {
  while( 1 ) {
   // Executing...
   while( !result_is_produced){
     pthread_yield();
   }
   read( result );
   result_is_produced = false;
   // Continue executing...
  }
}
```

Two tasks communicating information: 2nd try

```
#include <stdbool.h>
bool result_is_produced = false;
Data *result;
// Producer
void task_1( void ) {
  while ( 1 ) {
   // Executing...
   // Prepare something for
   // task 2 and assign to result
   while( result_is_produced ) {
     pthread_yield();
   }
   write( result );
   result_is_produced = true;
   // Continue executing...
  }
}
```

```
// Consumer
void task_2( void ) {
  while( 1 ) {
   // Executing...
   while( !result_is_produced){
     pthread_yield();
   }
   read( result );
   result_is_produced = false;
   // Continue executing...
  }
}
```

Round Robin… ok, but, priorities?…

Multiple tasks communicating information

- Imagine 2 copies of Task2
  - they can NOT access data structure at same time

Multiple tasks communicating information

```
#include <stdbool.h>
bool result_is_produced = false;
bool consumer_is_reading = false;
Data *result;
// Producer
void task_1( void ) {
  while ( 1 ) {
   // Executing...
   // Prepare something for
   // task 2 and assign to result
   while( result_is_produced ) {
     pthread_yield();
   }
   write( result );
   result_is_produced = true;
   // Continue executing...
  }
}
```

```
// Consumer
void task_2( void ) {
  while( 1 ) {
   // Executing...
   while( !result_is_produced
       || consumer_is_reading){
     pthread_yield();
   }
  consumer_is_reading = true;
  read( result );
  result_is_produced = false;
  consumer_is_reading = false;
  // Continue executing...
  }
}
```

Multiple tasks communicating information

Sharing problem! ☹

```
#include <stdbool.h>
bool result_is_produced = false;
bool consumer_is_reading = false;
Data *result;
// Producer
void task_1( void ) {
  while ( 1 ) {
   // Executing...
   // Prepare something for
   // task 2 and assign to result
   while( result_is_produced ) {
     pthread_yield();
   }
   write( result );
   result_is_produced = true;
   // Continue executing...
  }
}
```

```
// Consumer
void task_2( void ) {
  while( 1 ) {
   // Executing...
   while( !result_is_produced
       || consumer_is_reading){
     pthread_yield();
   }
  consumer_is_reading = true;
  read( result );
  result_is_produced = false;
  consumer_is_reading = false;
  // Continue executing...
  }
}
```

Summary

- We have seen 2 different problems:
  1. **Mutual exclusion**: preventing more than one task from accessing data, and
  2. **Serialization**: ensuring that one task is performed after another.

Shared Memory

- Introduction
- Problems
  - Critical Region
  - Serialization
- Mutual exclusion
  - Busy-waiting
  - Token [Harder18]sec 9.3 [sync]
  - Test-Set [Harder18]sec 9.4
  - Mutex [Harder18]sec 9.5.1
  - Semaphores [Harder18]sec 9.5 [sync]

Mutual Exclusion

- Protection mechanism for a critical region
- Ensure that no other thread may execute the region while a thread is inside
  - Even in presence of context switches!

Mutual Exclusion: Token passing

- A simple way: force thread execution order
- Token passing mechanism
  - Identify each consumer
  - Generate a token with one of the identifiers
  - When a consumer has accessed the result
    - → update token to the identifier of the next consumer

Token passing example

- Define token structure:

```
typedef struct {
  size_t this_id;
  size_t next_id;
} token_t;
```

- Establish the order (3 tasks):

```
token_t args[3] = {{1, 2}, {2, 3}, {3, 1}};
// 1->2->3->1->2->…
```

- Initalize the starting identifier task and tasks:

```
size_t reading = 1;

pthread_create( &thread_id[0], NULL, &consumer, &args[0] );
pthread_create( &thread_id[1], NULL, &consumer, &args[1] );
pthread_create( &thread_id[2], NULL, &consumer, &args[2] );
```

Token passing example

- Then the consumer becomes:

```
void *consumer( void *p_arguments ) {
  token_t *p_identifier = (token_t *)p_arguments;
  while ( 1 ) {
    while ( ( reading != p_identifier->this_id ) ||
              !result_is_produced ) {
      pthread_yield();
    }
    --result;
    printf( "%d ", result );
    result_is_produced = false;
    reading = p_identifier->next_id;
  }
}
```

Mutual Exclusion

- Allow any order
- Programmer needs to mark the start and end of the critical region:
  - **start**: if there is no thread in the critical region, enter; otherwise wait until the working thread finishes
  - **end**: release access to the critical region; if any other thread was waiting, allow it to enter

Mutual exclusion: 1st try

```
bool ready = true; //Global variable

// Start Critical Region
while ( !ready ) {
  scheduler();
}
ready = false;
// Inside the critical region!
// Access the data structure(Mutual exclusion)
// End critical region
ready = true;
// Out of Critical Region
```

Mutual exclusion: 1st try

```
bool ready = true; //Global variable

// Start Critical Region
while ( !ready ) {
  scheduler();
}
ready = false;
// Inside the critical region!
// Access the data structure(Mutual exclusion)
// End critical region
ready = true;
// Out of Critical Region
```

Mutual exclusion: 1st try

```
bool ready = true; //Global variable

// Start Critical Region
while ( !ready ) {
  scheduler();
}
ready = false;
// Inside the critical region!
// Access the data structure(Mutual exclusion)
// End critical region
ready = true;
// Out of Critical Region
```

- Same problem as before ☹
- We must have some means of test and set a variable
  - so that no interrupt can occur between both

Test-and-Reset*

- We want a function that if parameter is
  - *false*, it returns *false*,
  - *true*, it is set to *false*, but returns the old value (*true*).
- The function `test_and_reset(…)` is a function that is translated to a single machine instruction which can be thought of as:

```
bool test_and_reset( bool *value ) {
  bool previous = *value;
  *value = false;
  return previous;
}
```

Mutual exclusion: 2nd try

```
bool ready = true; //Global variable

// Enter Critical Region
while ( !test_and_reset(&ready) ) {
  scheduler();
}

// Inside the critical region!
// Access the data structure(Mutual exclusion)
// Exit critical region
ready = true;
// End Critical Region
```

Mutual exclusion: 2nd try

```
bool ready = true; //Global variable
```

mutex_init

```
// Enter Critical Region
while ( !test_and_reset(&ready) ) {
  scheduler();
}

// Inside the critical region!
// Access the data structure(Mutual exclusion)
// Exit critical region
ready = true;
// End Critical Region
```

Mutual exclusion: 2nd try

```
bool ready = true; //Global variable
```

mutex_init

```
// Enter Critical Region
while ( !test_and_reset(&ready) ) {
  scheduler();
}
```

mutex_lock

```
// Inside the critical region!
// Access the data structure(Mutual exclusion)
// Exit critical region
ready = true;
// End Critical Region
```

Mutual exclusion: 2nd try

```
bool ready = true; //Global variable
```

mutex_init

```
// Enter Critical Region
while ( !test_and_reset(&ready) ) {
  scheduler();
}
```

mutex_lock

```
// Inside the critical region!
// Access the data structure(Mutual exclusion)
// Exit critical region
ready = true;
// End Critical Region
```

mutex_unlock

Mutual exclusion: 2nd try

```
bool ready = true; //Global variable
```

mutex_init

```
// Enter Critical Region
while ( !test_and_reset(&ready) ) {
  scheduler();
}
```

mutex_lock

```
// Inside the critical region!
// Access the data structure(Mutual exclusion)
// Exit critical region
ready = true;
// End Critical Region
```

mutex_unlock

- But real-time systems are tricky and if one of the tasks is high priority …

Mutual Exclusion: Example

```
int first = 1; /* GLOBAL VAR == SHARED */
mutex_t critical;
mutex_init(&critical)
create_2_threads( f );
```

1st thread

```
int f() {
  mutex_lock(&critical)
  if (first) {
    first --;
    mutex_unlock(&critical)
    task1();
  else {
    mutex_unlock(&critical)
    task2();
  }
}
```

2nd thread

```
int f() {
  mutex_lock(&critical)
  if (first) {
    first --;
    mutex_unlock(&critical)
    task1();
  else {
    mutex_unlock(&critical)
    task2();
  }
}
```

Pthread Mutex API

- **#include <pthread.h>**
- `pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;`
- `int` **pthread_mutex_init**`(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);`
- `int` **pthread_mutex_lock**`(pthread_mutex_t *mutex);`
- `int` **pthread_mutex_trylock**`(pthread_mutex_t *mutex);`
- `int` **pthread_mutex_unlock**`(pthread_mutex_t *mutex);`
- `int` **pthread_mutex_destroy**`(pthread_mutex_t *mutex);`

Pthread Mutex example

- A couple of threads increasing a global var

Pthread Mutex example

- A couple of threads increasing a global var

```
pthread_mutex_t critical; // Declare a mutex global
```

Pthread Mutex example

- A couple of threads increasing a global var

