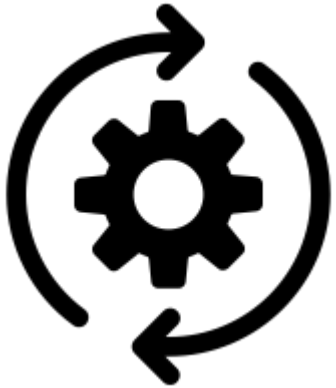


Processi

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

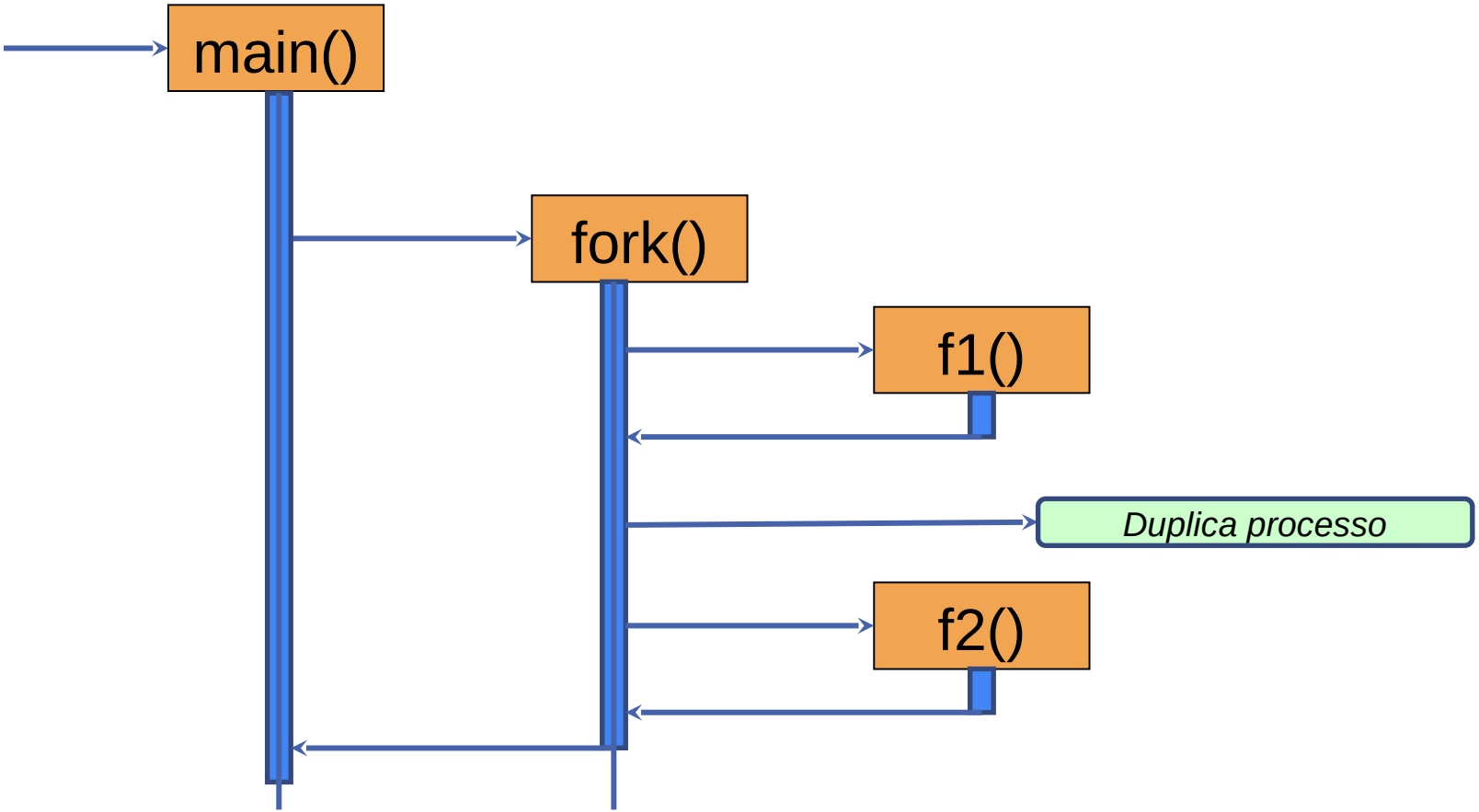
- Nel caso di programmi concorrenti, l'esecuzione di fork() crea un problema
  - Il processo figlio conterrà un solo thread
