Alessio Rubicini
Classe 4INC
I.T.T. Montani (A.S. 2019/20)

# SYSTEM V IPC IN PYTHON

**Documentazione ufficiale**: http://semanchuk.com/philip/sysv_ipc/

In Python i sistemi di comunicazione tra processi *UNIX System V* sono utilizzabili attraverso il modulo *sysv_ipc* scaricabile gratuitamente.

Il modulo funziona sia su sistemi originali UNIX quali macOS, Linux, FreeBSD, OpenSolaris 2008.11, HPUX, e AIX che su Windows con strumenti come Cygwin.

**Per scaricare ed installare il modulo**:

1. Aprire il terminale a riga di comando
2. Eseguire il comando

   ```
   sudo python3 -m pip install sysv_ipc
   ```

3. Attendere il termine dell'installazione

Il modulo *sysv_ipc* comprende tre principali sistemi di IPC:

- Memoria condivisa
- Code di messaggi
- Semafori

Altri sistemi come pipe e fifo fanno parte del modulo *os* compreso nella libreria standard di Python.

# CODE DI MESSAGGI

## 1) Creazione/apertura coda

| | |
|---|---|
| **Metodo** | MessageQueue() |
| **Descrizione** | Apre o crea una coda restituendo il descrittore da utilizzare per operare su di essa. |
| **Sintassi** | `coda = MessageQueue(key, flags = 0, mode = 600, max_message_size = QUEUE_MESSAGE_SIZE_MAX_DEFAULT)` |
| **Parametri** | • **Chiave / Key**: chiave identificativa della coda di messaggi.<br>Può essere:<br>• None: il modulo sceglie automaticamente una chiave non utilizzata<br>• IPC_PRIVATE: valore speciale. Implica che la coda sia accessibile solo dal processo che l'ha creata e i suoi figli.<br>• Intero maggiore di 0<br>• **Flag**: per specificare se si desidera creare una coda o aprirne una.<br>- `0` (default): prova ad aprire una coda esistente e restituisce un errore *ExistentialError* se non esiste.<br>- `O_CREAT`: il modulo apre la coda se esiste, se non esiste ne crea una nuova.<br>- `O_CREAT \| O_EXCL` (oppure `O_CREX`): crea una nuova coda, se esiste già una coda con il nome specificato restituisce un errore |

| | |
|---|---|
| | `ExistentialError`.<br>• **Max_messages_size**: definisce la dimensione massima (in byte) di ogni messaggio. |

## 2) Inviare un messaggio sulla coda

| | |
|---|---|
| **Metodo** | Send() |
| **Descrizione** | Invia un messaggio sulla coda specificata. |
| **Sintassi** | `coda.send(messaggio, block = True, type = 1)` |
| **Parametri** | • **Messaggio**: il messaggio da inviare sulla coda<br>• **Block**: specifica se la chiamata `send()` deve aspettare o no nel caso in cui il messaggio non possa essere mandato. Può essere True o False.<br>• **Type**: il tipo associato al messaggio. É rilevante quando si va a ricevere il messaggio. Deve essere maggiore di 0. |

## 3) Ricevere un messaggio dalla coda

| | |
|---|---|
| **Metodo** | Receive() |
| **Descrizione** | Riceve un messaggio dalla coda restituendo una Tuple (messaggio, tipo).<br>****ATTENZIONE****<br>Il messaggio viene ricevuto come un oggetto *bytes*. Occorre quindi decodificarlo utilizzando il metodo *decode* come di seguito: |

| | |
|---|---|
| | `messaggio = messaggio[0].decode(“utf-8”)` |
| **Sintassi** | `messaggio = coda.receive(block = True, type = 1)` |
| **Parametri** | • **Block**: specifica se la chiamata deve aspettare o no nel caso in cui non ci siano messaggi da ricevere. Può essere True o False.<br>• **Type**: il tipo associato al messaggio da ricevere. |

## 4) Rimuovere la coda

| | |
|---|---|
| **Metodo** | remove() |
| **Descrizione** | Rimuove la coda di messaggi. |
| **Sintassi** | `coda.remove()` |
| **Parametri** | - |

