Programmazione e Architetture (Modulo B)

Lezione 13
IPC e Thread

Luca Manzoni

Interprocess Communication

Comunicare tra processi

- Possiamo creare processi con fork()
- Possiamo far attendere un processo padre per il completamento di un processo figlio tramite wait()
- Però l’interazione tra processi è limitata:
  - Se modifichiamo una variabile questa è modificata solo in un processo (spazi degli indirizzi differenti)
  - Possiamo comunicare solo quando un processo termina
- In realtà esistono una serie di metodi per comunicare tra processi

Interprocess Communication

Metodi per comunicare

- Tra i molti metodi di IPC noi vedremo:
  - **Segnali**. In modo impreciso un “interrupt tra processi”
  - **Pipe**. Un canale di comunicazione tra due processi
  - **Memoria condivisa**. Avere la stessa memoria visibile in due processi distinti
- Esistono altri metodi di comunicazione che però sono simili come struttura (“pipe” più potenti, memoria condivisa con maggiori controlli, etc).

Segnali

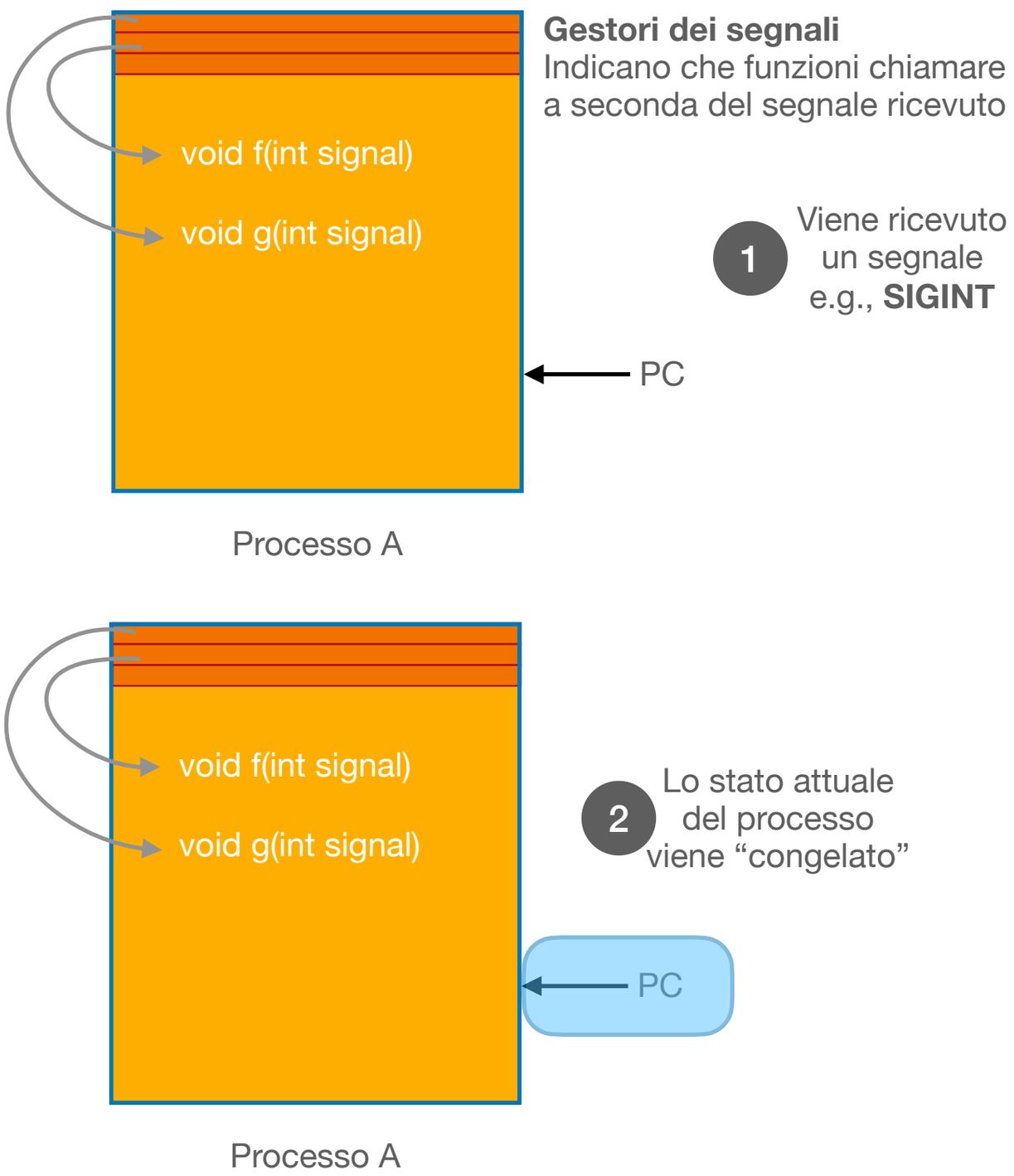
Segnali

“Interrupt” ma per processi

- Un segnale è una forma limitata di comunicazione tra processi
- Quando un processo riceve un segnale il suo stato viene salvato e viene chiamata una funzione dedicata a gestire il segnale
- Al ritorno della funzione l’esecuzione prosegue dove si era interrotta
- Vi è un numero finito di tipologie di segnali
- Per ogni segnale possiamo “registrare” una funzione che lo gestisce
- La funzione che gestisce un segnala ha restrizioni abbastanza forti su quello che può fare