  - Gli oggetti di sincronizzazione presenti nel padre possono trovarsi in stati incongruenti
- ```
  int pthread_atfork(
      void (*prepare)(void),
      void (*parent)(void),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

Esempio

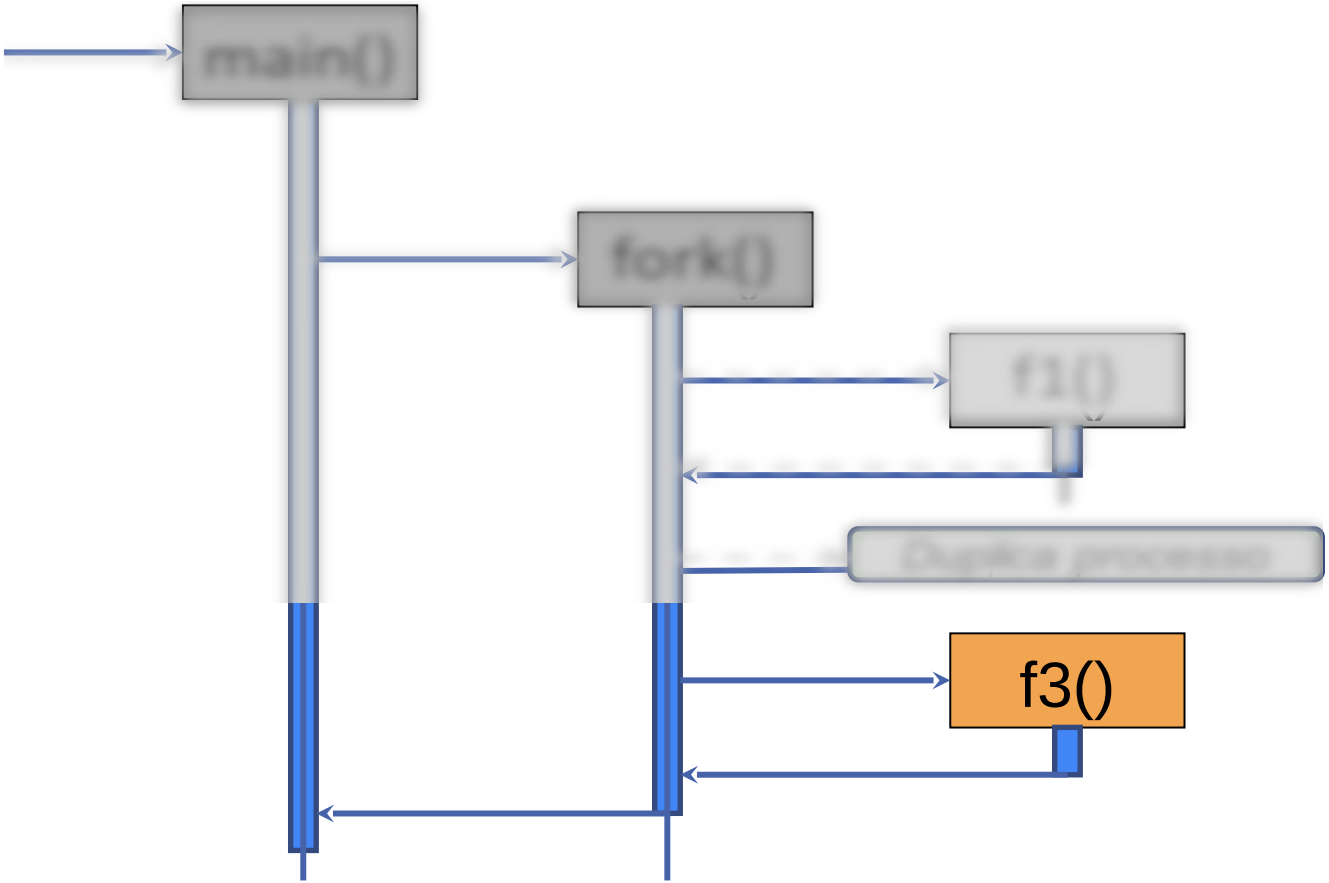
```
void f1() { … }
void f2() { … }
void f3() { … }

