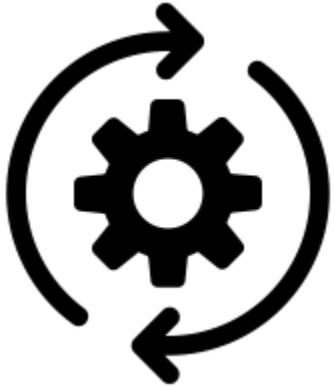


Processi

Gestire l'esecuzione

Argomenti

- Processi e isolamento
- Gestione di processi
- Comunicazione tra processi

Processi

- Un processo costituisce l’unità base di esecuzione di un applicativo nel contesto di un sistema operativo
  - Esso viene identificato in modo esplicito a livello di sistema tramite un numero intero PID – Process ID
  - Definisce uno spazio di indirizzamento all'interno del quale possono operare uno o più thread, flussi di esecuzione schedulabili indipendentemente
  - Lo spazio di indirizzamento fornisce un meccanismo naturale di separazione (isolamento)
  - Allo scopo di evitare che le attività nel contesto di un processo possano "disturbare" quelle di altri processi

Processi e isolamento

- Il livello di isolamento offerto dal concetto di processo è parziale
  - Due processi possono interferire attraverso il file system, sia a livello di file "dati" che a livello di eseguibili e librerie
  - Il sistema di autenticazione/autorizzazione/accounting è una seconda fonte di possibile interferenza
  - Il sottosistema di rete è un'altra fonte di potenziale interferenza
  - In generale, l'accesso alle periferiche di sistema ed alle risorse centralizzate può causare incompatibilità
- In alcune situazioni, il progettista vuole esplicitamente ridurre il livello di isolamento tra due o più processi
  - Offrendo un meccanismo controllato di comunicazione in grado di superare i limiti imposti dalla separazione degli spazi di indirizzamento
- I sistemi operativi offrono, a questo proposito, meccanismi opportuni che ricadono sotto il nome generico di IPC
  - Inter-Process Communication

Concorrenza e processi

- L'uso dei thread permette di sfruttare le risorse computazionali presenti in un elaboratore
  - La presenza di uno spazio di indirizzamento condiviso facilita la coordinazione e la comunicazione
- Ci sono situazioni in cui la presenza di un singolo spazio di indirizzamento non è possibile o desiderabile
  - Riuso di programmi esistenti
  - Scalabilità su più computer
  - Sicurezza

Concorrenza e processi

- È possibile decomporre un sistema complesso in un insieme di processi collegati
  - Creandoli a partire da un processo genitore
  - Permettendo la cooperazione indipendentemente dalla loro genesi
- Ad ogni processo è associato almeno un thread (primary thread)
  - Un sistema multiprocesso è intrinsecamente concorrente
  - Solleva gli stessi problemi di interferenza e necessità di coordinamento

Processi in Windows

- Costituiscono entità separate, senza relazioni di dipendenza esplicita tra loro
- La funzione **`CreateProcess(…)`**
  - Crea un nuovo spazio di indirizzamento
  - Lo inizializza con l'immagine di un eseguibile
  - Attiva il thread primario al suo interno
- Il processo figlio può condividere variabili d'ambiente ed handle a file, semafori, pipe …
  - … ma non può condividere handle a thread, processi, librerie dinamiche e regioni di memoria

Creare processi in Windows

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>

void _tmain( int argc, TCHAR *argv[] )
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );
    if( argc != 2 ){
        printf("Usage: %s [cmdline]\n", argv[0]);
        return;
    }
    //continua…
```

Creare processi in Windows

```
// Start the child process
if( !CreateProcess(
    NULL,    // No module name (use command line)
    argv[1], // Command line
    NULL,    // Process handle not inheritable
    NULL,    // Thread handle not inheritable
    FALSE,   // Set handle inheritance to FALSE
    0,       // No creation flags
    NULL,    // Use parent's environment block
    NULL,    // Use parent's starting directory
    &si,     // Pointer to STARTUPINFO struct
    &pi )    // Pointer to PROCESS_INFORMATION struct
) {
    printf( "CreateProcess failed (%d).\n", GetLastError() );
    return;
}
//…continua…
```

Creare processi in Windows

```
    // Wait until child process exits.
    WaitForSingleObject( pi.hProcess, INFINITE );

    // Close process and thread handles.
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
}
```

```
typedef struct _PROCESS_INFORMATION {
  HANDLE hProcess;
  HANDLE hThread;
  DWORD  dwProcessId;
  DWORD  dwThreadId;
} PROCESS_INFORMATION, *PPROCESS_INFORMATION, *LPPROCESS_INFORMATION;
```

Processi in Linux

- Si crea un processo figlio con la system call **`fork()`**
  - Crea un nuovo spazio di indirizzamento «identico» a quello del processo genitore
  - I due processi condividono i riferimenti alle stesse pagine di memoria fisica
  - Il figlio inizia la propria computazione trovandosi già uno stack popolato con la storia delle chiamate effettuate nel padre, uno heap con della memoria allocata, codice e spazio globale nello stesso stato in cui erano nel padre
- Dopo l'esecuzione di **`fork()`**, tutte le pagine sono marcate con il flag **CopyOnWrite**
  - Eventuali scritture comportano la duplicazione della pagina e la separazione tra i due spazi di indirizzamento

Creazione di Processi

- Le funzioni **`exec*()`** sostituiscono l'attuale immagine di memoria dello spazio di indirizzamento
  - Ri-inizializzandola a quella descritta dall'eseguibile indicato come parametro
  - Differiscono tra loro nel modo di gestire i parametri ricevuti

Esempio

```
int main ( const int argc, const char* const argv[] ) {
    pid_t  childPid = fork();
    switch (childPid) {
         case -1:
            puts( "parent: error: fork failed!" );break;
       case  0:
            puts( "child: here (before execl)!" );
            if (execl( "./ch.exe", "./ch.exe", 0 )==-1)
                  perror( "child: execl failed:" );
            puts( "child: here (after execl)!" );
            //non si dovrebbe arrivare qui
            break;
     default:
               printf( "par: child pid=%d \n", ret );
        break;
    }
 return 0;
}
```

Fork() e thread

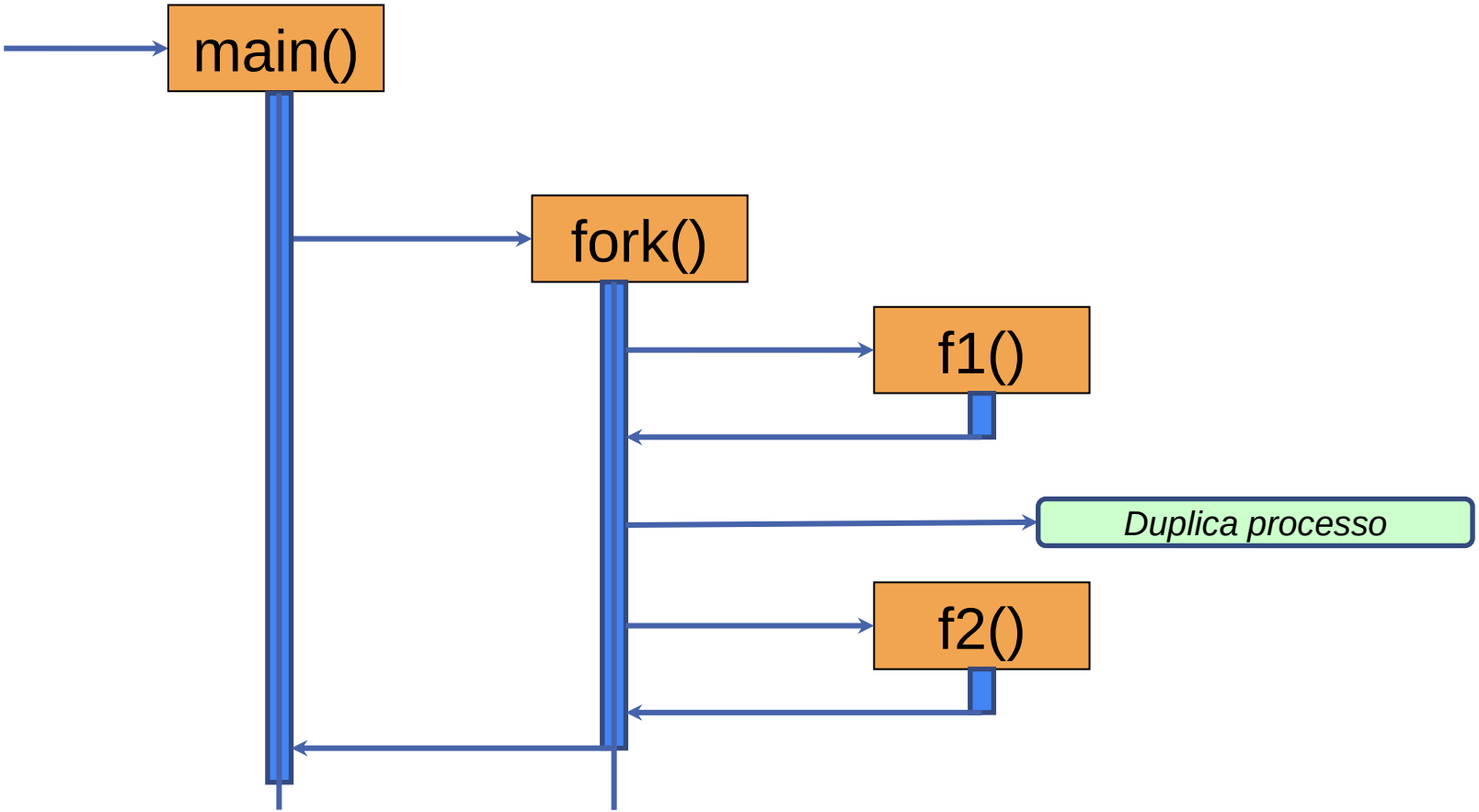
- Nel caso di programmi concorrenti, l'esecuzione di fork() crea un problema
  - Il processo figlio conterrà un solo thread
  - Gli oggetti di sincronizzazione presenti nel padre possono trovarsi in stati incongruenti
- ```
  int pthread_atfork(
      void (*prepare)(void),
      void (*parent)(void),
      void (*child)(void)
  );
  ```
  - Registra un gruppo di funzioni che saranno chiamate in corrispondenza delle invocazioni a fork()
  ```