Segnali

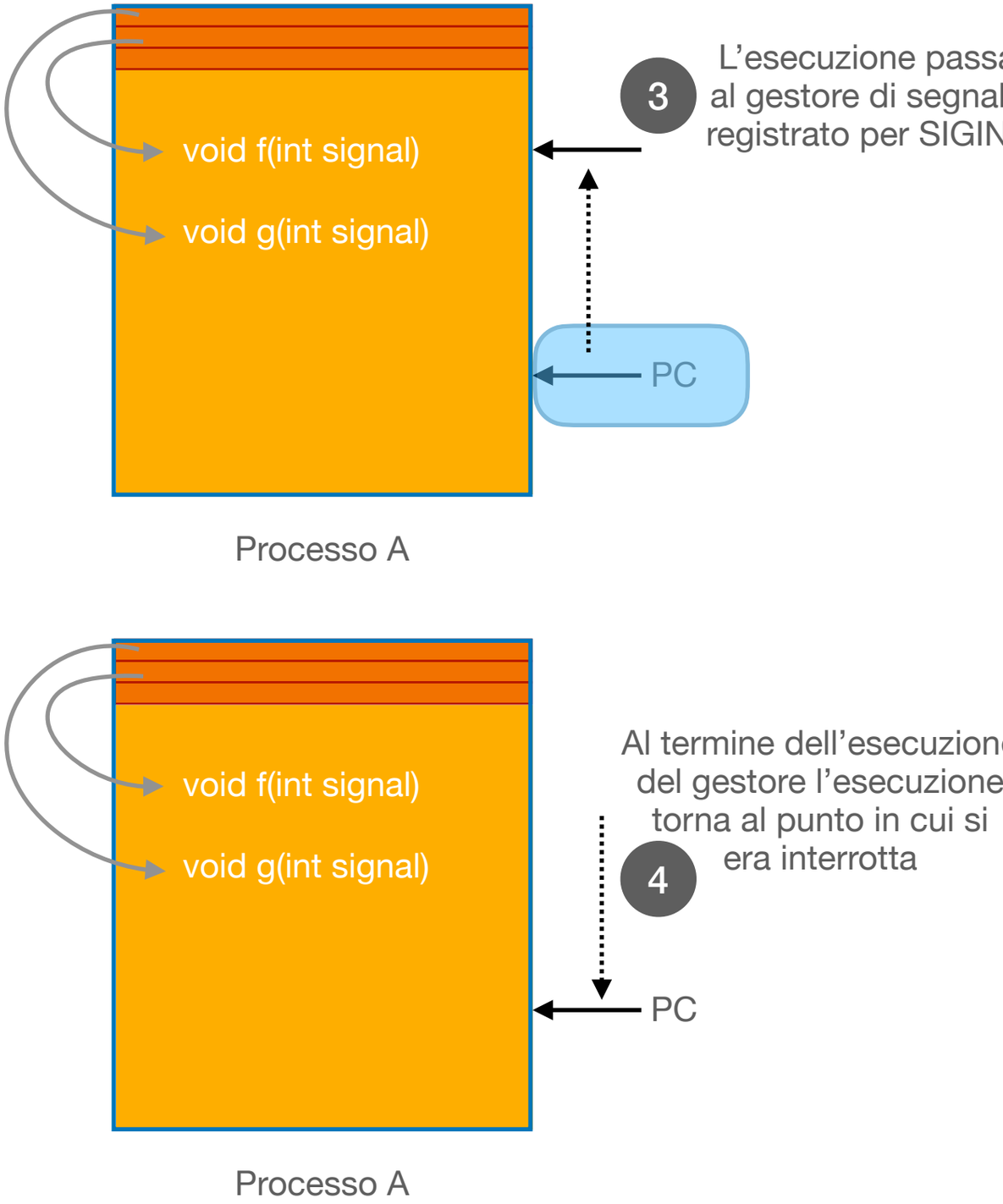
Esempio di ricezione



Gestori dei segnali
Indicano che funzioni chiamare a seconda del segnale ricevuto

1 Viene ricevuto un segnale e.g., **SIGINT**

2 Lo stato attuale del processo viene “congelato”



3 L’esecuzione passa al gestore di segnale registrato per SIGINT

4 Al termine dell’esecuzione del gestore l’esecuzione torna al punto in cui si era interrotta

Registrare un gestore

E limiti del gestore

- La funzione che gestisce un segnale deve avere tipo di ritorno **void** un argomento di tipo **int** (che indica che segnale è stato ricevuto)

- Possiamo registrare una funzione per gestire un segnale con `signal(nome_segnale, nome_funzione)`

- Per esempio `signal(SIGUSR1, gestore_segnale);`

- Viene ritornato `SIGERR` se non è stato possibile impostare il gestore

- Cosa può fare il gestore?

Registrare un gestore

E limiti del gestore

- Dato che il gestore viene chiamato quando viene ricevuto un segnale, interrompendo quello che il processo stava eseguendo, ci sono alcuni restrizioni:
- Solo alcune funzioni possono essere invocate in modo sicuro dal gestore.
  In particolare solo funzioni “reentant”: molteplici invocazioni della stessa funzione possono avvenire in modo concorrente.
  - Niente malloc, printf, etc.
  - Si può usare write()

Registrare un gestore

E limiti del gestore

- Lo stato delle variabili globali non è ben definito a meno che non siano di tipo `volatile sig_atomic_t` (un intero)

- Dato che un segnale può essere ricevuto mentre si sta gestendo un altro segnale ha senso fare poco lavoro dentro un gestore. E.g.,

  - Si rileva che segnale è stato ricevuto e si mette in una coda di segnali ricevuti

  - Il segnale si processa fuori dal gestore (senza avere quindi tutte le restrizioni)

Kill - inviare un segnale

Ma solo processi

- La chiamata di sistema per inviare un segnale a un processo è `kill(pid, nome_segnale)`:
  - pid indica il process id del processo a cui inviare il segnale
  - Il nome del segnale è uno di quelli predefiniti. Per esempio:
    - SIGUSR1 e SIGUSR2. Segnali ad uso dell'utente
    - SIGINT. Segnale di interruzione. Lo stesso che viene inviato premendo Control+C
    - SIGCHLD. Inviato al processo padre quando un figlio termina