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

```
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

Processi in Rust

- E’ possibile modificare la cartella in cui si avvia il processo figlio
  - Tramite il metodo **`current_dir<P: AsRef<Path>>(&mut self, dir: P)`**
- 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*)
- 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

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

Ridirigere i flussi di ingresso/uscita

```
use std::io::Write;
use std::process::{Command, Stdio};

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

let mut stdin = child.stdin.take().expect("Failed to open stdin");
std::thread::spawn(move || {
                            stdin.write_all("Hello, world!".as_bytes())
                                 .expect("Failed to write to stdin");
});

let output = child.wait_with_output().expect("Failed to read stdout");
assert_eq!(String::from_utf8_lossy(&output.stdout), "!dlrow ,olleH");
```

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

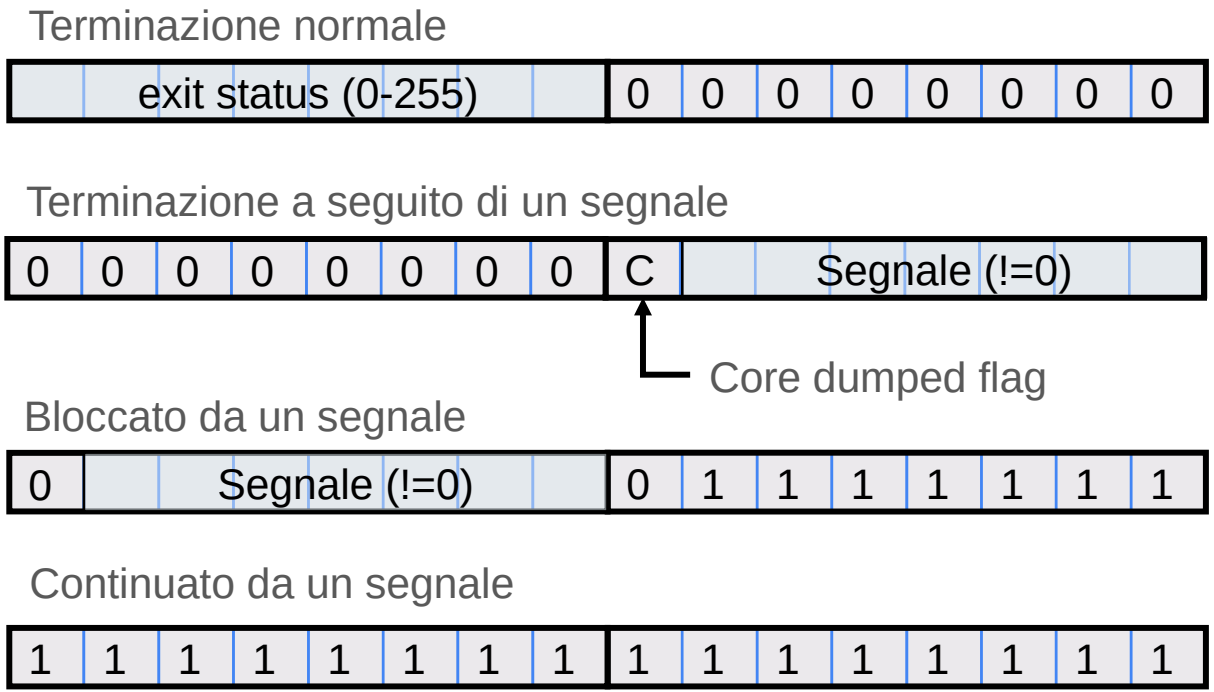
- La funzione **`pid_t wait(int *status)`**, definita in <sys/wait.h>, opera come segue:
  - Se un processo non ha processi figli, la funzione ritorna -1
  - Se nessuno dei figli dell'attuale processo è terminato, la chiamata si blocca in attesa di tale evento
  - In presenza di un figlio terminato:
    - 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
- Il crate sysinfo permette di accedere alle informazioni dei processi così come esposte dai diversi sistemi operativi