# MEMORIA CONDIVISA

## 1) Creare/aprire segmento

| | |
|---|---|
| **Metodo** | SharedMemory() |
| **Descrizione** | Crea o apre un segmento di memoria condivisa restituendo il descrittore per operare su di essa |
| **Sintassi** | `segmento = SharedMemory(key, flags = 0, mode = 600, size = 0)` |
| **Parametri** | • **Chiave/Key**: nome identificativo del segmento di memoria<br>Può essere di tipo: |

| | |
|---|---|
| | - None: il modulo sceglie automaticamente un nome random non ancora utilizzato<br>- IPC_PRIVATE<br>- Intero > 0<br>• **Flag**: specifica se si desidera creare un segmento o aprirne uno.<br>- Flag settati a `0` (default): prova ad aprire un segmento esistente e restituisce un errore se non esiste.<br>- Flag settati a `O_CREAT`: il modulo apre il segmento se esiste, se non esiste ne crea uno nuovo.<br>- Flag settati a `O_CREAT \| O_EXCL` (oppure `O_CREX`): crea un nuovo segmento, se esiste già un segmento con il nome specificato restituisce un errore `ExistentialError`.<br>• **Mode**: specifica i permessi per operare sul segmento.<br>• **Size**: specifica la dimensione del segmento di memoria.<br>Quando si apre un segmento esistente size deve essere minore o uguale alla dimensione del segmento esistente.<br>Quando si crea un nuovo segmento la maggior parte dei sistemi operativi chiede una dimensione maggiore di 0. |

## 2) Agganciare segmento

| | |
|---|---|
| **Metodo** | attach() |
| **Descrizione** | Aggancia il segmento di memoria specificato al processo attuale. |

| | |
|---|---|
| | Il segmento deve necessariamente essere agganciato prima di effettuare operazioni di lettura/scrittura. |
| **Sintassi** | `segmento.attach(address = 0, flags = 0)` |
| **Parametri** | • **address**: indirizzo di memoria a cui agganciare il segmento. Passare None a questo parametro equivale a passare NULL alla shmat().<br>• **Flags**: utilizzati solo se viene agganciato un indirizzo di memoria specifico. |

## 3) Leggere dal segmento

| | |
|---|---|
| **Metodo** | read() |
| **Descrizione** | Legge un certo numero di byte dal segmento di memoria a partire da un *offset*. |
| **Sintassi** | `msg = segmento.read(byte_count = 0, offset = 0)` |
| **Parametri** | • **byte_count**: numero di byte da leggere dal segmento di memoria<br>• **offset**: punto del segmento da cui leggere. Se viene passato offset = 0 legge tutto il buffer. |

## 4) Scrivere sul segmento

| | |
|---|---|
| **Metodo** | write() |
| **Descrizione** | Scrive un messaggio sul segmento di memoria condivisa a partire da offset. |
| **Sintassi** | `segmento.write(msg, offset = 0)` |
| **Parametri** | • **msg**: messaggio da scrivere sul segmento di |

| | |
|---|---|
| | memoria<br>• **offset**: punto del segmento da cui partire a scrivere |

## 5) Sganciare il segmento

| | |
|---|---|
| Metodo | detatch() |
| Descrizione | Sgancia il processo dal segmento di memoria |
| Sintassi | `segmento.detatch()` |
| Parametri | - |

## 6) Eliminare il segmento

| | |
|---|---|
| **Metodo** | remove() |
| **Descrizione** | Distrugge il segmento di memoria specificato. ATTENZIONE: la vera e propria distruzione avviene solo quando tutti i processi si saranno distaccati dal segmento di memoria. |
| **Sintassi** | `segmento.remove()` |
| **Parametri** | - |

# SEMAFORI

## 1) Creazione/apertura semaforo

| | |
|---|---|
| **Metodo** | Semaphore() |
| **Descrizione** | Crea un nuovo semaforo o ne apre uno esistente. |
| **Sintassi** | `sem = Semaphore(key, flags=0, mode = 660, initial_value=0)` |
| **Parametri** | • **Key**: chiave del semaforo. Si applicano le |

| | |
|---|---|
| | stesse regole degli altri sistemi di IPC.<br>• **Flags**: Si applicano le stesse regole degli altri sistemi di IPC. |