Esempio

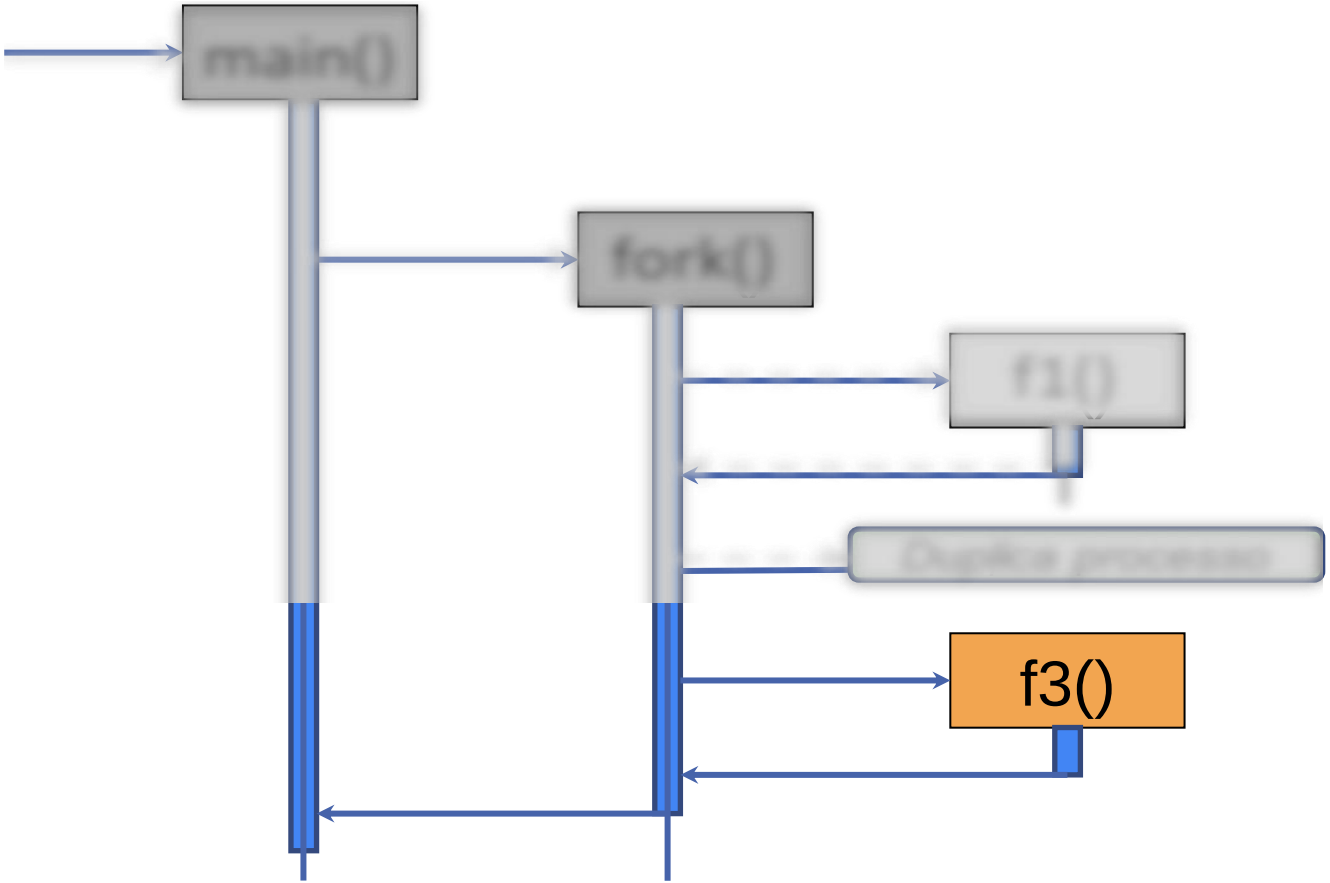
```
void f1() { … }
void f2() { … }
void f3() { … }

int main() {
   pthread_atfork(f1,f2,f3);
   //…
   int res= fork();
   if (res== -1 ) { /* errore */  }
   else if (res == 0) { /* child */ }
   else {      /* parent */ }
}
```

Processo genitore



Processo figlio



Terminare un processo

- Un processo continua la propria esecuzione fino a che non termina di propria volontà o viene terminato dall'esterno
  - Una opportuna system call permette di richiedere la terminazione del processo corrente e la conseguente deallocazione di tutte le risorse in uso (memoria, descrittori di file, lock e altri oggetti di sincronizzazione, primitive di IPC, socket di rete, …) e la loro restituzione al sistema operativo, che potrà renderle disponibili ad altri processi

Terminare un processo

- In Windows si invoca **`ExitProcess(int status)`**
  - In Linux la funzione **`_exit(int status)`**
- Entrambe causano l'IMMEDIATA terminazione di tutti gli altri thread associati al processo
  - Senza ulteriori possibilità di esecuzione
- Tutti i file aperti sono chiusi
  - Nel caso di Windows, nel contesto del thread che ha eseguito la funzione **`ExitProcess(…)`** vengono rilasciate le DLL eventualmente caricate

Terminare un processo

- Le librerie standard del C e del C++ offrono una soluzione portabile e più articolata
  - la funzione **`void exit(int status)`** definita in <stdlib.h>
  - la funzione **`void std::exit(int status)`** definita in <cstdlib>
- Queste supportano la possibilità di registrare una o più callback da invocare all'atto della terminazione
  - Tramite la funzione **`int std::atexit(void (*callback)())`**
  - Il valore restituito indica se la registrazione è andata a buon fine o meno

Gestire la terminazione

```
#include <iostream>
#include <cstdlib>

void atexit_handler_1() {
  std::cout << "at exit #1\n";
}

void atexit_handler_2() {
  std::cout << "at exit #2\n";
}

int main() {
  const int result_1 = std::atexit(atexit_handler_1);
  const int result_2 = std::atexit(atexit_handler_2);
  if ((result_1 != 0) || (result_2 != 0)) {
    std::cerr << "Registration failed\n";
    return EXIT_FAILURE;
  }
  std::cout << "returning from main\n";
  return EXIT_SUCCESS;
}
```

Gestire la terminazione

- Un programma termina anche quando la funzione principale ritorna
  - Questo è conseguenza del codice di startup inserito dalla libreria di supporto del linguaggio
  - Questa, inizializza l'ambiente di esecuzione, invoca la funzione **`main(…)`** e utilizza il valore da essa ritornato per invocare **`exit(status)`**
- Se si verifica un'eccezione non gestita nel thread principale o in uno secondario
  - Viene invocata la funzione **`ExitProcess(…)`** o **`_exit(…)`** con un codice di stato definito dalla libreria di esecuzione

Codice di ritorno

- Il valore restituito dalla funzione **`exit(…)`** è arbitrario
  - Vale una convenzione generale per la quale 0 indica una terminazione "pulita" e qualunque altro valore indica una terminazione con errore
  - Il significato di tale codice, tuttavia, non è oggetto di specifica da parte del sistema operativo
  - Occorre documentare opportunamente il valore e attenersi alle buone pratiche definiti in ciascun ambiente (OS + compilatore + libreria standard)

Processi in Rust

- Per la gestione dei processi, la standard library di Rust mette a disposizione il modulo std::process
  - I metodi offerti da Rust utilizzano al loro interno le system call esposte dal kernel del sistema operativo per gestire i processi
- La struct **`Command`** permette la creazione di un nuovo processo
  - Utilizza il pattern builder per configurare, creare ed interagire con il processo figlio
  - I metodi **`arg()`** ed **`args()`** possono essere utilizzati per passare al processo figlio rispettivamente uno o più argomenti
  - Il metodo **`output()`** genera il processo ed attende la sua terminazione ritornando un valore di tipo **`Result<Output>`**

```
pub struct Output {
    pub status: ExitStatus,
    pub stdout: Vec<u8>,
    pub stderr: Vec<u8>,
}
```

Processi in Rust

- Per la gestione dei processi, la standard library di Rust mette a disposizione il modulo std::process
  - I metodi offerti da rust utilizzano al loro interno le system call esposte dal kernel del sistema operativo per gestire i processi
- La struct **`Command`** permette la creazione di un nuovo processo
  - Utilizza il pattern builder per configurare, creare ed interagire con il processo figlio
  - I metodi **`arg()`** ed **`args()`** possono essere utilizzati per passare al processo figlio rispettivamente uno o più ar
  - Il metodo **`output()`** genera
    tipo **`Result<Output>`**

```
pub struct Output {
    pub status: ExitStatus,
    pub stdout: Vec<u8>,
    pub stderr: Vec<u8>,
}
```

`ExitStatus` è una struttura che permette di avere informazioni sul codice di uscita del processo e, nel caso di sistemi Unix, sulla motivazione della sua terminazione (fine dell'esecuzione, terminazione forzata tramite un segnale, sospensione/continuazione a seguito di un segnale, eventuale presenza di una copia dello spazio di indirizzamento - *core dump*)

Processi in Rust

```
use std::process::Command;

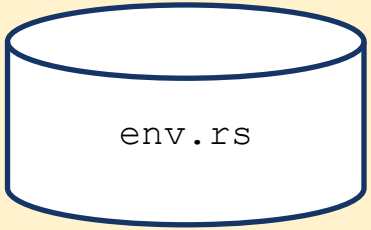
fn main() {
  let output = if cfg!(target_os = "windows") {
    Command::new("cmd")
            .args(["/C", "echo hello"])
            .output()
            .expect("failed to execute process")
  } else {
    Command::new("sh")
            .arg("-c")
            .arg("echo hello")
            .output()
            .expect("failed to execute process")
  };

  println!("{:?}", output)
  //Output { status: ExitStatus(unix_wait_status(0)), stdout: "hello\n", stderr: "" }
}
```

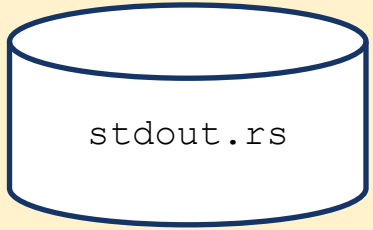
Processi in Rust

- E’ possibile configurare le variabili d’ambiente del processo prima di avviarlo
  - Per default, esso eredita quelle del processo corrente
  - I metodi `env<K,V>(&mut self, key: K, val: V)` e `envs<I,K,V>(&mut self, vars: I)` permettono di effettuare aggiunte o modifiche
  - I metodi `env_remove<K>(&mut self, key: K)` e `env_clear(&mut self)` permettono di eliminare una/tutte le variabili d’ambiente
  - E’ possibile conoscere l’elenco delle variabili d’ambiente usate da una struct di tipo Command con il metodo `get_envs(&self)` che restituisce un iteratore a tuple formate dalle coppie chiave/valore
- E’ possibile ridirigere i flussi standard di ingresso/uscita, tramite i metodi `stdin(...)`, `stdout(...)` e `stderr(...)`
  - E’ possibile passare loro una delle seguenti opzioni:
  - `inherit()` → Il processo figlio eredita il descrittore in uso nel processo genitore
  - `piped()` → Verrà creata una pipe monodirezionale, un’estremità della quale sarà passata al processo figlio mentre l’altra sarà memorizzata nella struttura restituita dal comando di avvio
  - `null()` → Il flusso sarà ignorato
  - default ad `inherit` quando si usa `spawn` o `status`, e default a `piped` quando si usa `output`.
- E’ possibile modificare la cartella in cui si avvia il processo figlio
  - Tramite il metodo `current_dir<P: AsRef<Path>>(&mut self, dir: P)`

```rust
use std::process::Stdio;
use std::process::Command;

fn main() {

    let output = Command::new("ls")
        .env("PATH", "/bin")
        .stdout(Stdio::inherit())
        .output()
        .expect("ls command failed to start");
    }
```



```rust
use std::process::Stdio;
use std::process::Command;

fn main() {

    let mut output = Command::new("echo")
        .arg("Hello, world!")
        .stdout(Stdio::null()) // This stream will be ignored.
                               // equivalent of attaching the stream to /dev/null
        .output()
        .expect("Failed to execute command");
        println!("{:?}", output);

    output = Command::new("echo")
        .arg("Hello, world!")
        .stdout(Stdio::inherit()) // The child inherits from the corresponding parent descriptor
        .output()
        .expect("Failed to execute command");
    println!("{:?}", output);

    output = Command::new("echo")
        .arg("Hello, world!")
        .stdout(Stdio::piped()) //  A new pipe should be arranged to connect the parent and child processe
        .output()
        .expect("Failed to execute command");
    println!("{:?}", output);
    }
```

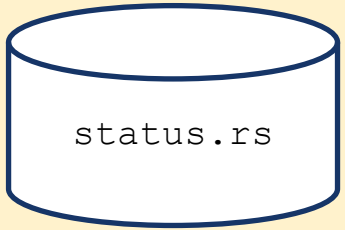



```
use std::process::Stdio;
use std::process::Command;

fn main() {
    let output = Command::new("ls")
         .current_dir("/bin")
        .stdout(Stdio::inherit())
        .output()
        .expect("ls command failed to start");
    }
```

`status()`

- Il metodo **`status(&mut self)`** avvia il processo, ne attende la terminazione
  - Esso restituisce un valore di tipo **`Result<ExitStatus>`**, dove **`ExitStatus`** è una struttura che permette di avere informazioni sul codice di uscita del processo e, nel caso di sistemi Unix, sulla motivazione della sua terminazione (fine dell'esecuzione, terminazione forzata tramite un segnale, sospensione/continuazione a seguito di un segnale, eventuale presenza di una copia dello spazio di indirizzamento - *core dump*)

```rust
use std::process::Command;

fn main() {
    // Esegui il comando ls e ottieni il suo stato di uscita
    let child = Command::new("ls")
        .status()
        .expect("failed to execute process");

    // Stampa lo stato di uscita del processo
    if child.success() {
        println!("Process exited successfully with status: {}", child);
    } else {
        println!("Process exited with failure status: {}", child);
    }
}
```



spawn()

- Il metodo **`spawn(&mut self)`** avvia invece il processo senza attenderne la terminazione
  - Esso restituisce un valore di tipo **`Result<Child>`**, dove **`Child`** è una struttura che consente di rappresentare ed interagire con il processo figlio

Le 3 modalità

- **output:** genera il processo figlio ed attende la sua terminazione ritornando un valore di tipo **`Result<Output>`**; **`stdin`** e **`stdout`** di default a **`piped`**().