Segnali

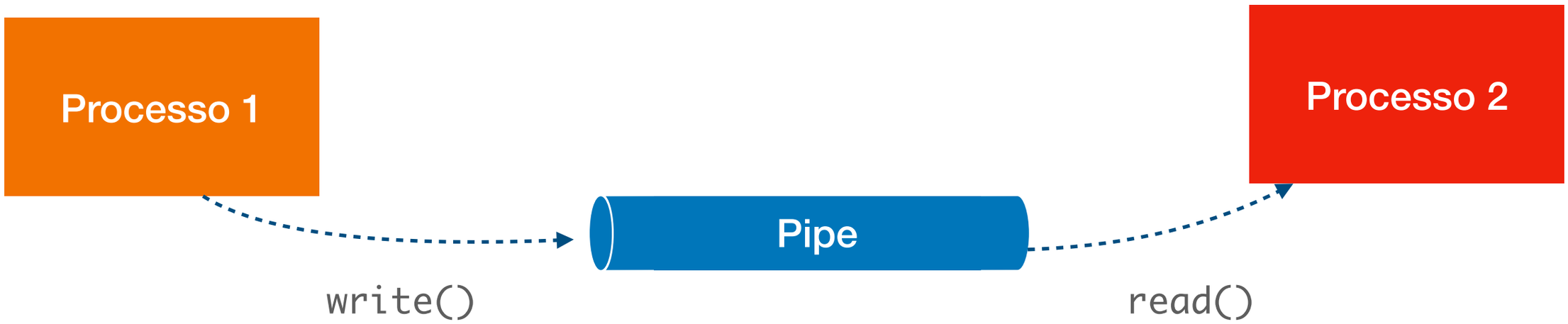
Perché sì e perché no

- Un segnale è una versione “per processi” degli interrupt
- Permette di interrompere un processo (che può quindi reagire “subito”)
- Però non abbiamo controllo su quando ci farebbe comodo ricevere una comunicazione!
- L’informazione che può essere passata è limitata (solo il tipo di segnale)
- Gestire i segnali è complesso ed è facile compiere errori

Pipe

Pipe

Un “tubo” per far passare messaggi



Un processo può scrivere sulla pipe usando la funzione *write…*

…quello che viene scritto sulla pipe può poi venire letto da un altro processo usando la funzione *read*

Read è una funzione bloccante:
il processo rimane sospeso fino
a quando non ha qualcosa da leggere!

Creazione di una pipe

Le due estremità

- Una pipe viene creata con `pipe(int fds[2])`.
  - L'array di due elementi passato come argomento conterrà due file descriptors:
    - In `fds[0]` la parte della pipe da cui si può leggere
    - In `fds[1]` la parte della pipe in cui si può scrivere
- Generalmente un processo crea una pipe e dopo fa fork. Uno dei processi risultanti (e.g., il figlio) userà la pipe in lettura e l'altro (e.g., il padre) userà la pipe in scrittura
- Per comunicazione bidirezionale è comodo usare più pipe