```
let mut system = sysinfo::System::new();
system.refresh_all();
let name = system.process(Pid::from(1)).unwrap().name();
println!("Process with id 1 is {}", name);
```

Stato terminale di un processo in Linux



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

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

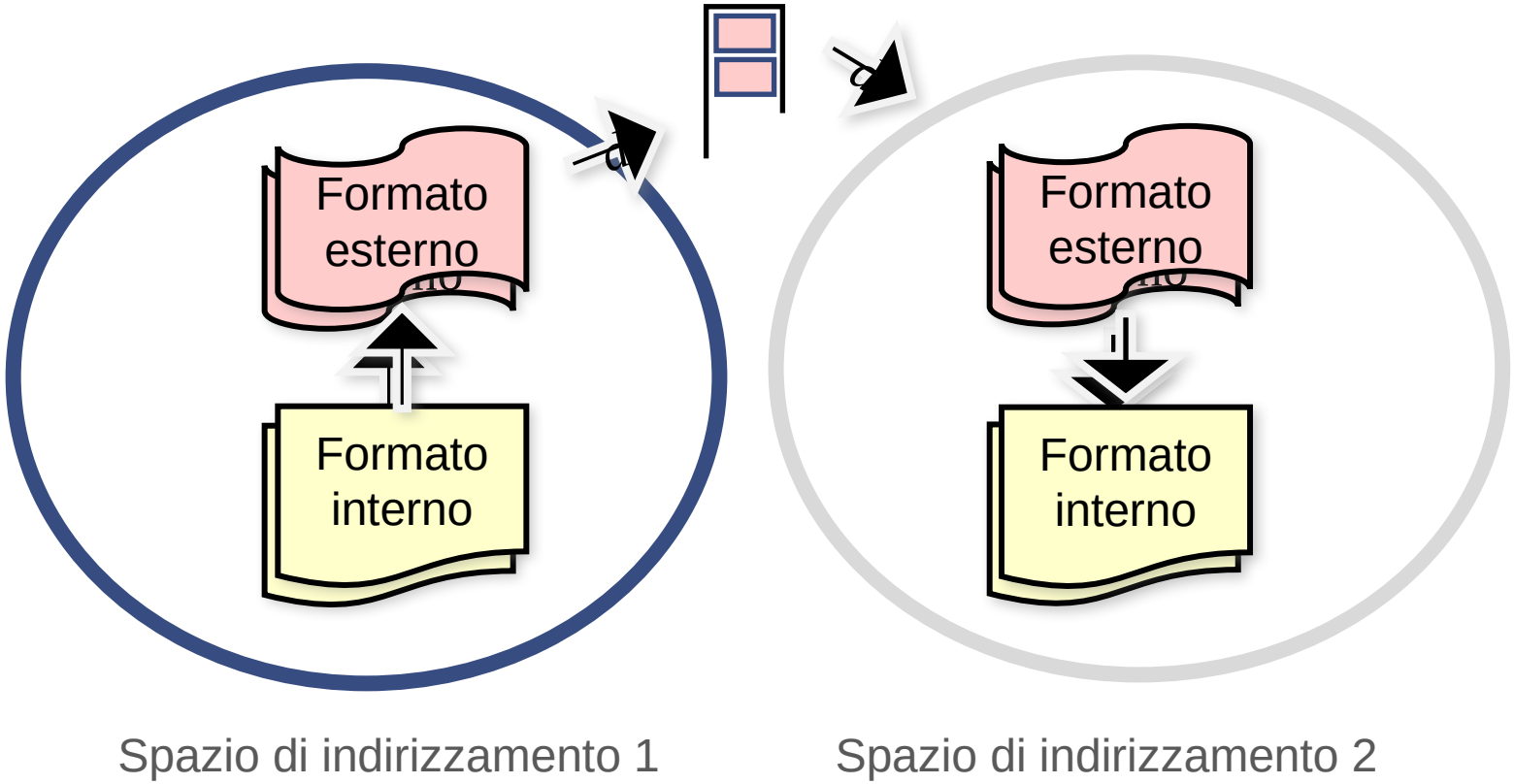
Rappresentazione esterna

- 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)
- Le operazioni di marshalling e unmarshalling possono essere codificate esplicitamente