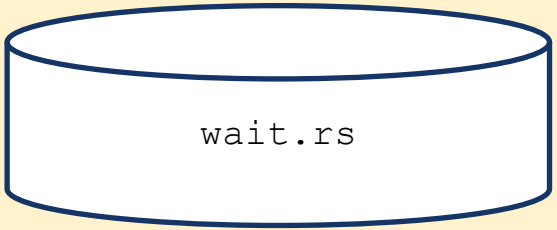
- **status**: genera il processo figlio ed attende la sua terminazione ritornando un valore di tipo **`Result<ExitStatus>`**; **`stdin`** e **`stdout`** di default a **`inherit`**().

- **spawn**: genera il processo figlio e restituisce un valore di tipo **`Result<Child>`** e non attende la terminazione del processo; **`stdin`** e **`stdout`** di default a **`inherit`**().

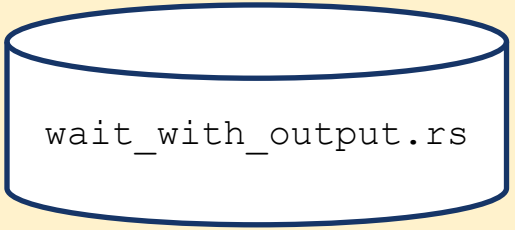
Processi in Rust

- La struttura Child offre vari meccanismi per controllare e condizionare lo svolgimento del processo figlio
  - Attraverso i campi **`stdin`**, **`stdout`**, **`stderr`** è possibile fare accesso alle handle dei relativi flussi, qualora siano stati catturati
  - Il metodo **`id(&self)`** restituisce l'identificativo univoco assegnato dal sistema operativo
  - Il metodo **`wait(&mut self)`** ne attende la terminazione, restituendo il codice di uscita
  - Il metodo **`wait_with_output(&mut self)`** chiude il flusso di ingresso del processo figlio, ne attende la terminazione e raccoglie quanto non ancora letto dei flussi di uscita ed errore in una struttura di tipo **`Output`**
  - il metodo **`kill()`** ne forza la terminazione

```rust
use std::process::Command;

fn main() {
        // Crea un'istanza di Command per il comando "echo" con l'argomento "Hello World"
        let mut process = Command::new("echo")
            .arg("Hello World")
            .spawn()
            .expect("Failed to spawn command");

        // Attendere che il processo figlio termini
        let exit_status = process.wait();

        // Verificare lo stato di uscita del processo figlio
      if exit_status.expect("Errore nel processo spawned").success()
        { println!("Child process has exited successfully."); }
}
```



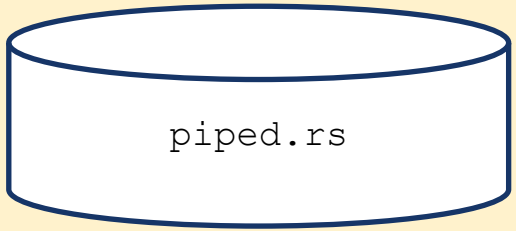
```rust
use std::process::{Command, Stdio};

fn main() {
    let child_process = Command::new("ls")
            .arg("-l")
            .arg("-a")
            .stdout(Stdio::piped())
            .spawn()
            .expect("Impossibile avviare il processo 'ls'");

    let output = child_process.wait_with_output().expect("Impossibile ottenere l'output del processo");

    if output.status.success() {
                let stdout = String::from_utf8_lossy(&output.stdout);
                println!("Output del processo 'ls':\n{}", stdout);
      } else {
                let stderr = String::from_utf8_lossy(&output.stderr);
                println!("Errore durante l'esecuzione del processo 'ls': {}", stderr);
       }
}
```



```rust
use std::process::{Command, Stdio};

fn main() {
        // Esegue un comando esterno
        let mut child = Command::new("sleep")
            .arg("10")
            .stdout(Stdio::piped())
            .spawn()
            .expect("Failed to start command");

        // Ottiene l'ID del processo
        let pid = child.id();

        // Termina il processo
        match child.kill() {
                Ok(_) => println!("Process {} killed", pid),
                Err(err) => println!("Error killing process {}: {}", pid, err),
        }
}
```

Ridirigere i flussi di ingresso/uscita

```
use std::io::prelude::*;
use std::process::{Command, Stdio};
fn main() {
    let process = match Command::new("rev")
        .stdin(Stdio::piped())
        .stdout(Stdio::piped())
        .spawn()
    {
        Err(err) => panic!("couldn't spawn rev: {}", err),
        Ok(process) => process,
    };
    match process.stdin.unwrap().write_all("'isoc ovitrevid im non ehc innaaaa
onarE".as_bytes()) {
        Err(why) => panic!("couldn't write to stdin: {}", why),
        Ok(_) => println!("sent text to rev command"),
    }
    let mut child_output = String::new();
    match process.stdout.unwrap().read_to_string(&mut child_output) {
        Err(err) => panic!("couldn't read stdout: {}", err),
        Ok(_) => print!("Output from child process is:\n{}", child_output),
    }
}
```

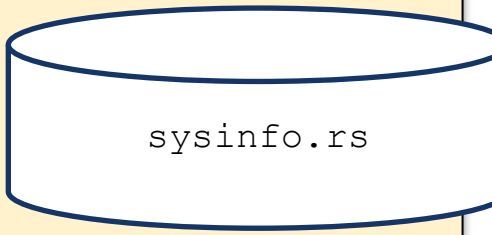


sysinfo

- Il crate `sysinfo` permette di accedere alle informazioni dei processi così come esposte dai diversi sistemi operativi

```
[dependencies]
sysinfo = "0.30.12"
```

```rust
use sysinfo::{Pid, System};
fn main()  {
    let mut sys = System::new();
// First we update all information of our `System` struct.
    sys.refresh_all();
    println!("=> system:");
// RAM and swap information:
    println!("total memory: {} bytes", sys.total_memory());
    println!("used memory : {} bytes", sys.used_memory());
    println!("total swap  : {} bytes", sys.total_swap());
    println!("used swap   : {} bytes", sys.used_swap());
// Display system information:
    println!("System name:             {:?}", System::name());
    println!("System kernel version:   {:?}", System::kernel_version());
    println!("System OS version:       {:?}", System::os_version());
    println!("System host name:        {:?}", System::host_name());
// Number of CPUs:
    println!("Number CPUs: {}", sys.cpus().len());
// Display processes ID, name na disk usage:
    for (pid, process) in sys.processes() {
        println!("[{pid}] {} {:?}", process.name(), process.disk_usage());
    }
}
```



Interazioni tra fork(), stdio e exit()

- Se, in Linux, un processo esegue **`fork()`**, tutto lo stato della sua memoria sarà duplicato
  - Nel caso in cui tale processo avesse avuto, nei buffer di IO contenuto pendente, tale contenuto verrà scaricato sul dispositivo corrispondente sia dal processo padre che da quello figlio, non appena eseguiranno una operazione di **`flush()`** o satureranno tale buffer con ulteriori operazioni
- Il problema è più evidente quando l'output di un programma è rediretto verso un file
  - Questo abilita una modalità di scrittura a blocchi (al posto di quella standard a linee) che aumenta la probabilità di osservare tale fenomeno
  - Prima di eseguire **`fork()`**, può essere conveniente eseguire un **`flush()`** di tutti i file aperti (compresi std::cout e std::cerr)

Terminare un processo

- La funzione `std::process:exit(code: i32) -> !` termina immediatamente il processo corrente, con tutti i thread presenti al suo interno
  - Nessun distruttore presente sullo stack del thread corrente né degli altri thread viene eseguito
  - E' responsabilità del programmatore invocare questa funzione solo in punti in cui si abbia la ragionevole sicurezza che tutto ciò che doveva essere liberato sia stato liberato
  - Nonostante il valore ricevuto sia a 32 bit, nella maggior parte dei sistemi Unix-like, solo gli 8 bit meno significativi sono effettivamente passati al sistema operativo
- La funzione `std::process:abort() -> !` termina immediatamente il processo corrente, con tutti i thread presenti al suo interno
  - E forza un codice di errore che viene interpretato come interruzione anomala
  - Valgono le stesse cautele espresse per la funzione soprastante
- La macro `panic!(...)` causa la contrazione dello stack corrente, con l'esecuzione di tutti i distruttori posti al suo interno, senza determinare la terminazione del processo
  - A meno che la chiamata avvenga nel contesto del thread principale

Gestire altri processi

- Ciascun sistema operativo offre meccanismi per attendere la terminazione di un processo di cui si conosce la handle
  - Tale attesa non comporta consumo di CPU e termina quando il processo osservato termina per qualunque motivo
- Le API offerte dai diversi sistemi operativi differiscono alquanto
  - In Windows si utilizzano **`WaitForSingleObject(…)`** / **`WaitForMultipleObjects(…)`** e **`GetExitCodeProcess(…)`**
  - In Linux, si utilizzano **`wait(…)`**, **`waitpid(…)`** e **`waitid(…)`**

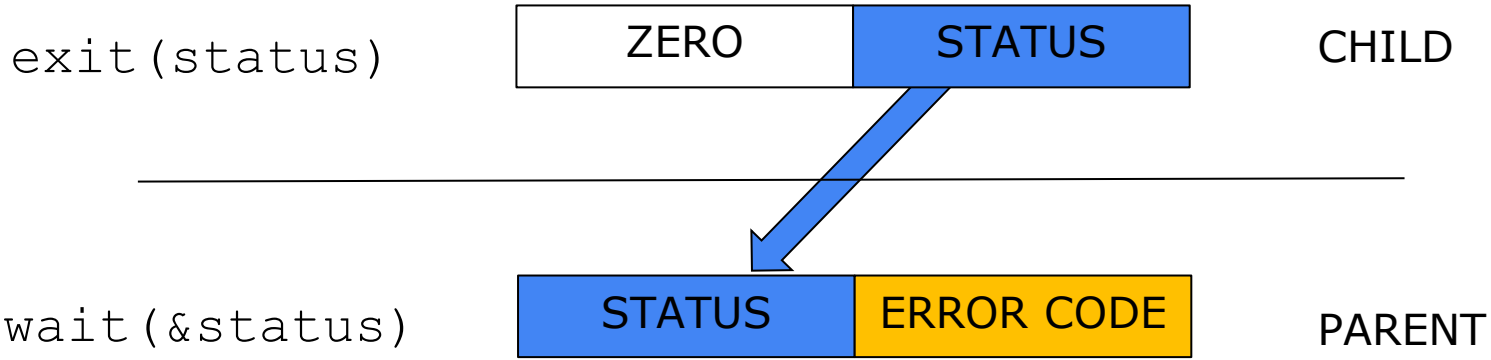
Gestire altri processi

- La funzione **`pid_t wait(int *status)`**, definita in <sys/wait.h>, opera come segue:
  - Se un processo non ha processi figli, la funzione ritorna -1
  - Se nessuno dei figli dell'attuale processo è terminato, la chiamata si blocca in attesa di tale evento
  - In presenza di un figlio terminato:
    - Se status non è nullo, al suo interno viene indicata la motivazione che ne ha causato la terminazione
    - Il sistema aggiorna i contatori di utilizzo del sistema (CPU/memoria/IO/rete) del processo corrente aggiungendo quelli del processo figlio
    - Viene restituito il ProcessID del figlio terminato
    - Eventuali ulteriori chiamate a **`wait(…)`** da parte del processo corrente non forniranno più indicazioni relative a questo processo figlio