```
pthread_mutex_t critical; // Declare a mutex global

main(){
 pthread_mutex_init(&critical, NULL);// BEFORE thread creation
 …
 pthread_create( … );
 pthread_create( … );
}
```

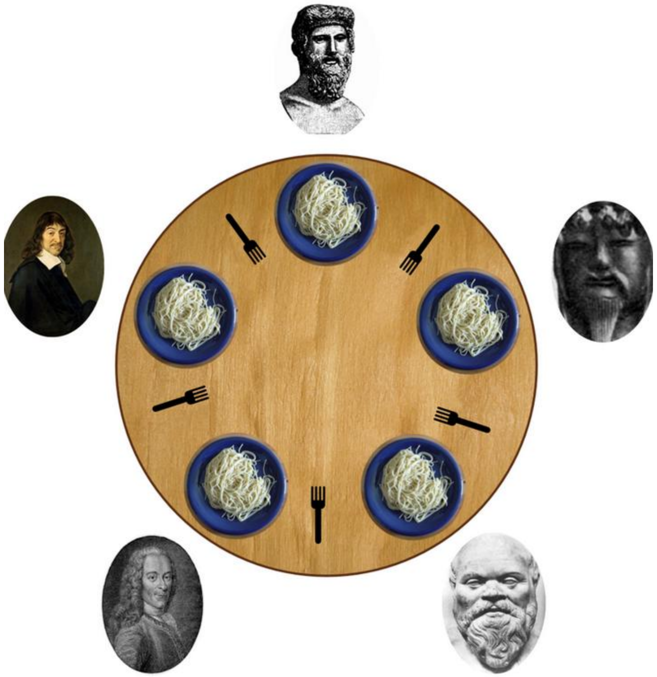
Pthread Mutex example

- A couple of threads increasing a global var

```
pthread_mutex_t critical; // Declare a mutex global

main(){
 pthread_mutex_init(&critical, NULL);// BEFORE thread creation
 …
 pthread_create( … );
 pthread_create( … );
}

pthread_mutex_lock(&critical);  // Enter critical region
pthread_mutex_unlock(&critical);// Exit critical region
```

Dining philosophers

- think →pick left fork → pick right fork → eat → drop righ fork → drop left fork → think → …
- Deadlock



https://en.wikipedia.org/wiki/Dining_philosophers_problem

Deadlock

- The following rules must comply to generate a deadlock:
  – Mutual exclusion: At least 2 non shareable resources
  – Hold & Wait: You get a resource and wait for another
  – No preemption: No one can remove my resource (just me)
  – Circular wait: There is circular sequence of 2 or more tasks, where each one needs a resource being used by another one.

Breaking deadlocks

- Break ANY of the previous rules:
  - Add shareable resources
  - Enable resource stealing
  - Pick all resources or none
  - Sort the requests (enforce the order in which the resources are requested)

POSIX semaphores API

- POSIX semaphores allow processes and threads to **synchronize** their actions.
  – Unnamed → shared memory
  – Named → shared filesystem
- A semaphore is an integer whose value is never allowed to fall below zero.
- Two operations can be performed on semaphores:
  – **decrement** the semaphore value by one (**sem_wait**).
    - If value is 0 → will block until the value becomes greater than zero.
  – and **increment** the semaphore value by one (**sem_post**)
    - if value becomes >0 → unblock a previously blocked proc/thread
- (Courtesy Linux Programmer's Manual)

POSIX semaphores(unnamed)

- **#include <semaphore.h>**
- int **sem_init**(sem_t *sem, int pshared, unsigned int value);
- int **sem_wait**(sem_t *sem);
- int **sem_post**(sem_t *sem);
- int **sem_trywait**(sem_t *sem);
- int **sem_timedwait**(sem_t *sem, const struct timespec *abs_timeout);
- int **sem_destroy**(sem_t *sem);

Semaphores use

- `int sem_init(sem_t *sem, int pshared, unsigned int value);`
- The **value** used to initialize the semaphore defines its use:
  - **1**: MUTEX. Allow a single thread
  - **0**: SYNCHRONIZATION
  - **N**: RESTRICT RESOURCE USAGE. Allow only N threads
- **pshared** defines the sharing of the semaphore
  - 0: Only shared between the threads of a process
  - 1: Shared between processes

POSIX semaphore example

- A couple of threads increasing a global var

POSIX semaphore example

- A couple of threads increasing a global var

```
sem_t critical; // Declare a semaphore global
```

POSIX semaphore example

- A couple of threads increasing a global var

```
sem_t critical; // Declare a semaphore global

main(){
 sem_init(&critical, 0, 1);// BEFORE thread creation
                           // A MUTEX
 …
 pthread_create( … );
 pthread_create( … );
}
```

POSIX semaphore example

- A couple of threads increasing a global var

```
sem_t critical; // Declare a semaphore global

main(){
 sem_init(&critical, 0, 1);// BEFORE thread creation
                           // A MUTEX
 …
 pthread_create( … );
 pthread_create( … );
}
sem_wait(&critical); // Enter critical region
sem_post(&critical); // Exit critical region
```

POSIX semaphore example

- Execute in order.

POSIX semaphore example

- Execute in order.

```
sem_t s1; // Declare a semaphore per thread
sem_t s2;
```

POSIX semaphore example

- Execute in order.

```
sem_t s1; // Declare a semaphore per thread
sem_t s2;
main(){
 sem_init(&s1, 0, 0);// BEFORE thread creation
                     // A SYNCHRONIZATION
 …
 pthread_create( … );
 pthread_create( … );
 sem_post(&s1); // Wake 1st
}
```

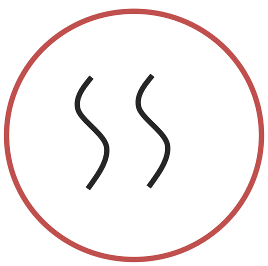
POSIX semaphore example

- Execute in order.

```
sem_t s1; // Declare a semaphore per thread
sem_t s2;
main(){
 sem_init(&s1, 0, 0);// BEFORE thread creation
                     // A SYNCHRONIZATION
 …
 pthread_create( … );
 pthread_create( … );
 sem_post(&s1); // Wake 1st
}

sem_wait(&s1); // Block at the beginning of the thread
sem_post(&s2); // Unblock following thread
```

Other ways to share memory

- Global variables are simple ways to share memory by related threads, but what for processes?

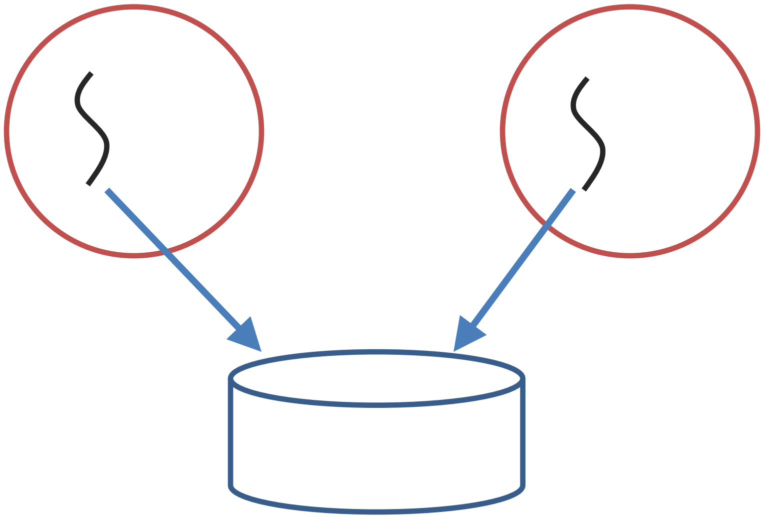


Other ways to share memory

- Global variables are simple ways to share memory by related threads, but what for processes?

Other ways to share memory

- Global variables are simple ways to share memory by related threads, but what for processes?



POSIX semaphores (named)

- **`#include <semaphore.h>`**
- `sem_t *`**`sem_open`**`(const char *name, int oflag);`
- `sem_t *`**`sem_open`**`(const char *name, int oflag,`
  `mode_t mode, unsigned int value);`
- `int `**`sem_close`**`(sem_t *sem);`
- `int `**`sem_unlink`**`(const char *name);`

POSIX named semaphore example

1. Create the semaphore

```
sem_t *s = sem_open("mysemaphore", O_CREAT|O_RDWR, 0777, 0);
```

2. Open the semaphore (from other process)

```
sem_t *s = sem_open("mysemaphore", O_RDWR);
```

3. Use normally
4. Destroy after use

```
sem_close(s);
sem_unlink("mysemaphore");
```

POSIX shared memory

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>
```

- `int` **`shm_open`** `(const char *name, int oflag, mode_t mode);`
- `int` **`ftruncate`** `(int fd, off_t length);`
- `void *`**`mmap`** `(void *addr, size_t length, int prot, int flags, int fd, off_t offset);`
- `int` **`munmap`** `(void *addr, size_t length);`
- `int` **`shm_unlink`** `(const char *name);`
- `int` **`close`** `(int fd);`

- Link with **`-lrt`**

POSIX shared memory example

1. Create the shared memory region

```
int r = shm_open("regionname", O_CREAT|O_RDWR, 0777, 0);
ftruncate(r, 4096); //Set region s size
```

2. Map the region into process

```
int *region = (int*) mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_SHARED, r, 0);
close(r); // file descriptor may be closed now
```

3. Use normally

```
region[0] = xxxx
region[1] = …
```

4. Destroy after use

```
munmap(region, 4096)
shm_unlink("regionname");
```

POSIX shared memory example

1. Open the shared memory region

```
int r = shm_open("regionname", O_RDWR, 0777, 0);
```

2. Map the region into process

```
int *region = (int*) mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_SHARED, r, 0);
close(r); // file descriptor may be closed now
```