  - O essere eseguite da codice generato automaticamente dall’ambiente di sviluppo

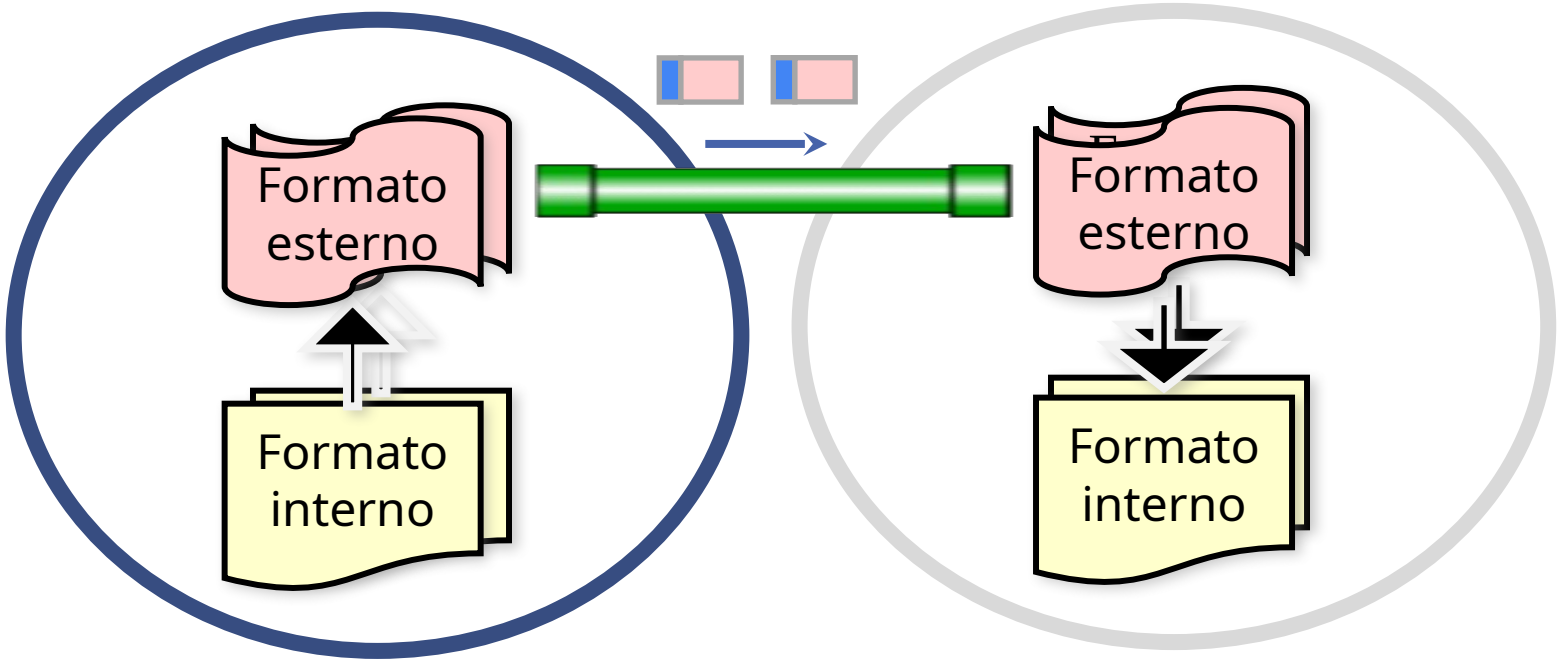
Coda di messaggi



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
- Implementate come buffer all'interno della memoria del kernel
  - Disponibili sia in Linux che Windows con alcune differenze sul modo di renderne disponibili i descrittori ai processi derivati

Pipe


Formato esterno
Formato interno
Formato esterno
Formato interno
Spazio di indirizzamento 1
Spazio di indirizzamento 2

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