Gestire altri processi

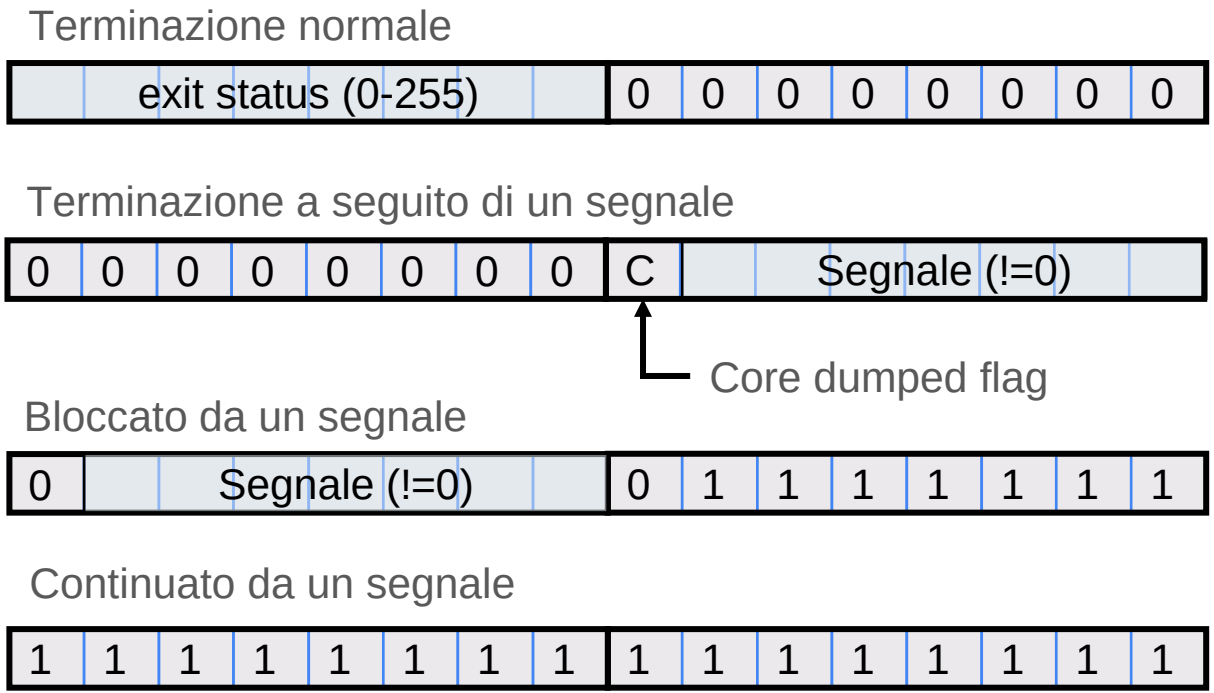
- La funzione **`pid_t waitpid(pid_t pid, int *status, int options)`** permette di indicare uno specifico processo figlio
  - E anche di eseguire una verifica non bloccante del suo stato attuale
  - Attenzione all'uso del polling che può causare consumi eccessivi di CPU e batteria!
- L'intero a cui punta status (se non è nullo) viene inizializzato con un valore a 16 bit
  - Il suo contenuto è definito in una serie di sotto-campi

Valore ritornato al processo padre



Valore ricevuto dal processo padre

Stato terminale di un processo in Linux



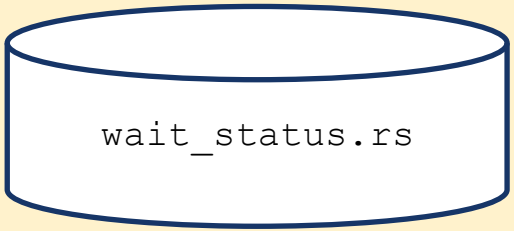
```rust
use std::process::Command;

fn main() {
    // Esegui un comando che si sa restituirà un valore di uscita specifico
    let command = Command::new("ls")
        .arg("/nonexistent_directory")
        .spawn();

    match command {
        Ok(mut child) => {
            // Attendere il completamento del processo figlio e ottenere l'ExitStatus
            match child.wait() {
                Ok(exit_status) => {
                    // Verificare se il processo è terminato correttamente o con un errore
                    if exit_status.success() {
                        println!("Il processo è terminato correttamente.");
                    } else {
                        println!("Il processo è terminato con errore.");
                    }

                    // Ottenere il codice di uscita del processo se disponibile
                    if let Some(code) = exit_status.code() {
                        println!("Codice di uscita del processo: {}", code);
                    } else {
                        println!("Il processo è stato terminato da un segnale.");
                    }
                }
                Err(e) => {
                    eprintln!("Errore durante l'attesa del processo: {}", e);
                }
            }
        }
        Err(e) => {
            eprintln!("Errore durante l'avvio del processo: {}", e);
        }
    }
}
```

```rust
use std::process::{Command, Stdio};
use std::os::unix::process::ExitStatusExt;
// ExitStatusExt is an extension trait for extracting any such signal, and other details, from the ExitStatus.
fn main() {
    // Esempio: avvia un processo figlio
    let mut child_process =  Command::new("timer")
        .env("PATH", "./target/debug/")
        .stdin(Stdio::piped())
        .stdout(Stdio::piped())
        .spawn()
        .unwrap();
    let child_pid = child_process.id(); // Ottieni l'ID del processo figlio
    let result = Command::new("kill") // Invia il segnale SIGTERM al processo figlio
        .arg("-15") // Codice del segnale SIGTERM
        .arg(child_pid.to_string())
        .status();
    match result {
        Ok(status) => {
            if status.success() {
                println!("Segnale SIGTERM inviato al processo figlio.");
            } else {
                println!("Errore nell'invio del segnale.");
            }
        }
        Err(_) => { println!("Errore nell'esecuzione del comando kill."); }
    }
    match child_process.wait() {
        Ok(exit_status) => {
            // Verificare se il processo è terminato correttamente o con un errore
            if exit_status.success() {
                println!("Il processo è terminato correttamente.");
            } else {
                println!("Il processo è terminato con errore: {:?}",exit_status);
            }
            // Ottenere il codice di uscita del processo se disponibile
            if let Some(code) = exit_status.code() {
                println!("Codice di uscita del processo: {}", code);
            } else {
                // Utilizza ExitStatusExt per ottenere informazioni
                if let Some(signal) = exit_status.signal() {
                    println!("Processo terminato dal segnale: {}", signal);
                }
                if exit_status.core_dumped() {
                    println!("Il processo ha generato un core dump");
                }
            }
        }
        Err(e) => { eprintln!("Errore durante l'attesa del processo: {}", e); }
    }
}
```

signal.rs

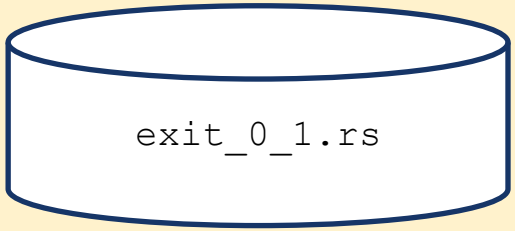
timer.rs

```rust
use std::time::Duration;
use std::thread::sleep;
fn main() {
    for i in 1..100 {
      sleep(Duration::from_secs(1));
        println!("{i}");
    }
}
```



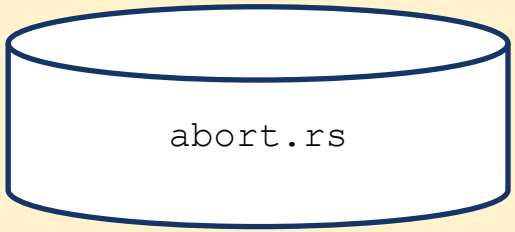

```rust
use std::process::{Command, Stdio};
use std::os::unix::process::ExitStatusExt;
fn main() {
    let mut child_process =  Command::new("exituno")
        .env("PATH", "./target/debug/")
        .stdin(Stdio::piped())
        .stdout(Stdio::piped())
        .spawn()
        .unwrap();
    let child_pid = child_process.id(); // Ottieni l'ID del processo figlio
    match child_process.wait() {
        Ok(exit_status) => {
            // Verificare se il processo è terminato correttamente o con un errore
            if exit_status.success() {
                println!("Il processo è terminato correttamente.{:?}",exit_status);
            } else {
                println!("Il processo è terminato con errore: {:?}",exit_status);
            }
            // Ottenere il codice di uscita del processo se disponibile
            if let Some(code) = exit_status.code() {
                println!("Codice di uscita del processo: {}", code);
            } else {
                // Utilizza ExitStatusExt per ottenere informazioni
                if let Some(signal) = exit_status.signal() {
                    println!("Processo terminato dal segnale: {}", signal);
                }
                if exit_status.core_dumped() {
                    println!("Il processo ha generato un core dump");
                }
            }
        }
        Err(e) => { eprintln!("Errore durante l'attesa del processo: {}", e); }
    }
}
```

```rust
use std::process;

fn main() {
    println!("Esempio di programma in Rust che usa exit()");

    // Condizione per terminare il programma
    let some_condition = true;

    if some_condition {
        println!("Condizione soddisfatta, il programma terminerà con codice 1.");
        process::exit(1);
    }
    // Se il programma non è terminato prima, proseguirà con altre operazioni

    // Terminare il programma con codice 0 (successo)
    process::exit(0);
}
```



- Se il processo termina con **`exit(1)`**
  - Il padre legge **`ExitStatus(unix_wait_status(256))`** => **`exit_status.code()`** ritorna il codice 1
- Se il processo termina con **`exit(0)`**
  - Il padre legge **`ExitStatus(unix_wait_status(0))`** => **`exit_status.code()`** ritorna il codice 0

```rust
use std::process;

fn main() {
    // Esegui alcune operazioni
    println!("Esempio di programma in Rust che usa abort()");

    // Condizione per terminare il programma con un abort
    let some_condition = true;

    if some_condition {
        println!("Condizione soddisfatta, il programma terminerà con abort.");
        process::abort();
    }

    // Se il programma non è terminato prima, proseguirà con altre operazioni
    println!("Il programma continuerà a funzionare normalmente.");
}
```



- Se il processo termina con **`abort()`**
  - Il padre legge **`ExitStatus(unix_wait_status(6))`** => **`exit_status.signal()`** ritorna il codice 6 corrispondente al signal SIGABRT.

```rust
fn main() {
    // Esegui alcune operazioni
    println!("Esempio di programma in Rust che usa panic!");

    // Condizione per far scattare il panic
    let some_condition = true;

    if some_condition {
        println!("Condizione soddisfatta, il programma terminerà con un panic.");
        panic!("Errore: si è verificato un panic!");
    }

    // Se il programma non è terminato prima, proseguirà con altre operazioni
    println!("Il programma continuerà a funzionare normalmente.");
}
```

- Se il processo termina con **`panic()`**
  - Il padre legge **`ExitStatus(unix_wait_status(25856))`** => **`exit_status.code()`** ritorna il codice 101.