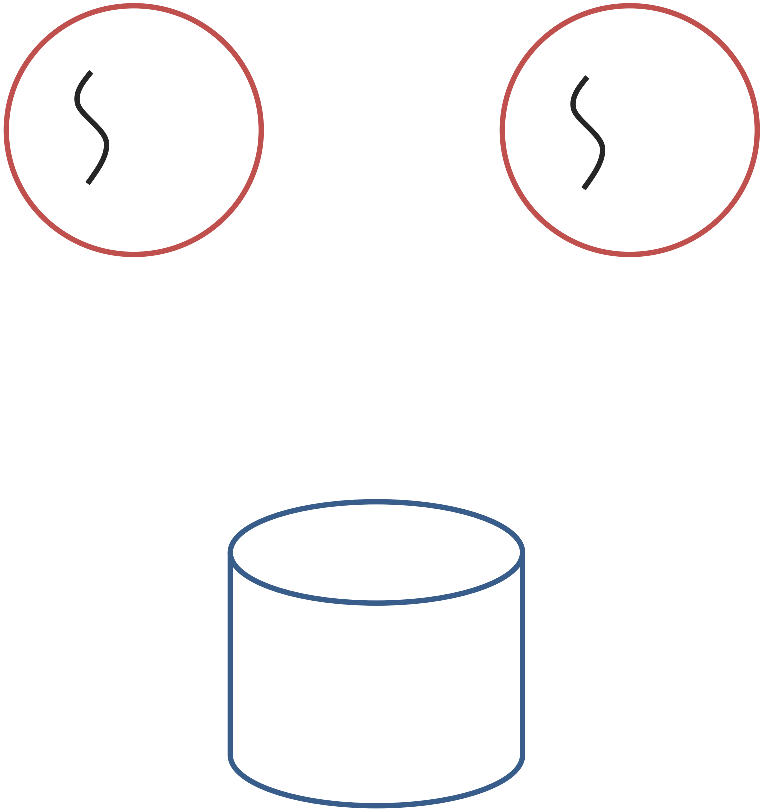
3. Use normally

```
region[0] = xxxx
region[1] = …
```

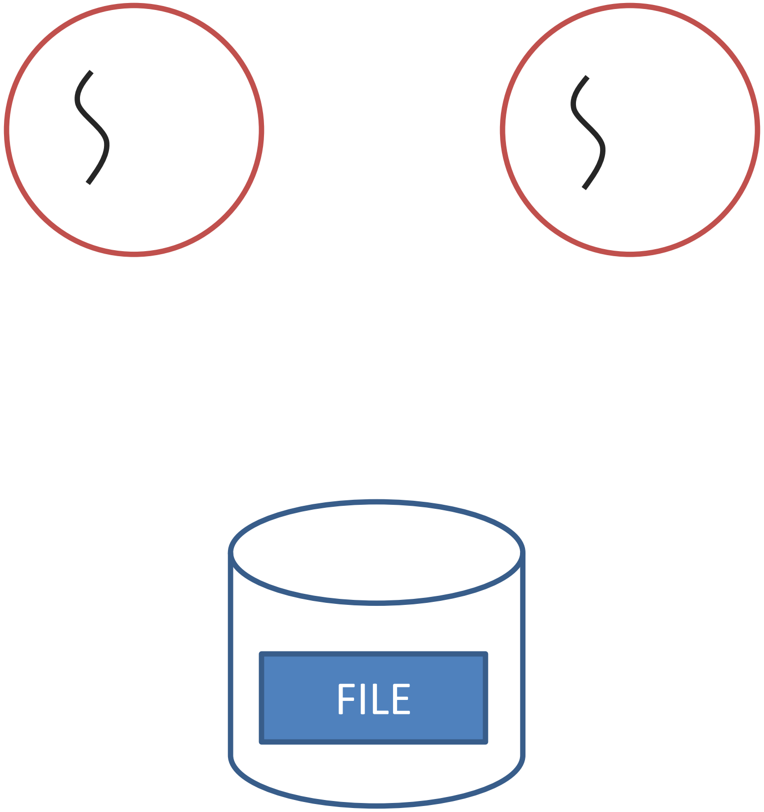
4. Unmap after use

```
munmap(region, 4096);
```