## 2) Acquisizione del semaforo

| | |
|---|---|
| **Metodo** | acquire() |
| **Descrizione** | Aspetta finché il valore del semaforo è maggiore di 0 e ritorna decrementando il semaforo. |
| **Sintassi** | `sem = acquire(timeout = None, delta = 1)` |
| **Parametri** | • **Timeout**: se settato di default a None implica nessun limite di tempo. La chiamata non ritornerà finché la condizione di attesa non sarà soddisfatta.<br>Se uguale a 0 la chiamata restituirà *BusyError* se non riuscirà ad acquisire subito il semaforo.<br>Se maggiore di 0 aspetterà n secondi specificati se non può acquisire subito il semaforo dopodiché ritornerà *BusyError*.<br>• Delta: dimensione di cui deve essere decrementato il valore del semaforo |

## 3) Rilascio del semaforo

| | |
|---|---|
| **Metodo** | Release() |
| **Descrizione** | Rilascia (incrementa) il semaforo. |
| **Sintassi** | `sem.release(delta = 1)` |
| **Parametri** | • **Delta**: quantità di cui decrementare il semaforo. |

### 4) Rimozione del semaforo

| **Metodo** | remove() |
|---|---|
| **Descrizione** | Rimuove il semaforo. |
| **Sintassi** | `sem.remove()` |
| **Parametri** | - |

# Pipe e fifo – Modulo OS

I sistemi di IPC pipe e fifo sono integrati nella libreria standard di Python all'interno del modulo os perciò non è necessaria alcuna libreria esterna.

# Pipe

### 1) Creazione pipe

| **Metodo** | pipe() |
|---|---|
| **Descrizione** | Crea una pipe restituendo i due descrittori di lettura e scrittura |
| **Sintassi** | `r, w = os.pipe()` |
| **Parametri** | - |

## 2) Lettura dalla pipe

| | |
|---|---|
| **Metodi** | Apertura: fdopen()<br>Lettura: read()<br>Chiusura: close() |
| **Descrizione** | fdopen = restituisce un oggetto file aperto collegato al descrittore di file specificato.<br>read = legge il contenuto della pipe tramite il descrittore specificato<br>close = chiude il descrittore specificato |
| **Sintassi** | `r = os.fdopen(r)`<br>`msg = r.read()`<br>`os.close(r)` |
| **Parametri** | r = descrittore di lettura restituito precedentemente da pipe() |

## 3) Scrittura sulla pipe

| | |
|---|---|
| **Metodi** | Apertura: fdopen()<br>Scrittura: write()<br>Chiusura: close() |
| **Descrizione** | fdopen = restituisce un oggetto file aperto collegato al descrittore di file specificato.<br>write = scrive il messaggio specificato sulla pipe tramite il descrittore specificato<br>close = chiude il descrittore specificato |
| **Sintassi** | `W = os.fdopen(w, 'w')`<br>`w.write(messaggio)`<br>`os.close(w)` |
| **Parametri** | W = descrittore di scrittura restituito precedentemente da pipe() |

# Fifo

### 1) Creazione della fifo

| | |
|---|---|
| **Metodo** | mkfifo() |
| **Descrizione** | Crea una fifo sul path specificato con i permessi specificati |
| **Sintassi** | `os.mkfifo(path, perm)` |
| **Parametri** | • Path: path della fifo da creare<br>• Perm: permessi della fifo |

### 2) Apertura della fifo

| | |
|---|---|
| **Metodo** | open() |
| **Descrizione** | Apre la fifo dal path specificato nella modalità specificata |
| **Sintassi** | `Fifo = open(path, mode)` |
| **Parametri** | • Path: path della fifo da aprire<br>• Mode: modalità (lettura o scrittura) |

### 3) Lettura dalla fifo

| | |
|---|---|
| **Metodo** | Iterazione sulla fifo tramite ciclo for |
| **Descrizione** | Per leggere il contenuto di una fifo basta iterare su di essa tramite un ciclo for e stampare il contenuto ad ogni ciclo |
| **Sintassi** | `for line in fifo:`<br>`  print(“Messaggio: “, line)` |
| **Parametri** | - |

### 4) Scrittura sulla fifo

| **Metodo** | write() |
|---|---|
| **Descrizione** | Scrive il contenuto specificato sulla fifo |
| **Sintassi** | `fifo.close()` |
| **Parametri** | - |