Orfani e zombie

- Se un processo padre termina prima che l'ultimo dei suoi figli sia terminato, il processo figlio diventa orfano
  - Linux riassegna a tale figlio il PID 1 (init) come ID del processo padre
  - Questo può essere sfruttato da un processo per sapere se il proprio padre è ancora vivo o meno (ipotizzando che non sia stato lanciato direttamente da init)
- Se un processo figlio termina prima che il rispettivo padre abbia eseguito `wait*(…)` diventa uno zombie
  - La maggior parte delle sue risorse viene restituita al sistema operativo, tranne l'ID, lo stato di terminazione e i contatori relativi all'utilizzo delle risorse
  - Quando il padre effettua una `wait(…)` lo zombie viene rimosso

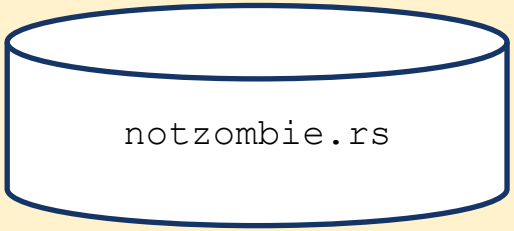
```rust
use std::process::{Command};
use std::thread;
use std::time::Duration;

fn main() {
    // Creazione di un processo figlio
    let mut child = Command::new("sleep")
        .arg("5")
        .spawn()
        .expect("failed to start child process");

    // Simulare un ritardo nel processo genitore
    println!("Waiting for the child process to finish...");
    thread::sleep(Duration::from_secs(7));

    // Attesa del processo figlio per evitare che diventi zombie
    let status = child.wait()
                      .expect("failed to wait on child process");

    println!("Child process exited with status: {}", status);
}
```



IPC - InterProcess Communication

- Il S.O. impedisce il trasferimento diretto di dati tra processi
  - Ogni processo dispone di uno spazio di indirizzamento separato
  - Non è possibile sapere cosa sta capitando in un altro processo
- Ogni S.O. offre alcuni meccanismi per superare tale barriera in modo controllato
  - Permettendo lo scambio di dati…
  - …e la sincronizzazione delle attività

Rappresentazione delle informazioni scambiate

- Indipendentemente dal tipo di meccanismo adottato, occorre adattare le informazioni scambiate
  - Così da renderle comprensibili al destinatario

Rappresentazione interna

- Internamente, un processo può usare una varietà di rappresentazioni
  - Tipi elementari (numerici, logici, caratteri, …)
  - Tipi strutturati (record, array, classi, …)
  - Puntatori per strutture dati complesse (alberi, grafi, …)
- La rappresentazione interna non è adatta ad essere esportata
  - I puntatori non hanno senso al di fuori del proprio spazio di indirizzamento
  - Alcune informazioni (handle) non sono esportabili

Rappresentazione esterna

- Formato intermedio che permette la rappresentazione di strutture dati arbitrarie
  - Sostituendo i puntatori con riferimenti indipendenti dalla memoria
- Formati basati su testo
  - XML, JSON, CSV, …
- Formati binari
  - XDR, HDF, protobuf …

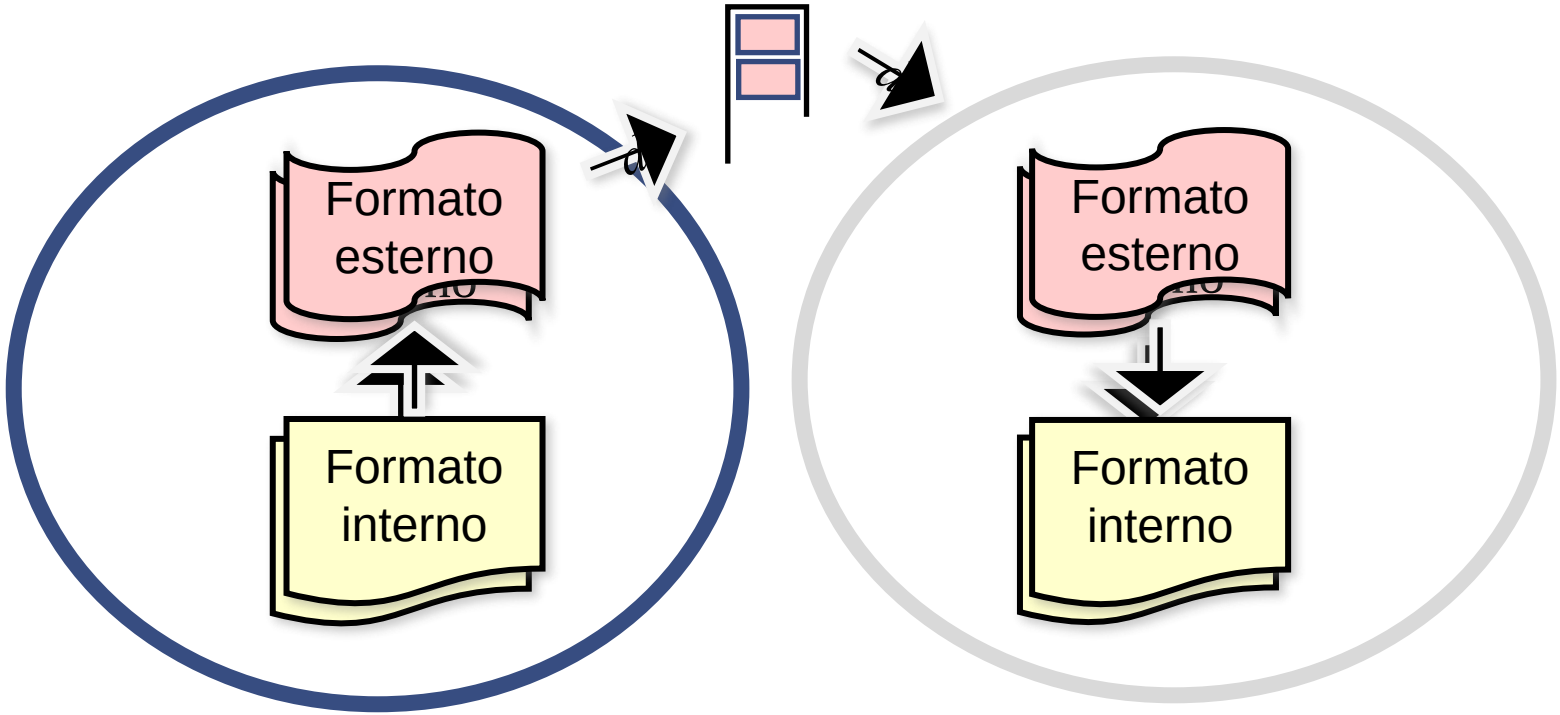
Serializzazione

- Le rappresentazioni esterne possono essere trattate come blocchi compatti di byte
  - Possono essere duplicati e trasferiti senza comprometterne il significato
- I dati vengono scambiati nel formato esterno
  - La sorgente esporta le proprie informazioni (marshalling)
  - Il destinatario ricostruisce una rappresentazione su cui può operare direttamente (unmarshalling)
- La ricostruzione operata dal destinatario può corrispondere o meno a quella utilizzata dalla sorgente
  - In certi casi può essere conveniente trasformare la struttura dati in un formato più consono alle operazioni che si intendono fare su di essa
- Le operazioni di marshalling e unmarshalling possono essere codificate esplicitamente dal programmatore
  - O essere eseguite da codice generato automaticamente dall'ambiente di sviluppo

Coda di messaggi

- Struttura dati mantenuta dal sistema operativo che permette a molti processi sorgente di inviare messaggi ad uno specifico destinatario
  - La comunicazione è di tipo asincrono, il sistema operativo si fa carico di memorizzare il messaggio inviato fino a che il destinatario non lo leggerà
  - La comunicazione è mono-direzionale
- Ogni coda è caratterizzata da un identificativo univoco a livello di sistema operativo (una stringa)
  - Il destinatario della coda è responsabile di inizializzarla e legarla al suo nome
  - Le sorgenti devono avere modo di conoscere a priori il nome della coda o servirsi di altri meccanismi di IPC per venirne a conoscenza
- Gli oggetti "mailslot" in Windows e "fifo" in Linux ricadono in questa categoria
  - Vengono costruiti in C/C++ tramite le funzioni **`CreateMailslot(...)`** e **`mkfifo(...)`**

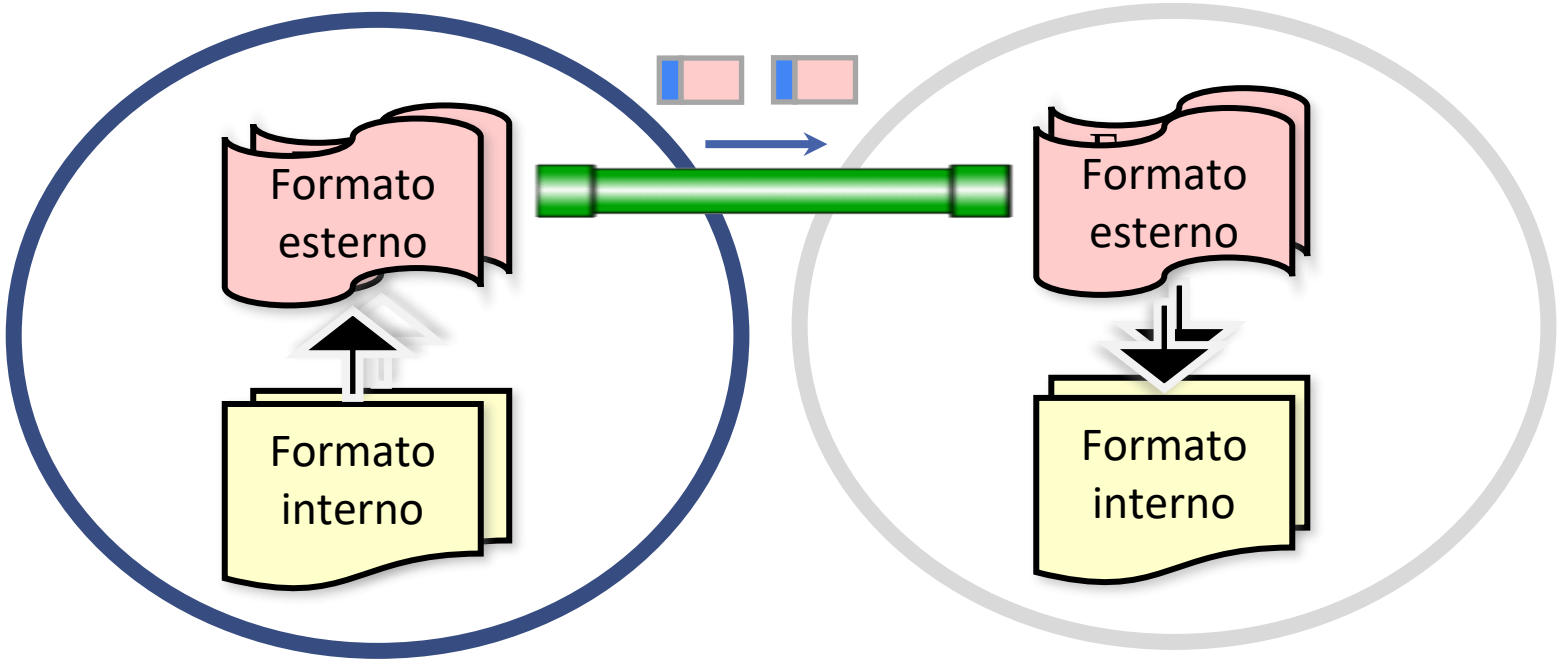
Coda di messaggi



Pipe

- «Tubi» che permettono il trasferimento di sequenze di byte di dimensioni arbitrarie
  - Occorre inserire marcatori che consentano di delimitare i singoli messaggi
  - Comunicazione sincrona 1-1
- Implementati come buffer all'interno della memoria del kernel
  - Disponibili sia in Linux che Windows con alcune differenze sul modo di fornire i descrittori ai processi derivati