Comunicare con pipe

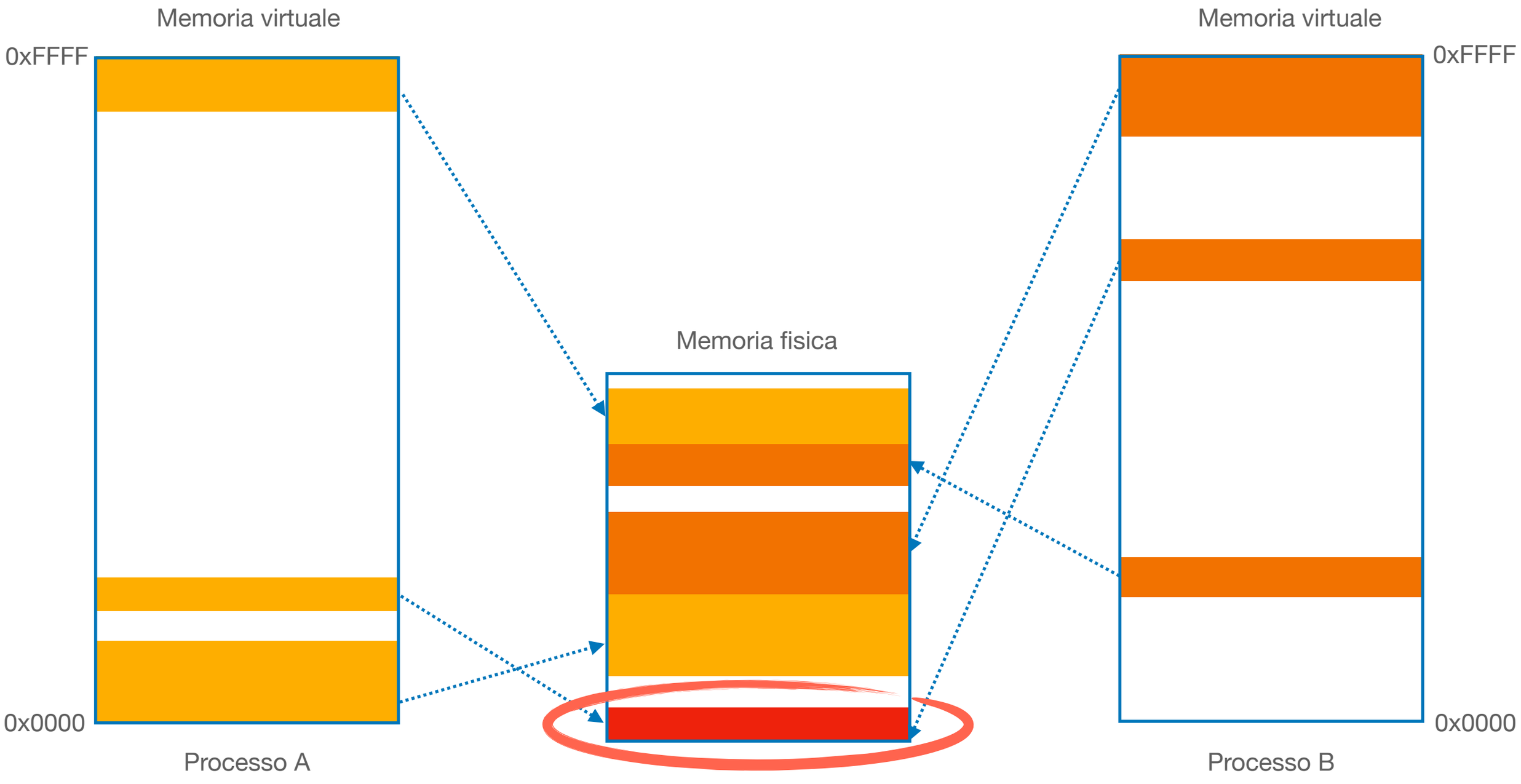
Cooperazione tra processi

- Al contrario dei segnali possiamo inviare messaggi molto più ricchi di informazione
- Il processo che riceve un messaggio non viene interrotto ma deve effettuare una `read()` per ricevere il messaggio
- Ma `read()` è bloccante, quindi se non ci sono messaggi il processo è bloccato fino a quando non ne riceve
- Quindi i due processi devono stabilire un protocollo con cui comunicare
- Esistono altri metodi di comunicazione simili (named pipes, message queues)

Memoria condivisa

Memoria condivisa

Stessa memoria in due processi distinti



Memoria condivisa

Come funziona?