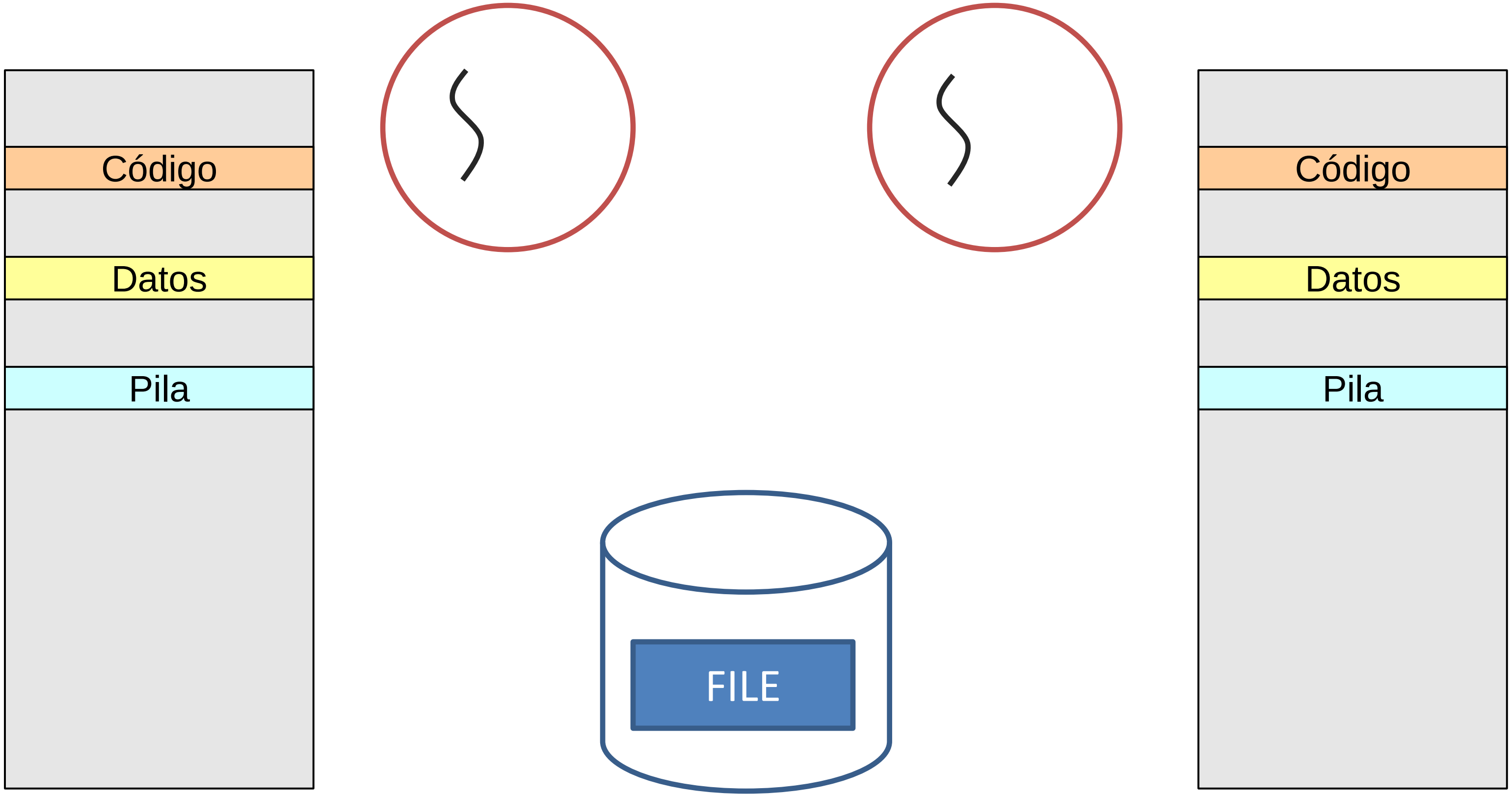
POSIX shared memory example



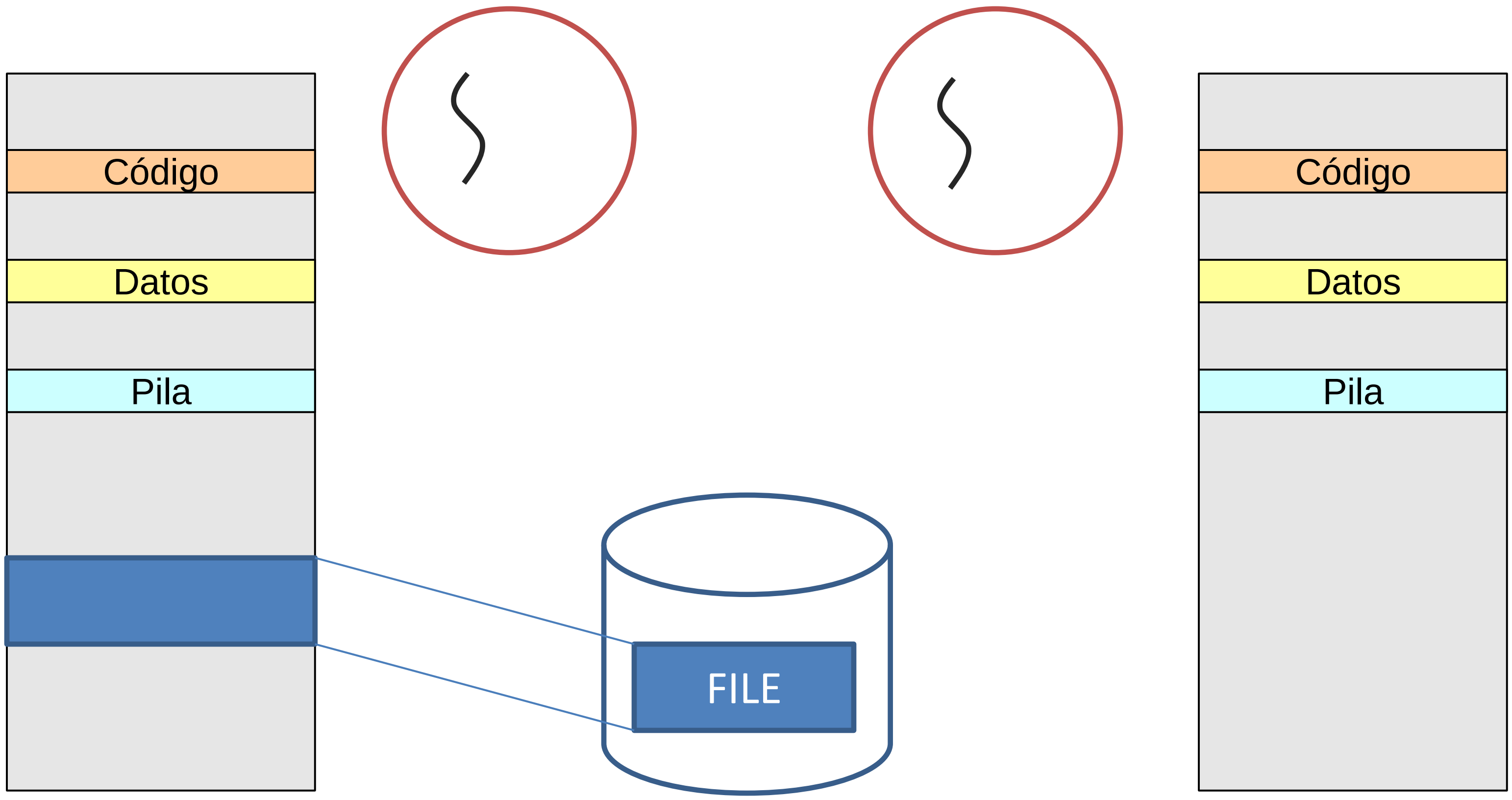
POSIX shared memory example



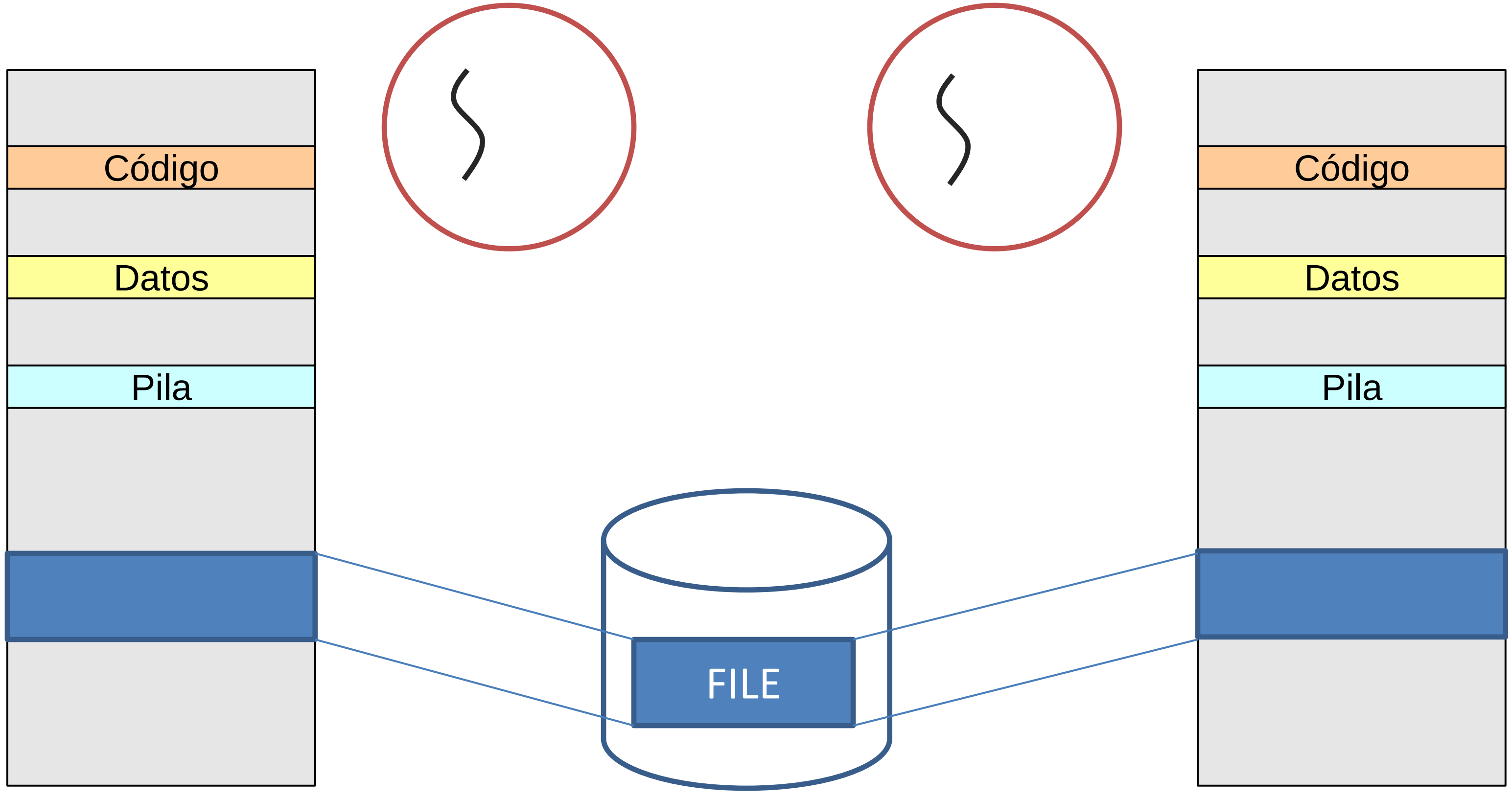
POSIX shared memory example



POSIX shared memory example



POSIX shared memory example



Exercises

- Detect critical regions
- Protect region with mutex
- Protect region with semaphores
- Petr Hudeček and Michal Pokorný. The Deadlock Empire. HackCambridge 2016. https://deadlockempire.github.io/

Shared memory Summary

- Problems:
  - Critical region
  - Serialization
- Solutions
  - Token passing
  - Mutual exclusion
  - Semaphores
- Sharing memory between processes

Index

- Why communicate and synchronize?
- Shared memory
- Distributed memory

Distributed memory

- 2 process may not share memory
  - same or different machines
  - [Harder18]ch12
- signals
- pipes
- message queues (Bertolotti12) 7
- sockets
- mailboxes (Bertolotti12)7
- Blocking operations

Considerations

- Message type
  - synchronization
  - communication
- Information of message
  - implicit
  - explicit
- Example: Post-office service 12.1

Considerations

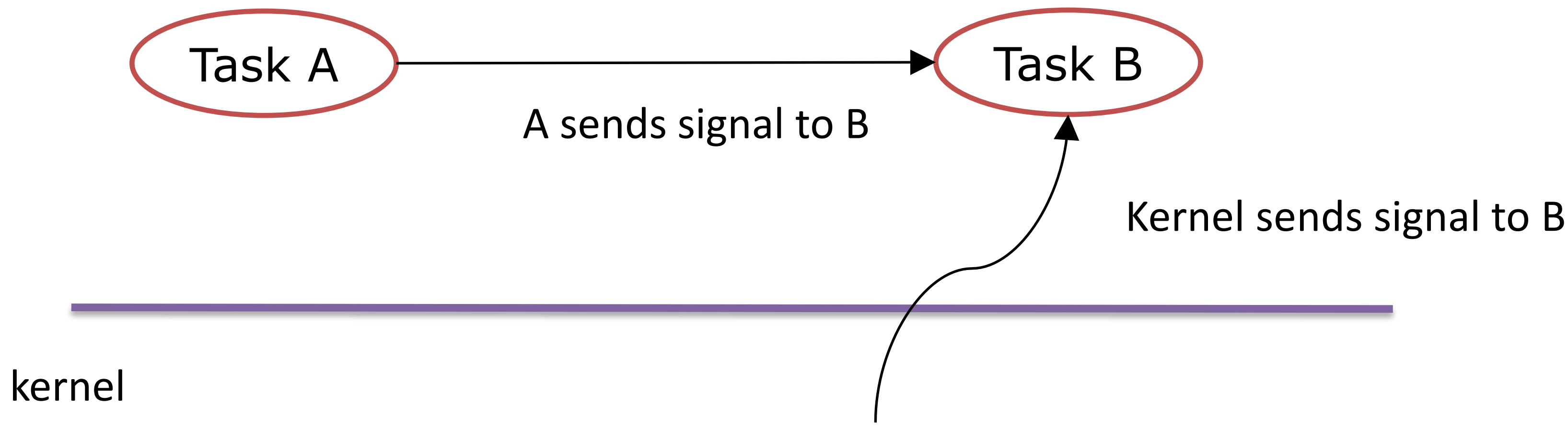
- 2 main operations
  - send
  - receive
- Direct or indirect naming (Bertolotti12 ch 6)
  - mailboxes
- Synchronous or asynchronous

Distributed memory index

- Signals
- File Descriptors
- Pipes
- Message queues
- Sockets
- Mailboxes

Signals

- Signal: A notification stating that an event has happened
- May be sent by the kernel or same user tasks



Signals

- Each signal has an associated **handler**
  - That manages the reaction to it (behavior)
  - Predefined by the kernel
- Each signal has an identification number
- A task may capture/change their behavior
  - There are exceptions: SIGKILL and SIGSTOP
- Each task may have different handlers