Pipe



Pipe

- In C/ C++, una pipe può essere creata da programma con le funzioni
  - ```
    BOOL CreatePipe(
         HANDLE *pHRead,
         HANDLE *pHWrite,
         SECURITY_ATTRIBUTES *lpPipeAttributes,
         DWORD nSize)
    ```
    – Windows (#include <namedpipeapi.h>)
  - `int pipe(int fd[2])`
    – Linux (#include <unistd.h>)
- Si legge e scrive da una pipe con le operazioni di lettura/scrittura del sistema operativo
  - `FileRead(…)` / `FileWrite(…)` – Windows
  - `read(…)` / `write(…)` – Linux
- Si chiude una pipe con la funzione corrispondente
  - `CloseHandle(…)` / `close(…)`
    ```

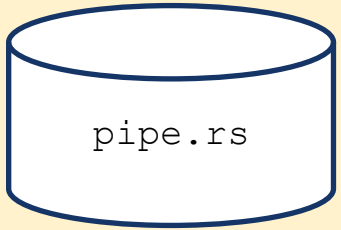
Pipe in Rust

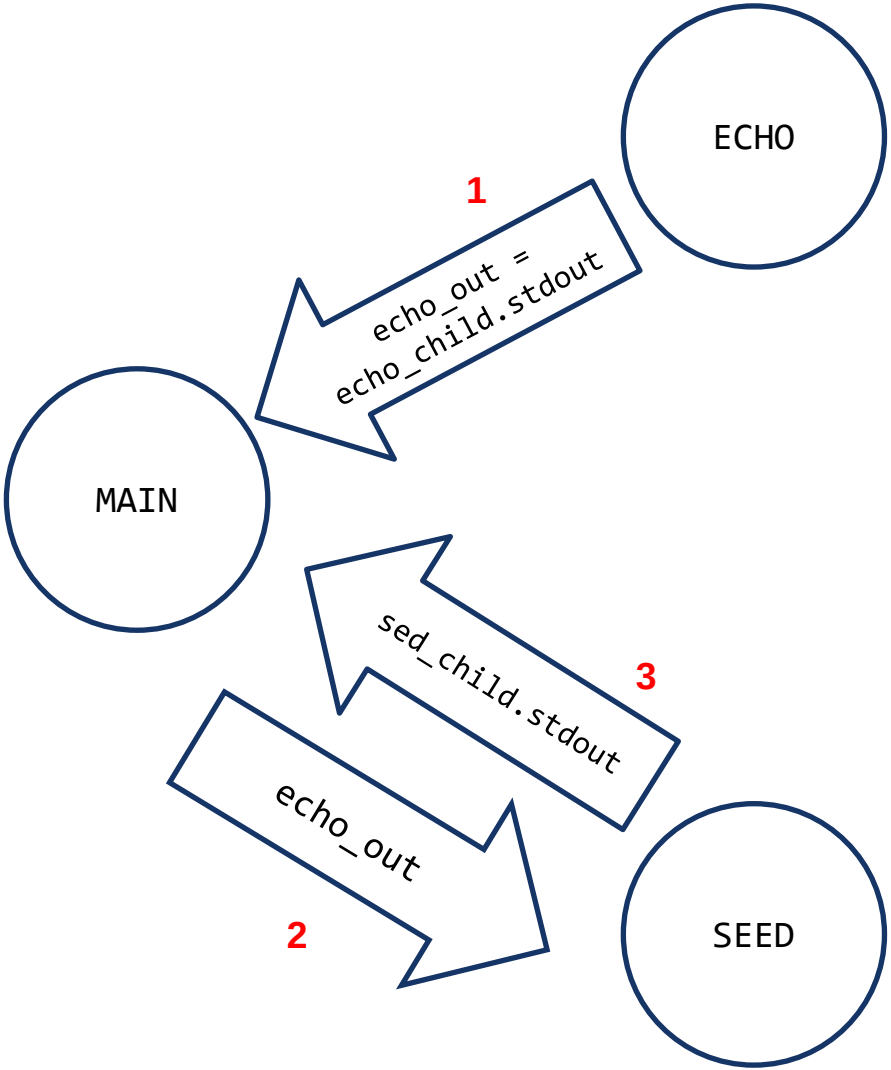
```
use std::process::Stdio;
use std::process::Command;
fn main(){
    let echo_child = Command::new("echo")
        .arg("Ciao Mama")
        .stdout(Stdio::piped())
        .spawn()
        .expect("Failed to start echo process");

    let echo_out = echo_child.stdout.expect("Failed to open echo stdout");

    let sed_child = Command::new("sed")
        .arg("s/Mama/Mamma/")
        .stdin(Stdio::from(echo_out))
        .stdout(Stdio::piped())
        .spawn()
        .expect("Failed to start sed process");

    let output = sed_child.wait_with_output().expect("Failed to wait on sed");
    println!("{}", String::from_utf8_lossy(&output.stdout));
}
```





Named pipe

- **Named pipe**, chiamate anche FIFO (first in, first out), sono meno restrittive delle pipe e offrono i seguenti vantaggi:
  - esistono nel file system
  - possono essere utilizzitati da processi che non hanno relazione di parentela padre/figlio
  - esistono fino a quando non sono esplicitamente cancellati
  - hanno una capacità di buffer superiore (tipicamente intorno a 40Kbyte)

- Come le unnamed pipe, sono utilizzabili sono in modo unidirezionale

- Una named pipe è creata con il commando `mkfifo`

- La named pipe deve essere inizialmente aperta con la funzione `open()`
- Le funzioni di scrittura aggiungono dati all'inizio della coda FIFO
- Le funzioni di lettura prelevano dati dalla fine della coda FIFO.

```rust
use std::process::Command;
use std::fs::OpenOptions;
use std::io::Write;
use std::path::Path;
fn main() {
    let pipe_path = "/tmp/my_pipe";

   Command::new("reader") // Avvia il processo lettore
        .env("PATH", "./target/debug/")
        .spawn()
        .expect("Failed to start child process");

    // Check if the named pipe exists
    if ! Path::new(pipe_path).exists() {
        eprintln!("Named pipe {} does not exist. Please create it first using mkfifo.", pipe_path);
        return;
    }

    // Open the named pipe for writing
    let mut pipe = match OpenOptions::new().write(true).open(pipe_path) {
        Ok(pipe) => pipe,
        Err(e) => {
            eprintln!("Failed to open named pipe: {}", e);
            return;
        }
    };

    let message = "Hello from Rust!\n";

    // Write a message to the named pipe
    match pipe.write_all(message.as_bytes()) {
        Ok(_) => println!("Message sent to the named pipe."),
        Err(e) => eprintln!("Failed to write to the named pipe: {}", e),
    }
}
```

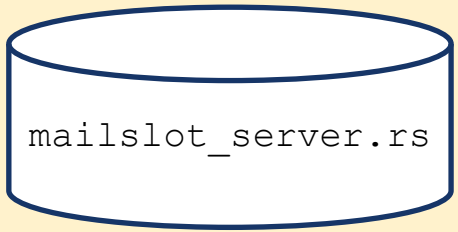
```rust
use std::fs::OpenOptions;
use std::io::{BufRead, BufReader};
use std::path::Path;

fn main() {
    let pipe_path = "/tmp/my_pipe";

    // Check if the named pipe exists
    if !Path::new(pipe_path).exists() {
        eprintln!("Named pipe {} does not exist. Please create it first using mkfifo.", pipe_path);
        return;
    }

    // Open the named pipe for reading
    let pipe = match OpenOptions::new().read(true).open(pipe_path) {
        Ok(pipe) => pipe,
        Err(e) => {
            eprintln!("Failed to open named pipe: {}", e);
            return;
        }
    };

    // Create a buffered reader for the pipe
    let reader = BufReader::new(pipe);

    // Read from the pipe line by line
    for line in reader.lines() {
        match line {
            Ok(content) => println!("Received: {}", content),
            Err(e) => eprintln!("Failed to read from named pipe: {}", e),
        }
    }
}
```

Mail slot

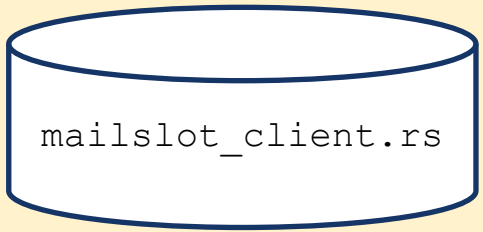
- Una mail slot in Windows è un meccanismo di IPC
- E’ una coda di messaggi unidirezionale, dove il mittente scrive un messaggio che viene letto dal destinatario
- La mail slot viene creata specificando un nome
- Il mittente si identifica attraverso il metodo `MailslotClient`
- Il destinatorio si identifica attraverso il metodo `MailslotServer`

- Note:
  - occorre prima creare il lato destinazione (che si mette in attesa) e poi il lato mittente
  - In `cargo.toml`
    - `[dependencies]`
    - `mail_slot = "0.1.3"`

```rust
use mail_slot::{MailslotServer, MailslotName};

fn main() {
    // Crea un nome per il mailslot (puoi cambiarlo a tuo piacimento)
    let name = MailslotName::local("naive");

    // Crea un server mailslot
    let mut server = MailslotServer::new(&name).unwrap();

    println!("Server is running, waiting for messages...");

    'outer: loop {
        while let Some(msg) = server.get_next_unread().unwrap() {
            let msg_str = String::from_utf8(msg).unwrap();

            if msg_str == "STOP" {
                println!("Stop from client");
                break 'outer;
            }
            println!("message from client {}", msg_str);

        }
    }
}
```



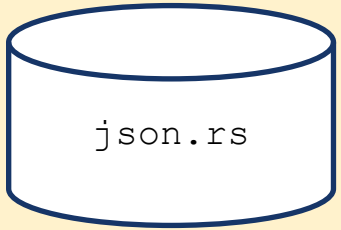
```rust
use mail_slot::{MailslotName, MailslotClient};

fn main() {
    // Crea un nome per il mailslot (puoi cambiarlo a tuo piacimento)
    let name = MailslotName::local("naive");

    // Crea un client mailslot
    let mut client = MailslotClient::new(&name).unwrap();

    // Data to be sent
    let numbers = [42, 256, 1024];
    let text = "Hello, server!";

    // Create the message
    let message = format!("{},{},{},{}", numbers[0], numbers[1], numbers[2], text);

    // Write the message to the mail slot
    client.send_message(message.as_bytes()).unwrap();

    client.send_message(b"STOP").unwrap();
}
```



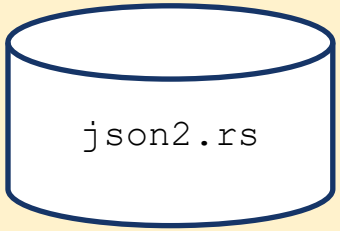
Scambiare messaggi strutturati tra processi in Rust

- Il crate **`serde`** offre le funzionalità necessarie a serializzare e deserializzare buona parte delle strutture dati Rust usando uno qualunque tra i seguenti formati esterni
  - JSON, Bincode, CBOR, YAML, MessagePack, TOML, Pickle, RON, BSON, Avro, JSON5, Postcard, URL query strings, Envy, S-expressions, D-Bus's binary wire format, FlexBuffers, Bencode, DynamoDB Items, Hjson, …
  - Per ciascuno di questi formati esiste un apposito crate che deve essere incluso insieme a quello base
- La libreria estende la macro **`#[derive(Serialize, Deserialize)]`** per aggiungere in modo automatico il supporto alle operazioni di conversione a tipi definiti dall'utente
  - Occorre includere il crate "serde" con la relativa versione e abilitare la funzionalità "derive"
  - `serde = {version = "1.0.137", features = ["derive"]}`

```
use serde::{Serialize, Deserialize};
#[derive(Serialize, Deserialize, Debug)]
struct Point {x: i32, y: i32}
#[derive(Serialize, Deserialize, Debug)]
struct X(i32, i32);
#[derive(Serialize, Deserialize, Debug)]
struct Y(i32);
#[derive(Serialize, Deserialize, Debug)]
struct Z;

fn main() {
    let point = Point { x: 1, y: 2 };
    let x = X(0, 0);
    let y = Y(0);
    let z = Z;

    // Convert the Point to a JSON string.
    let serialized = serde_json::to_string(&point).unwrap();
    println!("serialized = {}", serialized); // Prints serialized = {"x":1,"y":2}
    // Convert the JSON string back to a Point.
    let deserialized: Point= serde_json::from_str(&serialized).unwrap();
    println!("deserialized = {:?}", deserialized); // Prints deserialized = Point { x: 1, y: 2 }

    let serialized = serde_json::to_string(&x).unwrap();
    println!("serialized = {}", serialized); // Prints serialized = [0,0]
    let deserialized: X = serde_json::from_str(&serialized).unwrap();
    println!("deserialized = {:?}", deserialized); // Prints deserialized = X(0, 0)

    let serialized = serde_json::to_string(&y).unwrap();
    println!("serialized = {}", serialized); // Prints serialized = 0
    let deserialized: Y = serde_json::from_str(&serialized).unwrap();
    println!("deserialized = {:?}", deserialized); // Prints deserialized = Y(0)

    let serialized = serde_json::to_string(&z).unwrap();
    println!("serialized = {}", serialized); // Prints serialized = null
    let deserialized: Z = serde_json::from_str(&serialized).unwrap();
    println!("deserialized = {:?}", deserialized); // Prints deserialized = Z
}
```