- Perché processi distinti possano individuare che memoria condividere tra loro devono associare un ID
- Serve poi creare un segmento di memoria condiviso (è necessario scegliere la dimensione del segmento e con che permessi)
- Serve poi “attaccare” il segmento di memoria al proprio spazio degli indirizzi
- Ogni processo avrà un puntatore al segmento di memoria che è mappato da più processi (non necessariamente lo stesso indirizzo virtuale!)

Memoria condivisa

Creare un ID

- L'ID in realtà può essere creato in più modi, ma un modo comune è tramite l'uso della funzione ftok()
- `ftok(path_to_file, char)` ritorna una chiave (sperabilmente univoca) identificata dal path e dal carattere forniti come argomento.
- E.g., `ftok("/home/luca/file123", 'r')`
- Il tipo del valore ritornato è `key_t` ed è -1 in caso di errore

Memoria condivisa

Creare (o ottenere) un segmento di memoria

- Per ottenere un segmento di memoria si utilizza `shmget` (**shm** è una comune abbreviazione per **sh**ared **m**emory)
- `shmget(key, dimensione, permessi)` ritorna un ID del segmento di memoria (un intero) e non un puntatore al segmento di memoria (per questo servirà “attaccare” il segmento)
- E.g., `shmget(key, 1000, 0644 | IPC_CREAT)`
- I permessi posso essere espressi in stile unix ed è comune aggiungere (in OR logico) IPC_CREAT per significare che se il segmento non esiste già deve essere creato

Memoria condivisa

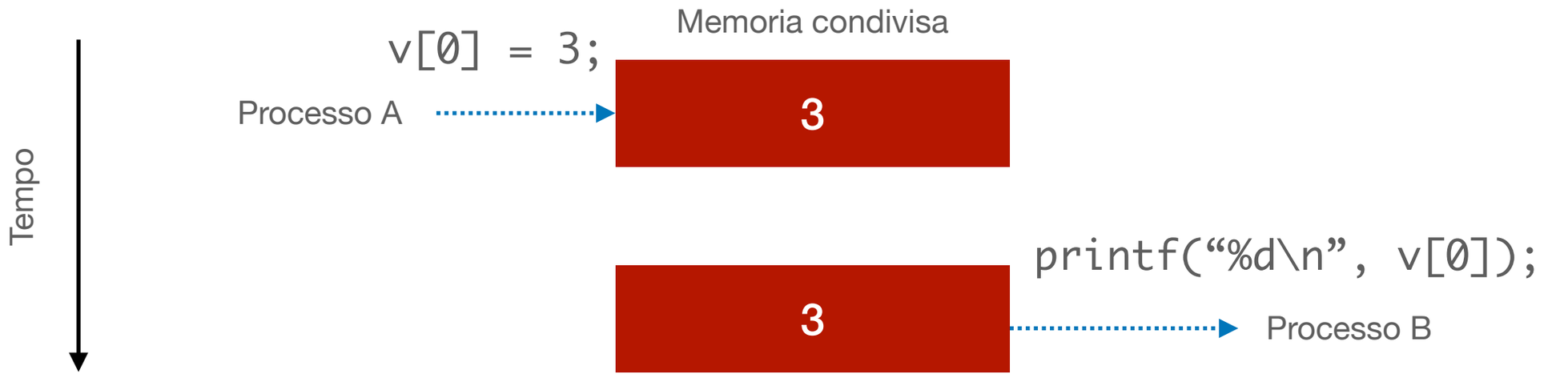
Attaccare un segmento

- Una volta ottenuto un ID di un segmento questo si può "attaccare" allo spazio degli indirizzi del processo con `shmat`
- `shmat(id, indirizzo, flag)` ritorna un puntatore al segmento di memoria
- E.g., `shmat(id,(void *)0, 0)`
- L'indirizzo ha senso sia zero (in quel caso viene scelto dove posizionare il segmento dal sistema operativo)
- I flag posso essere 0 oppure SHM_RDONLY per avere il tutto in sola lettura