Signals types

- Some signals:

Name	Default action	Event
SIGCHLD	IGNORE	A child has terminated or stopped
SIGCONT	CONTINUE	Continue if stopped
SIGSTOP	STOP	Stop task
SIGINT	TERMINATE	Keyboard interrupt (Ctrl-C)
SIGALRM	TERMINATE	The counter defined in ´alarm´ call has finished
SIGKILL	TERMINATE	Finish the task
SIGSEGV	CORE	Invalid reference to memory
SIGUSR1	TERMINATE	**User defined**
SIGUSR2	TERMINATE	**User defined**

Signal management

- Works like a hardware interrupt generated by sw
  - When a task receives a signal
    - interrupt current code execution
    - jump to signal management code (handler)
    - if the handler does not kill the process
      - return to previous code execution
- A task may block/unblock the reception of signals
  - It has a **mask of blocked signals**
  - Except SIGKILL and SIGSTOP
  - On reception of a blocked signal
    - It is queued (*just one*)
    - It will be handled when unblocked

Linux: Signals interface

Service	System call
Send an specific signal	kill
Capture/reprogram a signal handler	sigaction
Block/unblock signals	sigprocmask
Wait UNTIL any event is received (BLOCK)	sigsuspend
Program the automatic sending of signal SIGALRM	alarm

- File /usr/include/bits/signum.h
  - Linux implements POSIX interface

Interface: send/capture signals

- Send

```
int kill(int pid, int signum)
```

  – signum → SIGUSR1, SIGUSR2, …
- Capture/Reprogram

```
int sigaction(int signum, struct sigaction *handler,
struct sigaction *old_handler)
```

  – signum → SIGUSR1, SIGUSR2, …
  – handler → *struct sigaction* describes what to do
  – old_handler → describes the old behavior
    - May be NULL if not interested

A sends signal to B

- Task A sends a signal to B and B runs a specific action when it is received
  - Time of signal reception is unknown

```
Task A
.....
kill( pid, event );
....
```

```
Task B
int main()
{
struct sigaction hand, old_hand;
/* code to initialize hand*/
sigaction( event, &hand, &old_hand );
....
}
```

struct sigaction

```
struct sigaction {
              void      (*sa_handler)(int);
              void      (*sa_sigaction)(int, siginfo_t *, void *);
              sigset_t   sa_mask;
              int        sa_flags;
              void      (*sa_restorer)(void);
          };
```

- sa_handler: Defines signal management
  - SIG_IGN: Ignore signal at reception
  - SIG_DFL: Default behavior
  - the user function to call when receiving signal
    - The signal identifier is received as the parameter
- sa_mask: signals to block during the handler execution
  - The blocked signals are added to the thread mask
  - At least, the captured signal is added
  - At exit from the handler the mask before enter is restored

struct sigaction

```
struct sigaction {
              void      (*sa_handler)(int);
              void      (*sa_sigaction)(int, siginfo_t *, void *);
              sigset_t   sa_mask;
              int        sa_flags;
              void      (*sa_restorer)(void);
          };
```

- sa_flags: configure behavior
  - 0: Use the default configuration
  - SA_RESETHAND: after executing handler, restore default signal behavior
  - SA_RESTART: if process was blocked in a system_call, restart the blocking system_call

Signals example

```
void main()
{   char buffer[128];
    struct sigaction hand;
    sigset_t         mask;

    sigemptyset(&mask);
    hand.sa_mask    = mask;
    hand.sa_flags   = 0;
    hand.sa_handler = int_handler;

    sigaction(SIGINT, &hand, NULL); // Execute int_handler when
                                    // receiving SIGINT
    while(1) {
        sprintf(buffer, "I am working hard\n");
        write(1, buffer, strlen(buffer));
    }
}

void int_handler(int s)
{
    char buffer[128];
    sprintf(buffer, "SIGINT RECIBIDO!\n");
    exit(0);
}
```

Signal mask management

- sigemptyset: initialize a mask without blocked signals

```
int sigemptyset(sigset_t *mask)
```

- sigfillset: initialize a mask with all signals blocked

```
int sigfillset(sigset_t *mask)
```

- sigaddset: add a signal to block to the mask

```
int sigaddset(sigset_t *mask, int signum)
```

- sigdelset: remove from the mask a signal

```
int sigdelset(sigset_t *mask, int signum)
```

- sigismember: check if a signal is in the mask

```
int sigismember(sigset_t *mask, int signum)
```

Block/unblock signals

- sigprocmask

```
int sigprocmask(int operation, sigset_t *mask, sigset_t *old_mask)
```

- operation value:
  - SIG_BLOCK: **Add** signals in *mask* to blocked signals mask of task.
  - SIG_UNBLOCK: **Remove** signals in *mask* from blocked signals mask of task
  - SIG_SETMASK: **Set** *mask* as blocked signals mask

Wait for signals

- sigsuspend

```
int sigsuspend(sigset_t *mask)
```

- Suspend task until a **non ignored signal** is delivered
  - Except: SIGKILL or SIGSTOP
- Change current task´s mask by *mask*
  → select signal that will unblock
- After handler execution,
  - signal mask → state before sigsuspend
  - pending signals will be handled

Synchronization: 1st try

- Task B wants to wait for an event from A

Task A
```
.....
kill( pid, event);
....
```

Task B
```
int main()
{
sigaction(event, &hand, NULL);
....
sigemptyset(&mask);
sigsuspend(&mask);
....
}
```

¿What happens if A sends the event BEFORE B arrives to the sigsuspend?
¿What happens if B receives another event while in sigsuspend?

Synchronization: 2nd try

```
Task A
.....
kill( pid, event);
....
```

- Event is blocked until sigsuspend
- Event is the only accepted to unblock

```
Task B
int main()
{
sigemptyset(&mask);
sigaddset(&mask, event);
sigprocmask(SIG_SETMASK,&mask,NULL);
sigaction(event, &hand, NULL);
....
sigfillset(&mask);
sigdelset(&mask, event)
sigsuspend(&mask);
....
}
```

Time control

- Program the automatic sending of a SIGALRM
  - The kernel will send it to current process
- `int alarm(int secs);`

```
ret = rem_time;
si (secs == 0) {
        send_SIGALRM = OFF
} else {
        send_SIGALRM = ON
        rem_time = secs,
}
return ret;
```

Time control example

- Task programs a timer for 2 seconds and blocks itself until that time passes

```
int main()
{
sigemptyset(&mask);
sigaddset(&mask, SIGALRM);
sigprocmask(SIG_SETMASK,&mask, NULL);
sigaction(SIGALRM, &hand, NULL);
....
sigfillset(&mask);
sigdelset(&mask,SIGALRM);
alarm(2);
sigsuspend(&mask);
....
}
```

Fork/exec relation

- FORK: **New process**
  - Table of signal handlers is inherited
  - Events are sent to specific processes (PID´s), child is a new process → the list of pending signals is reset
    - (pending timers are also reset)
  - The blocked signals mask is inherited
- EXECLP: **Same process**, different code
  - Table of signal handlers is reset
  - Events are sent to specific processes (PID´s), process is the same → the list of pending signals is kept
  - The blocked signals mask is kept

Ejemplo 1: gestión de 2 signals (1)

```
void main()
{
   sigemptyset(&mask1);
   sigaddset(&mask1,SIGALRM);
   sigprocmask(SIG_BLOCK,&mask1,NULL);

   trat.sa_flags=0;
   trat.sa_handler = f_alarma;
   sigemptyset(&mask2);
   trat.sa_mask=mask2;
   sigaction(SIGALRM, &trat, NULL);

   sigfillset(&mask3);
   sigdelset(&mask3,SIGALRM);

   for(i = 0; i < 10; i++) {
     alarm(2);
     sigsuspend(&mask3);
     crea_ps();
   }
}
```

```
void f_alarma()
{
}
void crea_ps()
{
  pid = fork();
  if (pid == 0)
    execlp("ps", "ps",
               (char *)NULL);
}
```

Podéis encontrar el código completo en: cada_segundo.c

Ejemplo 2: espera activa vs bloqueo (1)

```c
void main()
{
    configurar_esperar_alarma()
    trat.sa_flags = 0;
    trat.sa_handler=f_alarma;
    sigsetempty(&mask);
    trat.sa_mask=mask;
    sigaction(SIGALRM,&trat,NULL);
    trat.sa_handler=fin_hijo;
    sigaction(SIGCHLD,&trat,NULL);
    for (i = 0; i < 10; i++) {
        alarm(2);
        esperar_alarma(); // ¿Qué opciones tenemos?
        crea_ps();
    }
}
void f_alarma()
{
    alarma = 1;
}
void fin_hijo()
{
    while(waitpid(-1,NULL,WNOHANG) > 0);
}
```

```c
void crea_ps()
{
    pid = fork();
    if (pid == 0)
        execlp("ps", "ps",
               (char *)NULL);
}
```

Podéis encontrar el código completo en: cada_segundo_sigchld.c

Ejemplo 2: espera activa vs bloqueo (2)

Opción 1: espera activa

```
void configurar_esperar_alarma() {
   alarma = 0;
}
void esperar_alarma(){
   while (alarma!=1);
   alarma=0;
}
```