```rust
use serde::{Serialize, Deserialize};

#[derive(Serialize, Deserialize, Debug)]
enum E {
    W { a: i32, b: i32 },
    X(i32, i32),
    Y(i32),
    Z,
}

fn main() {
    let w = E::W { a: 0, b: 0 }; // Rappresentato come l'oggetto `{"W":{"a":0,"b":0}}`
    let x = E::X(0, 0);          // Rappresentato come l'oggetto `{"X":[0,0]}`
    let y = E::Y(0);             // Rappresentato come l'oggetto `{"Y":0}`
    let z = E::Z;                // Rappresentato come la stringa `"Z"`

    let serialized = serde_json::to_string(&w).unwrap();
    println!("serialized = {}", serialized); // Prints serialized = {"W":{"a":0,"b":0}}
    let deserialized: E = serde_json::from_str(&serialized).unwrap();
    println!("deserialized = {:?}", deserialized); // Prints deserialized = W { a: 0, b: 0 }

    let serialized = serde_json::to_string(&x).unwrap();
    println!("serialized = {}", serialized); // Prints serialized = {"X":[0,0]}
    let deserialized: E = serde_json::from_str(&serialized).unwrap();
    println!("deserialized = {:?}", deserialized); // Prints ddeserialized = X(0, 0)

    let serialized = serde_json::to_string(&y).unwrap();
    println!("serialized = {}", serialized); // Prints serialized = {"Y":0}
    let deserialized: E = serde_json::from_str(&serialized).unwrap();
    println!("deserialized = {:?}", deserialized); // Prints deserialized = Y(0)

    let serialized = serde_json::to_string(&z).unwrap();
    println!("serialized = {}", serialized); // Prints serialized = "Z"
    let deserialized: E = serde_json::from_str(&serialized).unwrap();
    println!("deserialized = {:?}", deserialized); // Prints deserialized = Z
}
```



- E' possibile alterare il comportamento di default, ottenendo rappresentazioni alternative delle enumerazioni, quando occorra interoperare con altri linguaggi
- Si veda https://serde.rs/

```
use crate::SmallPrime::*;
use serde_repr::*;

#[derive(Serialize_repr, Deserialize_repr)]
#[repr(u8)]
enum SmallPrime {
    Two = 2,
    Three,
    Five = 5,
    Seven = 7,
    Eight
}
fn main() {
    let nums: Vec<SmallPrime> = vec![Two, Three, Five, Eight, Seven];

    // Prints [2,3,5,8,7]
    println!("{}", serde_json::to_string(&nums).unwrap());
}
```

Esempio con Mail Slot

- Prendendo spunto dai programmi client/server che comunicano tramite Mail Slot si riportano le versioni che fanno marshalling e unmarshalling per mezzo di `serde`.

```rust
use mail_slot::{MailslotServer, MailslotName};
use serde::{Serialize, Deserialize};

#[derive(Serialize, Deserialize, Debug)]
struct Point {x: i32, y: i32}

fn main() {
    let name = MailslotName::local("naive");

    // Crea un server mailslot
    let mut server = MailslotServer::new(&name).unwrap();

    println!("Server is running, waiting for messages...");

    'outer: loop {
        while let Some(msg) = server.get_next_unread().unwrap() {
            let msg_str = String::from_utf8(msg).unwrap();

            if msg_str == "STOP" {
                println!("Stop from client");
                break 'outer;
            }
            let deserialized: Point= serde_json::from_str(&msg_str).unwrap();
            println!("message from client {}", msg_str);
            println!("deserialized = {:?}", deserialized);
        }
    }
}
```

```rust
use serde::{Serialize, Deserialize};
#[derive(Serialize, Deserialize, Debug)]
struct Point {x: i32, y: i32}

use mail_slot::{MailslotName, MailslotClient};

fn main() {
    let name = MailslotName::local("naive");

    let mut client = MailslotClient::new(&name).unwrap();

    let point1 = Point { x: 1, y: 2 };
    let point2 = Point { x: 2, y: 1 };


    // Convert the Point to a JSON string.
    let serialized = serde_json::to_string(&point1).unwrap();
    println!("serialized = {}", serialized); // Prints serialized = {"x":1,"y":2}

    // Write the message to the mail slot
    client.send_message(serialized.as_bytes()).unwrap();

    let serialized = serde_json::to_string(&point2).unwrap();
    println!("serialized = {}", serialized); // Prints serialized = {"x":2,"y":1}

    // Write the message to the mail slot
    client.send_message(serialized.as_bytes()).unwrap();

    client.send_message(b"STOP").unwrap();
}
```

Pipe

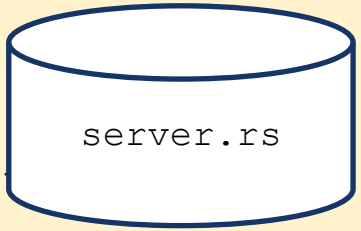
```
use std::process::Command;
use std::io::prelude::*;
use std::process::Stdio;
static PANGRAM: &'static str = "Stringa di prova organizzata su 2 righe\nQuesta e' la seconda riga\n";
fn main() {
    // Spawn the wc command
    let process = match Command::new("wc")
        .stdin(Stdio::piped())
        .stdout(Stdio::piped())
        .spawn() {
        Err(why) => panic!("couldn't spawn wc: {}", why),
        Ok(process) => process,
    };

    // Write a string to the stdin of wc.
    match process.stdin.unwrap().write_all(PANGRAM.as_bytes()) {
        Err(why) => panic!("couldn't write to wc stdin: {}", why),
        Ok(_) => println!("sent my string to wc"),
    }
    // Because stdin does not live after the above calls, it is dropped, and the pipe is closed.
    // This is very important, otherwise wc wouldn't start processing the input we just sent.

    let mut s = String::new();
    match process.stdout.unwrap().read_to_string(&mut s) {
        Err(why) => panic!("couldn't read wc stdout: {}", why),
        Ok(_) => print!("wc responded with:\n{}", s),
    }
}
```

```rust
use std::fs::File;
use std::io::{self, Read, Write};
use std::process::{Command, Stdio};
use std::thread;
fn main() -> io::Result<()> {
    let filename = "test.txt";
    // Creare un thread per leggere il contenuto del file
    let handle = thread::spawn(move || -> io::Result<String> {
        let mut file = File::open(filename)?;
        let mut contents = String::new();
        file.read_to_string(&mut contents)?;
        Ok(contents)
    });
    let mut child = Command::new("rev") // Creare un processo per eseguire il comando rev
        .stdin(Stdio::piped())
        .stdout(Stdio::piped())
        .spawn()
        .expect("Failed to spawn child process");
    // Attendere che il thread finisca e ottenere il contenuto del file
    let file_contents = handle.join().unwrap()?;
    // Ottenere lo stdin del processo e scrivere il contenuto del file
    if let Some(mut stdin) = child.stdin.take() {
        stdin.write_all(file_contents.as_bytes())?;
    }
    // Ottenere lo stdout del processo e leggere l'output
    let output = child.wait_with_output()?;
    println!("Output:\n{}", String::from_utf8_lossy(&output.stdout));
    Ok(())
}
```

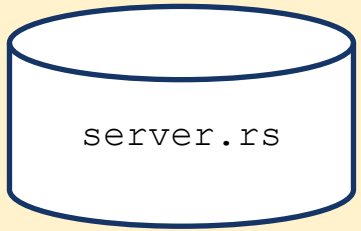
Esempio: client/server

- Il client manda un numero al server per confermare la sua presenza
- Il server risponde con un ACK per avviare l'elaborazione
- Il client manda al server un numero casuale tra 0 e 9
- Il server deve indovinare il numero:
  - se indovina il numero manda al client il messaggio STOP e il client termina,
  - altrimenti manda al client il messaggio CONTINUE e il client fornisce un nuovo numero casuale tra 0 e 9
- Il server attende che il client termini il processo.

```rust
use std::io::{Read, Write};
use std::process::{Command, Stdio};
use rand::Rng;
fn main() {
   let mut child = Command::new("client") // Avvia il processo figlio (client)
        .env("PATH", "./target/debug/")
        .stdin(Stdio::piped())
        .stdout(Stdio::piped())
        .spawn()
        .expect("Failed to start child process");

    let mut rng = rand::thread_rng();

    let mut pipe_reader = child.stdout.take().expect("Failed to open stdout");
    let mut pipe_writer = child.stdin.take().unwrap();

    let mut number_str = [0; 2];
    pipe_reader.read_exact(&mut number_str).expect("Failed to read number");

    println!("Il client è vivo!");

    // Continua
```



```
loop {
        // Invia un acknowledge al client
        pipe_writer.write_all("ACK\n".as_bytes()).expect("Failed to send acknowledge");

        let mut number_str = [0; 2];
        pipe_reader.read_exact(&mut number_str).expect("Failed to read number");
        let number = std::str::from_utf8(&number_str).unwrap().trim();
        println!("Ricevuto dal client:{}", number);
        let random_number = rng.gen_range(0..=9);

        if random_number == number.parse::<i32>().unwrap(){
            pipe_writer.write_all("STOP\n".as_bytes()).expect("Failed to send acknowledge");
            break;
        }

    }
    // Attendere che il processo figlio termini
    child.wait().expect("Failed to wait for child process");
}
```

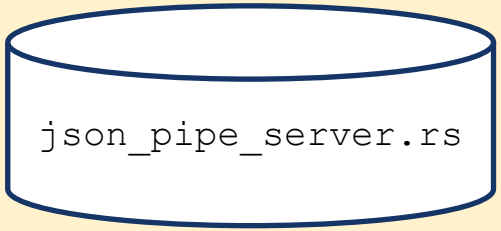


```rust
use std::io;
use rand::Rng;

fn main() {

    let mut rng = rand::thread_rng();

    let random_number = rng.gen_range(0..=9);

    // Scrive il numero casuale nello stdout
    println!("{}", random_number);


    loop {
        let mut response = String::new();
        io::stdin().read_line(&mut response).expect("Failed to read response");

        if response != "ACK\n" {
            break;
        }

        let random_number = rng.gen_range(0..=9);

        // Scrive il numero casuale nello stdout
        println!("{}", random_number);
    }
}
```

Serializzazione/Deserializzazione con Pipe

- Prendendo spunto dai programmi client/server che comunicano tramite pipe si riportano le versioni che fanno marshalling e unmarshalling per mezzo di `serde`.

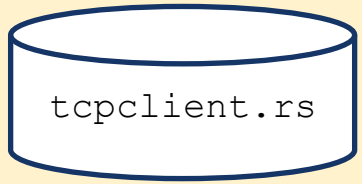
```rust
use serde::{Serialize, Deserialize};
use std::io::Read;
use std::process::{Command, Stdio};

#[derive(Serialize, Deserialize, Debug)]
struct Point {x: i32, y: i32}

fn main() {
    let mut child = Command::new("client") // Avvia il processo figlio (client)
        .env("PATH", "./target/debug/")
        .stdin(Stdio::piped())
        .stdout(Stdio::piped())
        .spawn()
        .expect("Failed to start child process");

    let mut pipe_reader = child.stdout.take().expect("Failed to open stdout");

    let mut s = String::new();
    match pipe_reader.read_to_string(&mut s) {
        Err(why) => panic!("couldn't read client stdout: {}", why),
        Ok(_) => print!("Client responded with:\n{}", s),
    }
    let deserialized: Point= serde_json::from_str(&s).unwrap();
    println!("message from client {}", s);
    println!("deserialized = {:?}", deserialized);

    // Attendere che il processo figlio termini
    child.wait().expect("Failed to wait for child process");
}
```



```rust
use serde::{Serialize, Deserialize};
#[derive(Serialize, Deserialize, Debug)]
struct Point {x: i32, y: i32}

fn main() {

    let point1 = Point { x: 1, y: 2 };

    // Convert the Point to a JSON string.
    let serialized = serde_json::to_string(&point1).unwrap();
    println!("{}", serialized); // Prints serialized = {"x":1,"y":2}