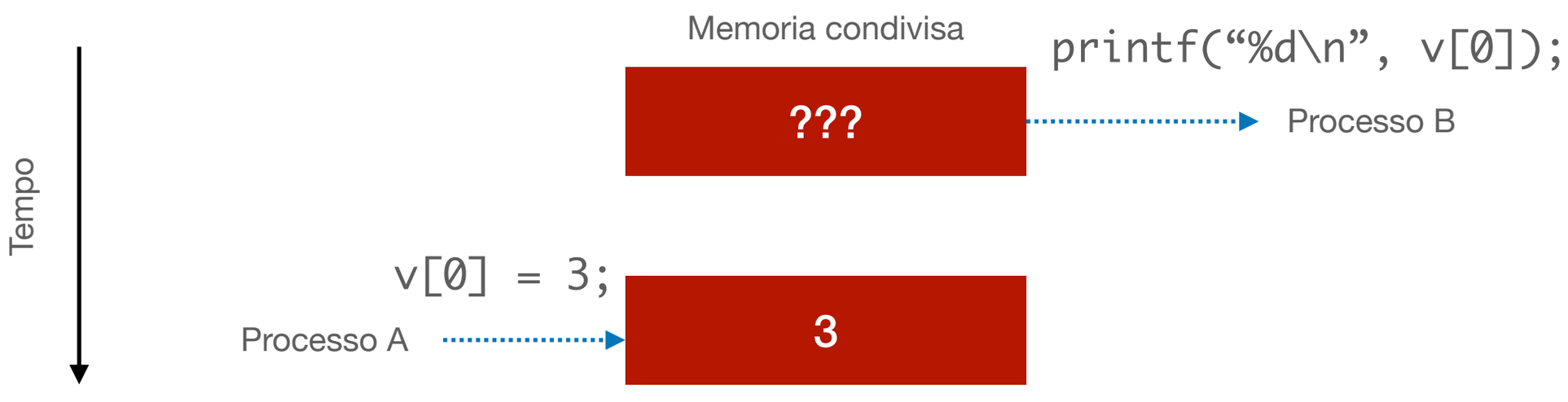
Memoria condivisa

Problemi di coordinazione

Obiettivo: il processo A scrive “3” e il processo B legge il valore scritto



Ma è anche possibile un altro ordine di esecuzione!



Memoria condivisa

Problemi di concorrenza

- Quando una stessa area di memoria è condivisa tra più processi diventa possibile avere dei problemi dovuti all'ordine in cui le operazioni vengono eseguite

- Questi sono problemi di **concorrenza**, che vedremo in futuro

- Esistono una serie di costrutti che ci permettono di forzare un certo ordine di accesso…