Opción 2: bloqueo

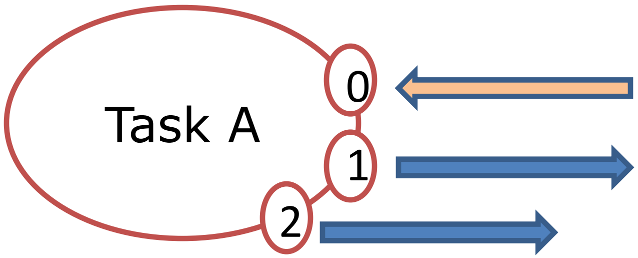
```
void configurar_esperar_alarma() {
   sigemptyset(&mask);
   sigaddset(&mask, SIGALRM);
   sigprocmask(SIG_BLOCK,&mask, NULL);
}

void esperar_alarma(){
         sigfillset(&mask);
         sigdelset(&mask,SIGALRM);
         sigsuspend(&mask);
}
```

Distributed memory index

- Signals
- File Descriptors
- Pipes
- Message queues
- Sockets
- Mailboxes

File Descriptors

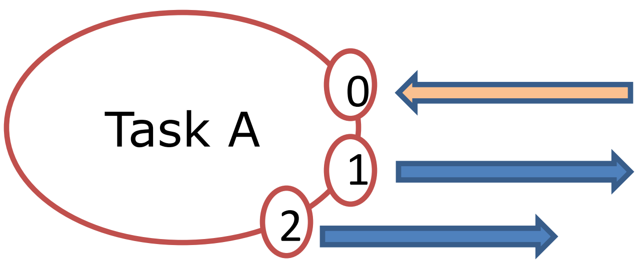
- In order to send and receive data, a process uses channels to access other devices
- By default, 3 channels
  - 0 → standard INPUT
  - 1 → standard OUTPUT
  - 2 →standard ERROR



- The 3 first entries of the kernel's file descriptor table.
  - A per process table
- When opening a file (or a device) a new entry is added (4, 5, …)
  - uses the first empty entry in the table

Read and Write

- With these channels you may read/write data
  - (or any other system call using a file descriptor)
- int read(int fd, char *buffer, int size)
- int write(int fd, char *buffer, int size)



- `write(1, “hello world”, 11);`
- `read(0, &buffer, 20);`

IPC

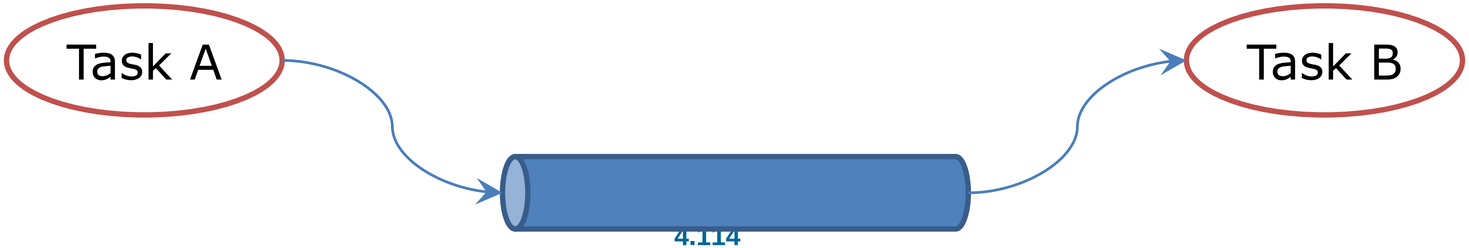
- Connecting the channels of two processes enables a communication between them
- Let’s see a couple of ways of doing that…

Distributed memory index

- Signals
- File Descriptors
- Pipes
- Message queues
- Sockets
- Mailboxes

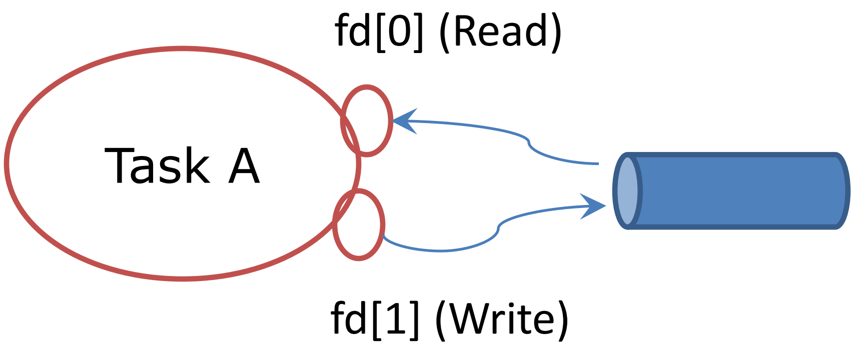
Pipe

- Unidirectional interprocess communication channel
- A read-end and a write-end
  – 2 new file descriptors
- Processes MUST be related
  – For example: father/child
- Limited capacity (buffered by kernel)



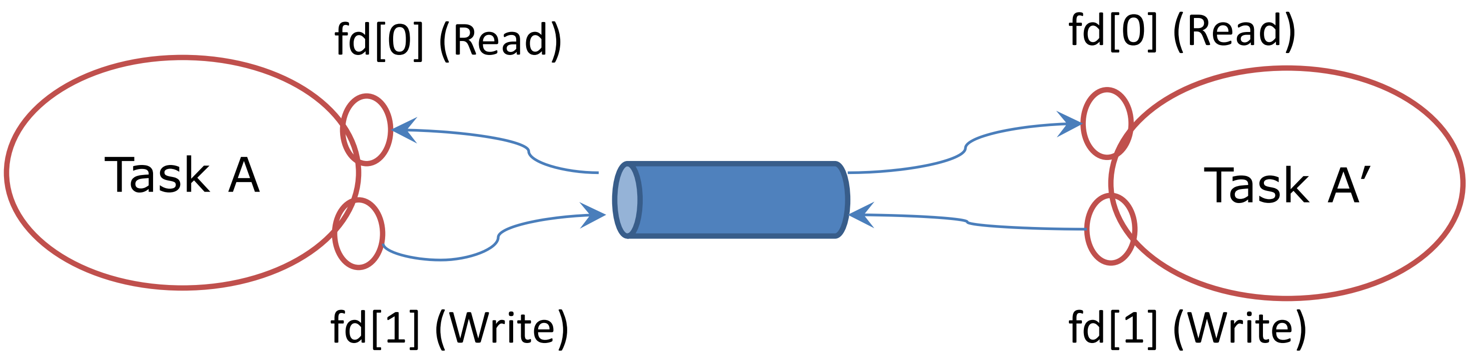
Pipe interface

- Create a pipe
  - int pipe(int fd[2])



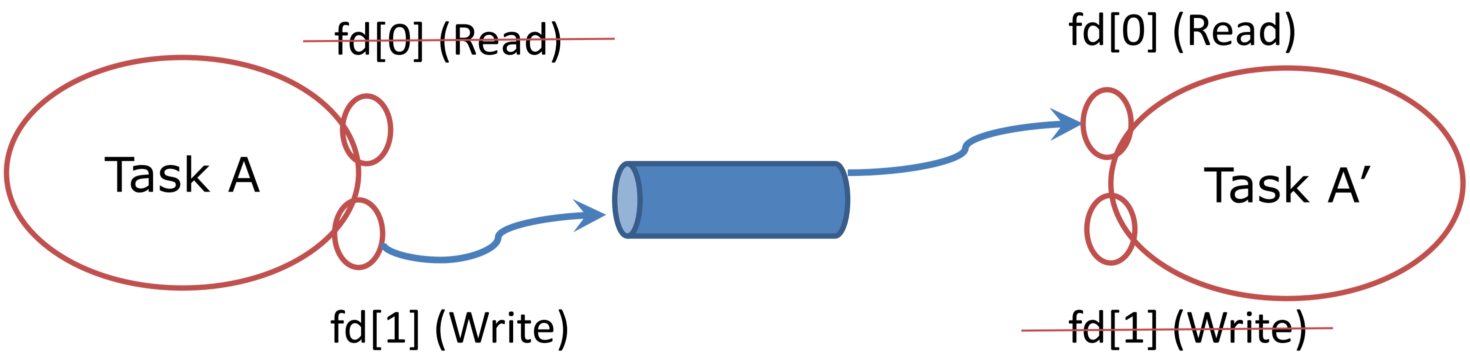
Pipe interface

- Duplicate process
  - fork();



Pipe interface

- Select direction
  - close()

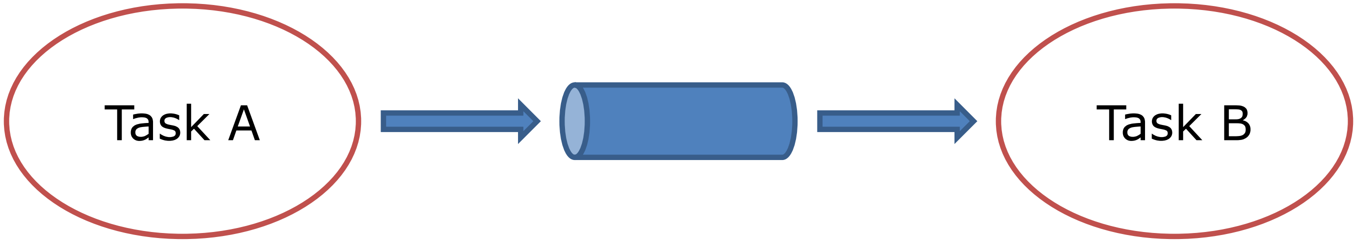


Pipe behavior

- Blocking device
- Read:
  - **Block** process until data in pipe (not all data requested)
  - If pipe is empty and there are no writers, then EOF (returns 0)
    - If pipe had blocked processes → they are unblocked
- Write:
  - Process write until the pipe is full, then **block** until it is emptied
  - If there is no reader process, writing process will receive SIGPIPE
    - Any other blocked process → unblock
- Unused channels must be closed! Or block.