}
```

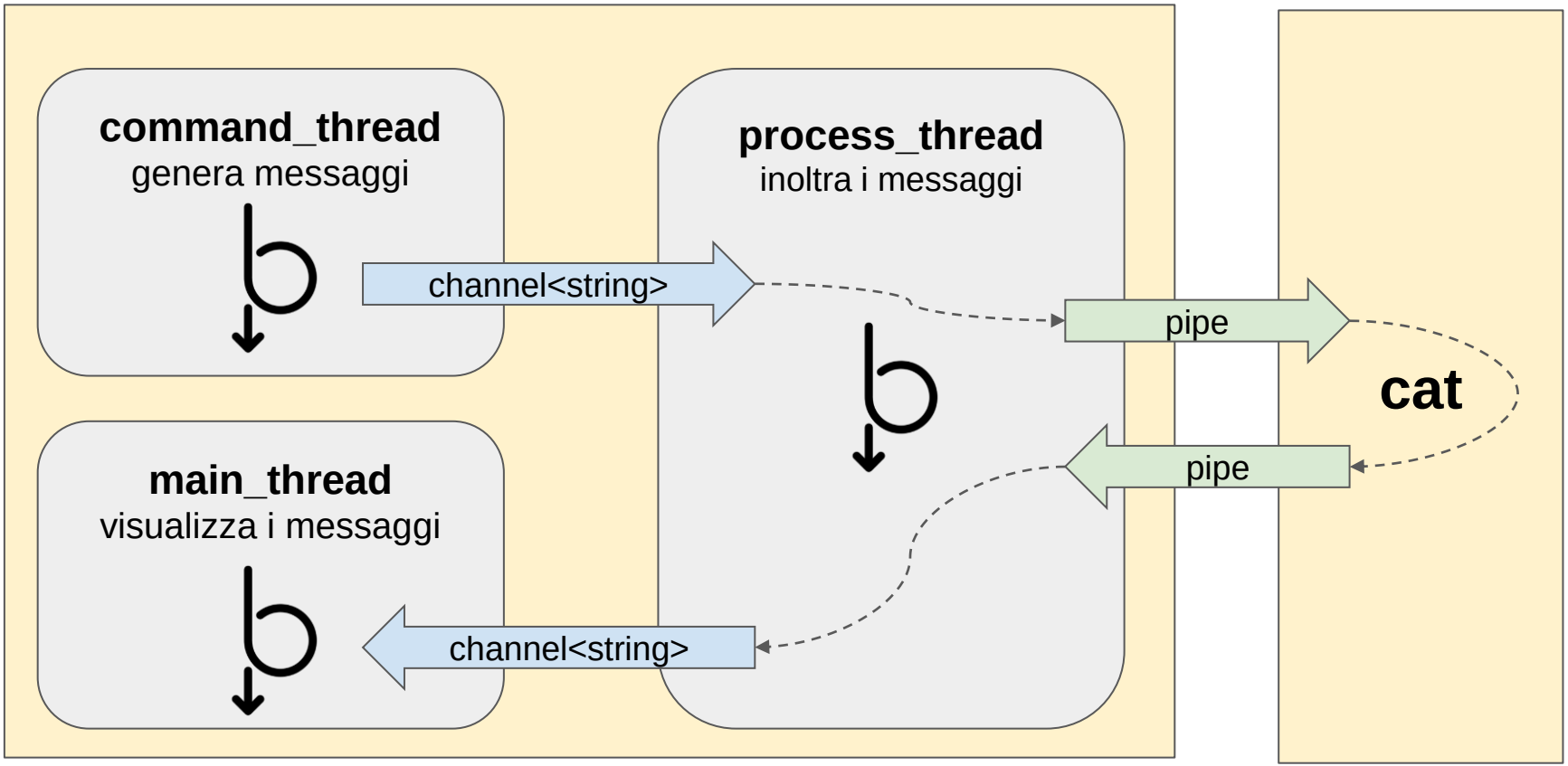
Tcpstream

```
use std::io::{Read, Write};
use std::net::TcpStream;

fn main() {
    let mut stream = TcpStream::connect("127.0.0.1:8080").expect("Errore nella connessione al server");

    let num: i32 = 42; // Numero da inviare al server
    stream.write_all(&num.to_be_bytes()).expect("Errore nell'invio del numero");

    let mut buffer = [0; 4];
    stream.read_exact(&mut buffer).expect("Errore nella lettura della risposta");

    let result = i32::from_be_bytes(buffer);
    println!("Risposta dal server: {}", result);
}
```



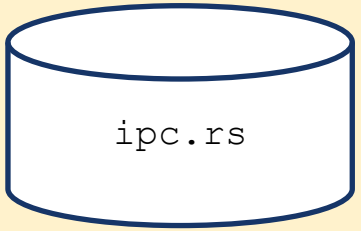
```rust
use std::io::{Read, Write};
use std::net::{TcpListener, TcpStream};

fn handle_client(mut stream: TcpStream) {
    let mut buffer = [0; 4];
    stream.read_exact(&mut buffer).expect("Errore nella lettura del buffer");

    let num = i32::from_be_bytes(buffer);

    println!("Ricevuto {}", num);
    let doubled = num * 2;

    stream.write_all(&doubled.to_be_bytes()).expect("Errore nell'invio della risposta");
}
fn main() {
    let listener = TcpListener::bind("127.0.0.1:8080").expect("Errore nel binding del listener");
    println!("Server in ascolto su 127.0.0.1:8080...");
    for stream in listener.incoming() {
        match stream {
            Ok(stream) => {
                std::thread::spawn(|| {
                    handle_client(stream);
                });
            }
            Err(e) => {
                eprintln!("Errore nell'accettare la connessione: {}", e);
            }
        }
    }
}
```

Comunicazione tra processi



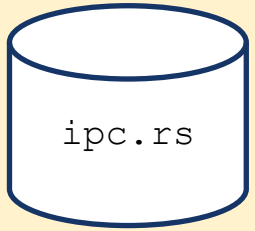
Esempio comunicazione processi

```
use std::io::{BufRead, BufReader, Write};
use std::process::{Command, Stdio};
use std::sync::mpsc::{channel, Receiver, Sender};
use std::thread;
use std::thread::sleep;
use std::time::Duration;

fn start_process(sender: Sender<String>, receiver: Receiver<String>) {

  let child = Command::new("cat")
    .stdin(Stdio::piped())
    .stdout(Stdio::piped())
    .spawn()
    .expect("Failed to start process");

  //continua
```



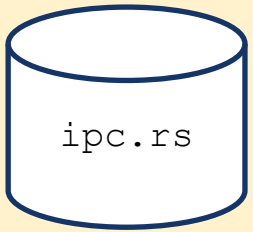
Esempio comunicazione processi

```
  thread::spawn(move || {
        let mut pipe_in = BufReader::new(child.stdout.unwrap());
        let mut pipe_out = child.stdin.unwrap();
        for line in receiver {
            pipe_out.write_all(line.as_bytes()).unwrap();
            let mut buf = String::new();
            match pipe_in.read_line(&mut buf) {
                Ok(_) => {
                    // inoltra quanto ricevuto dalla pipe sul canale di uscita
                    sender.send(buf).unwrap();
                    continue;
                }
                Err(e) => {
                    println!("an error!: {:?}", e);
                    break;
                }
            }
        }
  });
}
```

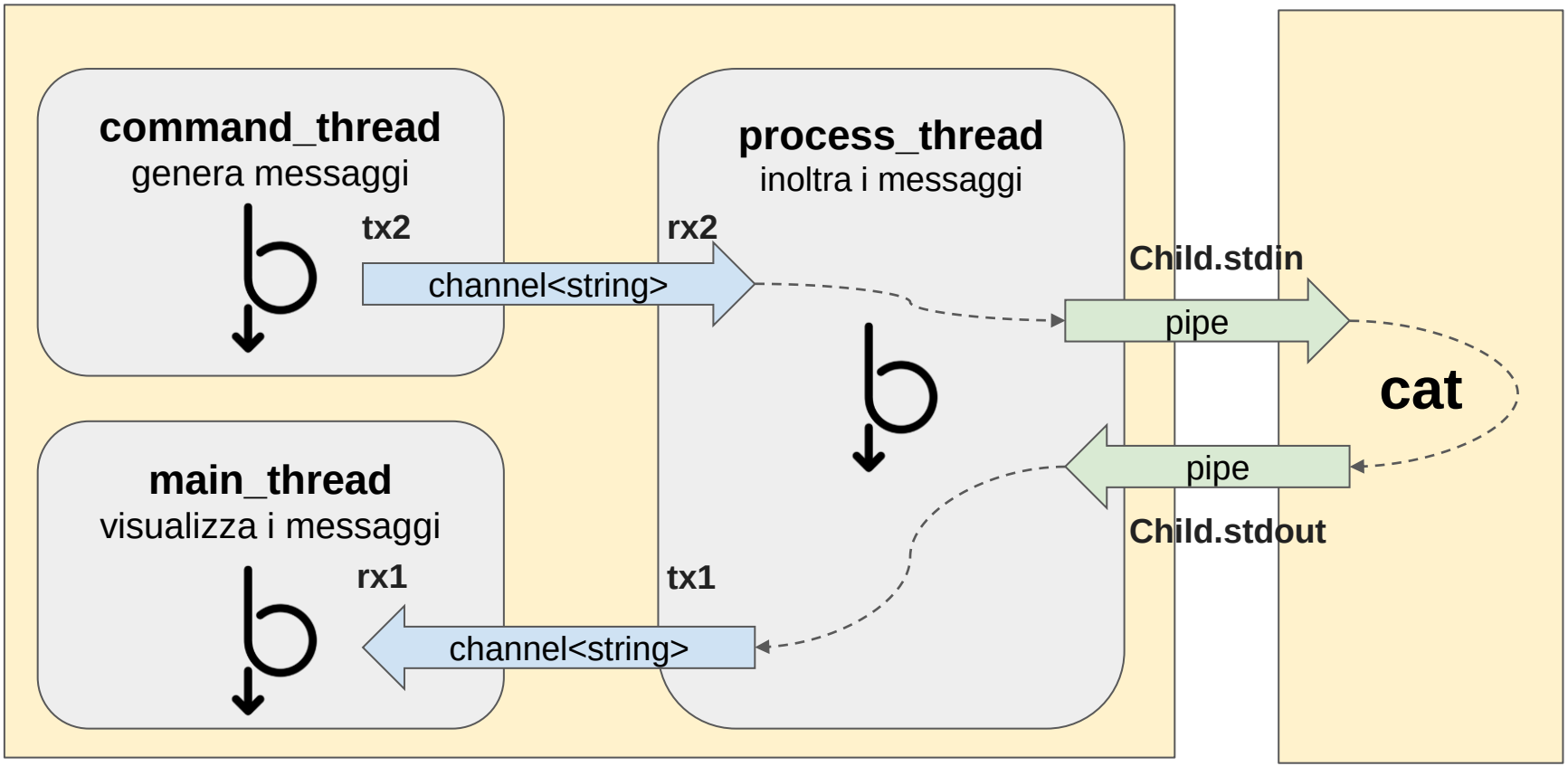


Esempio comunicazione processi

```
fn start_command_thread(sender: Sender<String>) {
  thread::spawn(move || {
    for i in 1..10 {
        sleep(Duration::from_secs(3));
        sender.send(String::from(format!("Message {} from command thread\n", i)))
        .unwrap();
    }
  });
}

fn main() {
  let (tx1, rx1) = channel();
  let (tx2, rx2) = channel();

  start_process(tx1, rx2);

  start_command_thread(tx2);

  rx1.iter().for_each(|line| println!("Echo process response: {}", line))
}
```



Comunicazione tra processi



Comunicazione tra processi

- Il crate **`interprocess`** offre la possibilità di gestire la comunicazione tra processi tramite un'interfaccia univoca multipiattaforma continuando a garantire le funzionalità specifiche dei S.O.
  - Unnamed/named pipes, Posix/C signals, socket
- Il crate **`zbus`** è l'implementazione rust del protocollo D-Bus ed offre una vasta gamma di astrazioni per la comunicazione tra processi
  - Disponibile esclusivamente su piattaforme Linux
  - Ampio supporto alla programmazione asincrona tramite l'utilizzo del Tokio runtime