```
let echo_child = Command::new("echo")
    .arg("Oh no, a tpyo!")
    .stdout(Stdio::piped())
    .spawn()
    .expect("Failed to start echo process");

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

let mut sed_child = Command::new("sed")
    .arg("s/tpyo/typo/")
    .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");
assert_eq!(b"Oh no, a typo!\n", output.stdout.as_slice());
```

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"]}`**

Regole di serializzazione json

```
struct W {
    a: i32,
    b: i32,
}
let w = W { a: 0, b: 0 }; // Rappresentato come l'oggetto `{"a":0,"b":0}`

struct X(i32, i32);
let x = X(0, 0); // Rappresentato come l'array `[0,0]`

struct Y(i32);
let y = Y(0); // Rappresentato come il solo valore `0`

struct Z;
let z = Z; // Rappresentato come `null`
```

Regole di serializzazione json

```
enum E {
    W { a: i32, b: i32 },
    X(i32, i32),
    Y(i32),
    Z,
}

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"`
```

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

Esempio di serializzazione

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

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

fn main() {
    let point = Point { x: 1, y: 2 };

    let serialized = serde_json::to_string(&point).unwrap();

    println!("{}", serialized); // {"x":1,"y":2}

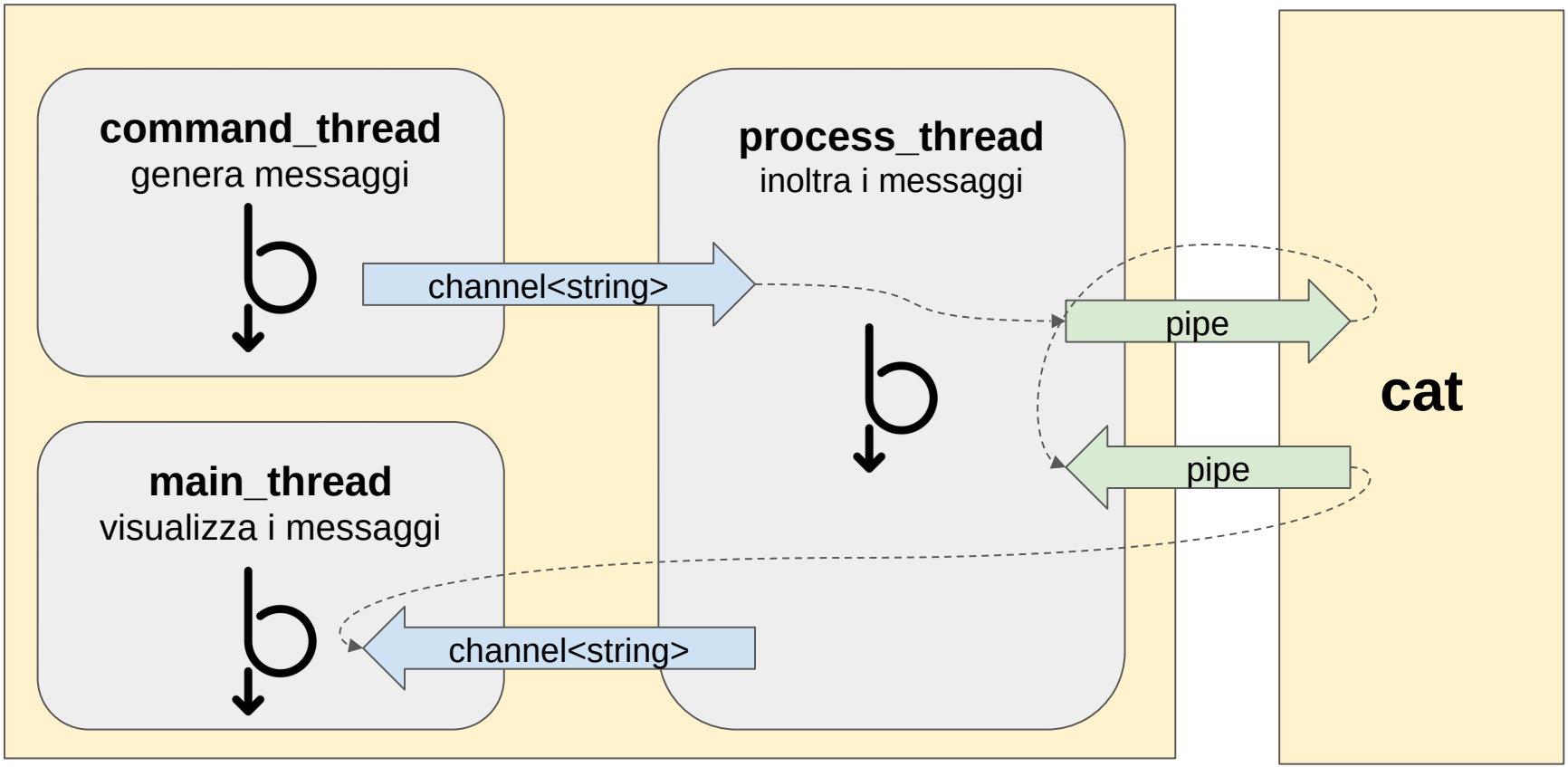
    let deserialized: Point = serde_json::from_str(&serialized).unwrap();

    println!("{:?}", deserialized); // Point { x: 1, y: 2 }
}
```

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/Windows 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

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))
}
```