- This blocking behavior may be used for synchronization

Pipes example




```
void main()
{
  int pipefd[2];
  pipe(pipefd);
  int p = fork();
  if (p == 0) { //Child (Task B)
    close(pipefd[1]);
    int len = read(pipefd[0], buffer, sizeof(buffer));
    write(1,  buffer, len);
  else { // Father (Task A == main process)
    close(pipefd[0]);
    write(pipefd[1], “hola”, 4);
  }
}
```

Named Pipes

- What happens if you need to communicate 2 processes that are not related?
  - Unnamed pipe can not be used
- A **named pipe** == pipe but:
  - with a name in file system
    - visible globally in the system
  - needs to be opened at both ends before operation
  - opening for read blocks until a writer appear
    - and vice versa

Named pipe creation

- Create a named pipe
  - #include <syst/stat>
  - int mknod(char *path, mode_t mode, dev_t dev)

    or
  - int mkfifo(const char *pathname, mode_t mode);

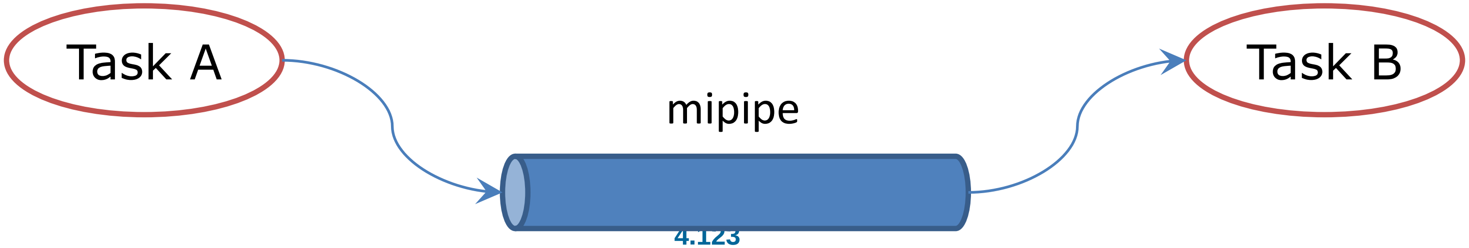
- mknod(“mipipe”, S_IFIFO|S_IRUSR|S_IWUSR, 0)

  or
- mkfifo(“mipipe”, S_IRUSR|S_IWUSR)

mipipe

Acces named pipe

- Open named pipe
  - int open(const char *pathname, int flags);
  - `Process A`
    - `fd = open("mipipe", O_WRONLY)`
    - `write(fd, "hola", 4);`
  - `Process B`
    - `fd = open("mipipe", O_RDONLY)`
    - `read(fd, buffer, sizeof(buffer))`



Avoid blocking

- The blocking on the pipe can be avoided
  - O_NONBLOCK flag
- At *open* or at creation through *fcntl*
  - `#include <fcntl.h>`
  - `int fcntl(int fildes, int cmd, ...);`

  - open(“mipipe”, O_RDONLY **| O_NONBLOCK**)

    or
  - **fcntl(fd, F_SETFL, O_NONBLOCK)**

Non blocking behavior

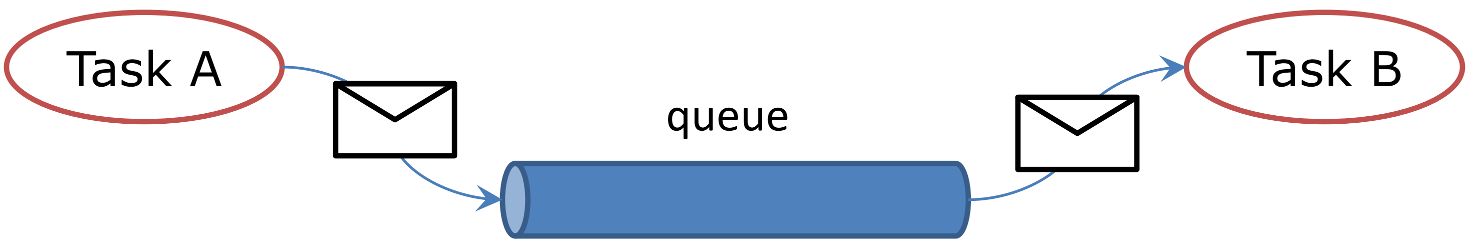
- Open named pipe:
  - Read only: returns the file descriptor wo block
  - Write only: returns error (errno==ENXIO)
- Read on an empty pipe with writers:
  - returns error (errno == EAGAIN)
- Write on a full pipe:
  - returns error (errno == EAGAIN)

Distributed memory index

- Signals
- File Descriptors
- Pipes
- Message queues
- Sockets
- Mailboxes

Message queues

- Solve some of the limitations of pipes
  - Granularity is message not bytes
  - Messages are atomic



Message queues interface

```
#include <fcntl.h>        /* For O_* constants */
#include <sys/stat.h>     /* For mode constants */
#include <mqueue.h>
```

- `mqd_t` **`mq_open`**`(const char *name, int oflag);`
- `mqd_t` **`mq_open`**`(const char *name, int oflag, mode_t mode,`
  `struct mq_attr *attr);`
- `int` **`mq_send`**`(mqd_t mqdes, const char *msg_ptr,`
  `size_t msg_len, unsigned int msg_prio);`
- `ssize_t` **`mq_receive`**`(mqd_t mqdes, char *msg_ptr,`
  `size_t msg_len, unsigned int*msg_prio);`
- `int` **`mq_close`**`(mqd_t mqdes);`

- `Link with -lrt.`

POSIX Message queues

- Create and open using `mq_open(3)`

  →returns a message queue descriptor (mqd_t)
- Transfer messages to and from a queue using `mq_send(3)` and `mq_receive(3)`
- When a process has finished using the queue, it closes it using `mq_close(3)`
- When the queue is no longer required, it can be deleted using `mq_unlink(3)`