- …che però noi vedremo nell'ambito dei thread

Thread

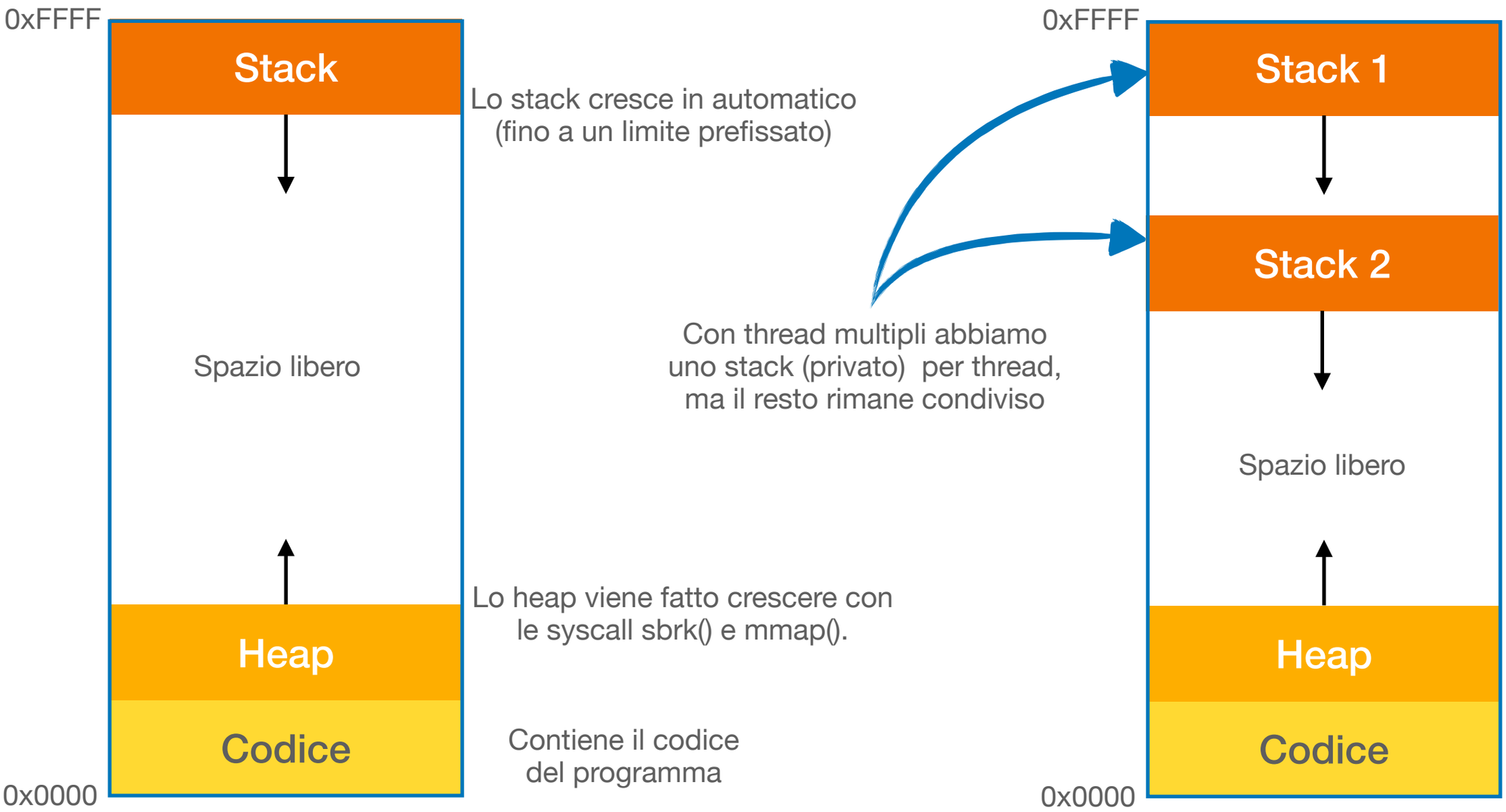
Thread

E differenze rispetto ai processi

- Abbiamo visto che possiamo creare processi con fork()
- Ogni processo ha un suo spazio degli indirizzi, risorse, program counter, etc.
- Però a volte vorremmo lavorare in modo concorrente su qualche struttura dati che viene condivisa.
- Possiamo condividere alcune parti di memoria tra processi diversi…
- …oppure possiamo usare i **thread**, dove la memoria viene condivisa come default.

Memoria di un processo

Versione semplificata



Thread

Cosa sono?

- Nei thread ogni flusso di esecuzione ha una sua area privata (stack, stato dei registri incluso il PC, altre variabili impostate come tali)

- Tutto il resto della memoria è condiviso

- I thread possono essere implementati a livello di sistema operativo (thread nativi) oppure interamente in spazio utente (green thread)

- Questo implica che bisogna gestire i problemi di concorrenza!

- Noi vedremo quali sono i problemi di concorrenza e come implementare e gestire thread (in C coi Posix Thread)