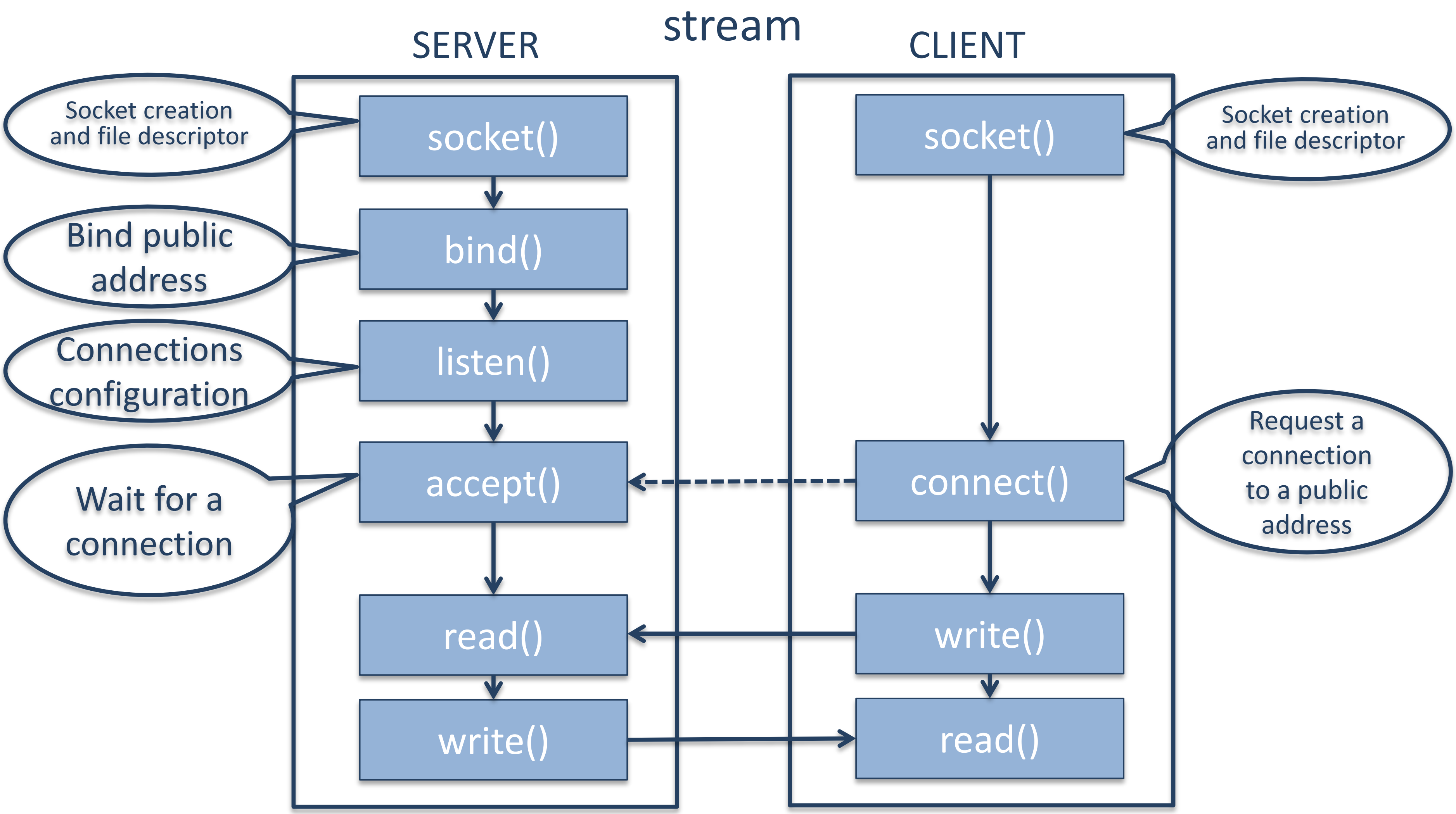
Distributed memory index

- Signals
- File Descriptors
- Pipes
- Message queues
- Sockets
- Mailboxes

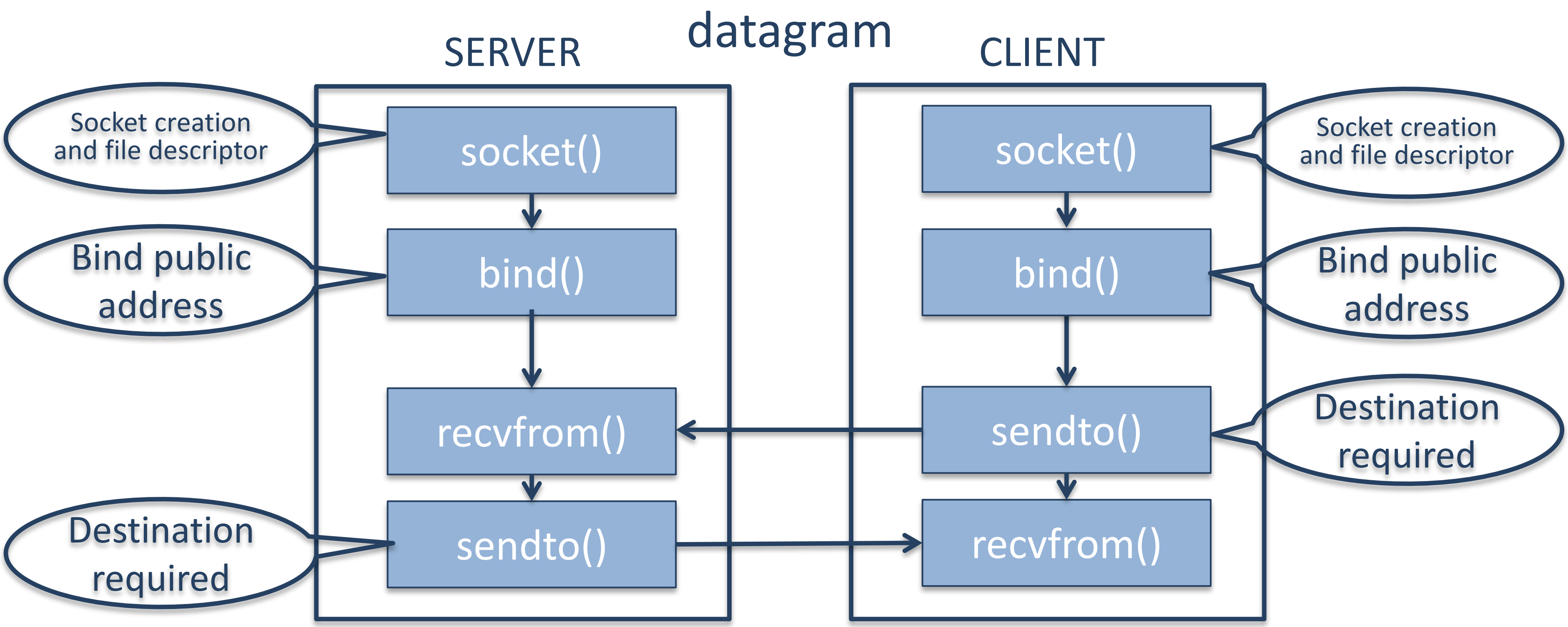
Sockets

- Bidirectional channel between processes
  – Local
  – Remote (through network)
- The following must be defined:
  – Namespace (Unix, INET, …)
  – Communication type (stream or datagram)
  – Communication protocol (TCP, UDP, …)

Sockets: Stream communication



Sockets: Datagram connection



Endianness

- When transmitting data from one machine to another, they may have different ways to store complex types
  - Big Endian
  - Little Endian
- In order to transfer these types you need to translate them
  - 1) from host to network (htons, htonl)
  - 2) from network to host (ntohs, ntohl)

Create socket

- `#include <sys/types.h>`
- `#include <sys/socket.h>`
- `int` **`socket`** `(int af, int type, int proto)`
  - af: address family(namespace to use)
    - AF_UNIX
    - AF_INET
  - type: connection type
    - SOCK_STREAM
    - SOCK_DGRAM
  - proto: Use 0 to automatically use the most apropiate
  - Returns the *file descriptor* associated to the socket or -1 if error

Bind address

- #include <sys/socket.h>
- int **bind** (int fd,
  struct sockaddr *addr, int addr_size)
  - fd: the one returned by socket
  - addr: address to connect to this socket
  - addr_size: socket address size in bytes
  - returns 0 if ok or -1 if error

- *addr* uses a generic type (struct sockaddr) but depending on the socket's namespace you must use the specific type
  - AF_UNIX: nombre de fichero [struct sockaddr_un]
  - AF_INET: IP+puerto [struct sockaddr_in]

Socket addresses

- Generic structure for addresses

```
#include <sys/un.h>
 struct sockaddr {
     unsigned short sa_family;    // address family, AF_xxx
     char           sa_data[14];  // 14 bytes of protocol address
 };
```

- This is just a “container” for the specific addresses… DO NOT USE directly!

Socket addresses

- Socket namespace: PF_UNIX: path to a file

```
#include <sys/un.h>
struct sockaddr_un {
    sa_family_t sun_family;      /* 1 byte == AF_UNIX */
    char sun_name[UNIX_PATH_MAX]
}
```

- Socket namespace: PF_INET: ip address + port

```
#include <sys/netinet/in.h>
struct sockaddr_in {
    sa_family_t      sin_family; /* 1 byte == AF_INET */
    struct in_addr   sin_addr;   /* 4 bytes */
    in_port_t        sin_port;   /* 1 byte */
    char             sin_zero [8]; /* not used, must be 0 */
}
```

  - sin_addr: INADDR_ANY represents machine's IP address running the code
  - sin_port : if 0, system assigns an empty one; otherwise ensure that it is not used.
    - Use ports > 5000
    - Use **Network codification**

Getting socket addresses

- IP → binary

```
struct sockaddr_in sa; // IPv4
inet_pton(AF_INET, "10.12.110.57", &(sa.sin_addr));
```

- binary → IP

```
char ip4[INET_ADDRSTRLEN];  // space to hold
                            //  the IPv4 string
struct sockaddr_in sa; // pretend this is
     loaded with a valid address…

inet_ntop(AF_INET, &(sa.sin_addr),
          ip4, INET_ADDRSTRLEN);
```

Request a connection

- `#include <sys/socket.h>`
- `int` **`connect`**`(int fd,`

  `struct sockaddr *addr, int addr_size)`

- If the server may not attend request

  → return -1

- Else, **block** until server accepts the request
  - If O_NONBLOCK enabled → no block and return -1

Automating address resolution

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo (const char *node,
                 const char *service,
                 const struct addrinfo *hints,
                 struct addrinfo **res);
```

- node: host name or IP
- service: port number or service name
- hints: specifies criteria for selecting the socket address structures returned in the list res
- res: linked list with the results

- Returns 0 if it succeeds

Struct addrinfo

```
 struct addrinfo {
        int              ai_flags;     // AI_PASSIVE, AI_CANONNAME, etc.
        int              ai_family;    // AF_INET, AF_INET6, AF_UNSPEC
        int              ai_socktype;  // SOCK_STREAM, SOCK_DGRAM
        int              ai_protocol;  // use 0 for "any"
        size_t           ai_addrlen;   // size of ai_addr in bytes
        struct sockaddr *ai_addr;      // struct sockaddr_in or _in6
        char            *ai_canonname; // full canonical hostname

        struct addrinfo *ai_next;      // linked list, next node
    };

struct addrinfo hints;
memset(&hints, 0, sizeof hints); // make sure the struct is empty
hints.ai_family = AF_UNSPEC;     // don't care IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM; // TCP stream sockets
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me
…
```

Wait for a connection request

- `#include <sys/socket.h>`
- `int accept(int fd, struct sockaddr *addr, int *addr_size)`
  - fd: the one returned by socket
  - addr: will contain the address requesting the connection
  - addr_size: address size in bytes
  - returns
    - **a new file descriptor to use for connection**
    - or -1 if error
- If pending connections → accept first one
- Otherwise, **block** until a new request
  - unless the flag O_NONBLOCK enabled

    → no block and return error

Sockets example

- Beej's Guide to Network Programming
  http://beej.us/guide/bgnet

Distributed Memory Summary

- There is a need to communicate and synchronize processes
  – Without shared memory
- Different mechanisms
  – Events: Signals
  – IPC
    - Pipes(Unnamed & Named)
    - Message Queues
    - Sockets

References

- [Harder18] Harder, Douglas W et al. 2018. “A practical introduction to real-time systems for undergraduate engineering”
- [Downey08] Downey, Allen B. 2008. “The Little Book of Semaphores”. Green Tea Press
- [JPL09] Jet Propulsion Laboratory. 2009. “JPL Institutional Coding Standard for the C Programming Language” California Institute of Technology.
- [Burns09] Burns, Alan and Wellings, Andy. 2009. “Real-Time Systems and Programming Languages”
- [Bertolotti12] Bertolloti, Ivan and Manduchi, Gabriele.2012.“Real-time embedded systems. Open-source Operating systems perspective”
- [Stevens98] Stevens, Richard. 1998. “Unix Network Programming”, volumes 1-2.